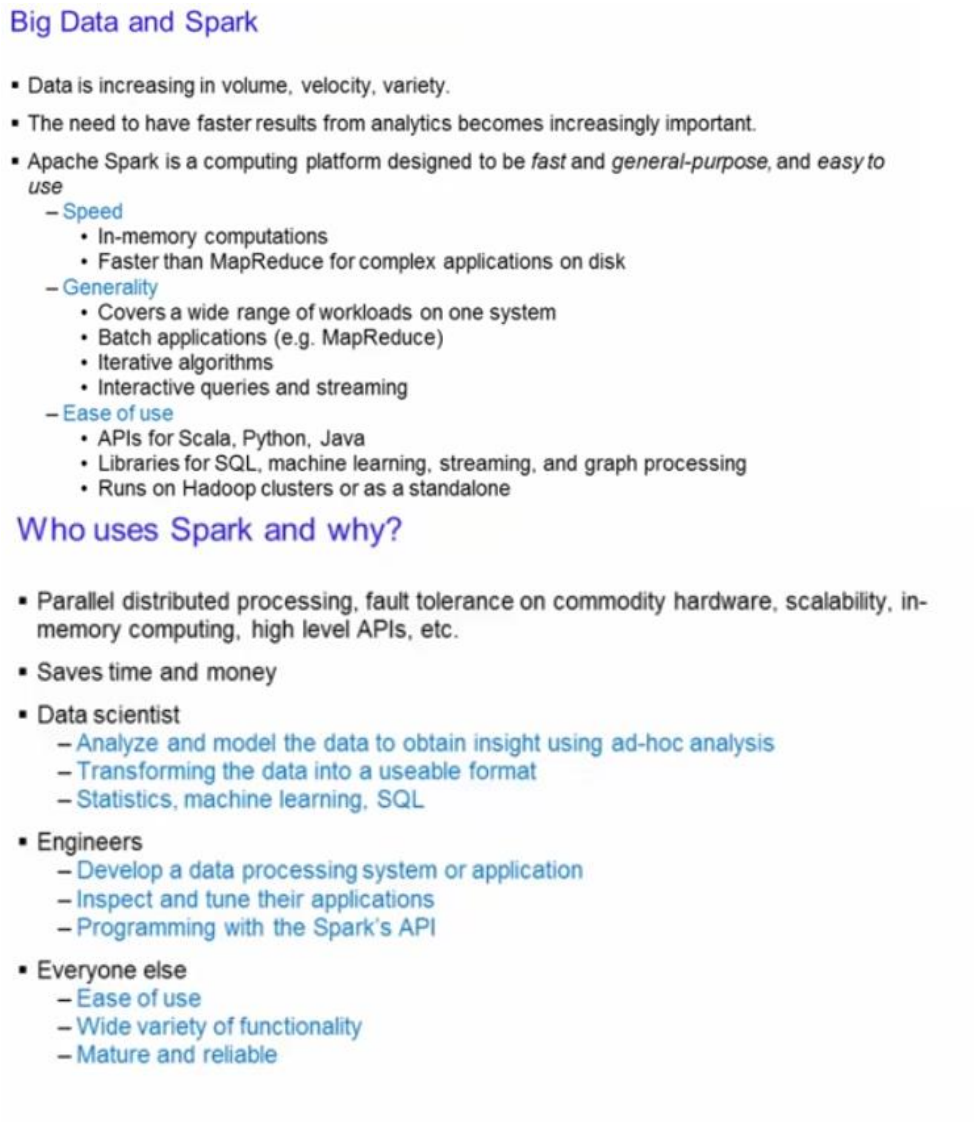
MODULE 1 – INTRODUCTION TO SPARK



Hi - Welcome to Spark Fundamentals. Introduction to Spark.

Objectives: After completing this lesson, you should be

able to explain the purpose of Spark and understand why and when you would use Spark. You should

be able to list and describe the components of the Spark unified stack. You will be able

to understand the basics of the Resilient Distributed Dataset, Spark's primary data abstraction.

Then you will see how to download and install Spark standalone to test it out yourself.

You will get an overview of Scala and Python to prepare for using the two Spark shells.

There is an explosion of data. No matter where

you look, data is everywhere. You get data from social media such as Twitter feeds, Facebook

posts, SMS, and a variety of others. The need to be able to process those data as quickly

as possible becomes more important than ever. How can you find out what your customers want

and be able to offer it to them right away? You do not want to wait hours for a batch

job to complete. You need to have it in minutes or less.

MapReduce has been useful, but the amount of time it takes for the jobs to run is no

longer acceptable in most situations. The learning curve to writing a MapReduce job is also difficult as it takes specific programming knowledge and the know-how. Also, MapReduce jobs only work for a specific set of use cases. You need something that works for a wider set of use cases.

Apache Spark was designed as a computing platform to be fast, general-purpose, and easy to use.

It extends the MapReduce model and takes it to a whole other level.

The speed comes from the in-memory computations. Applications running in memory allows for a much faster processing and response. Spark is even faster than MapReduce for complex applications on disks.

This generality covers a wide range of workloads under one system. You can run batch application such as MapReduce types jobs or iterative algorithms that builds upon each other. You can also run interactive queries and process streaming data with your application. In a later slide, you'll see that there are a number of libraries which you can easily use to expand beyond the basic Spark capabilities.

The ease of use with Spark enables you to quickly pick it up using simple APIs for Scala, Python and Java. As mentioned, there are additional libraries which you can use for SQL, machine learning, streaming, and graph processing. Spark runs on Hadoop clusters such as Hadoop YARN or Apache Mesos, or even as a standalone with its own scheduler.

You may be asking, why would I want to use Spark and what would I use it for? As you know, Spark is related to MapReduce in a sense that it expands on its capabilities.

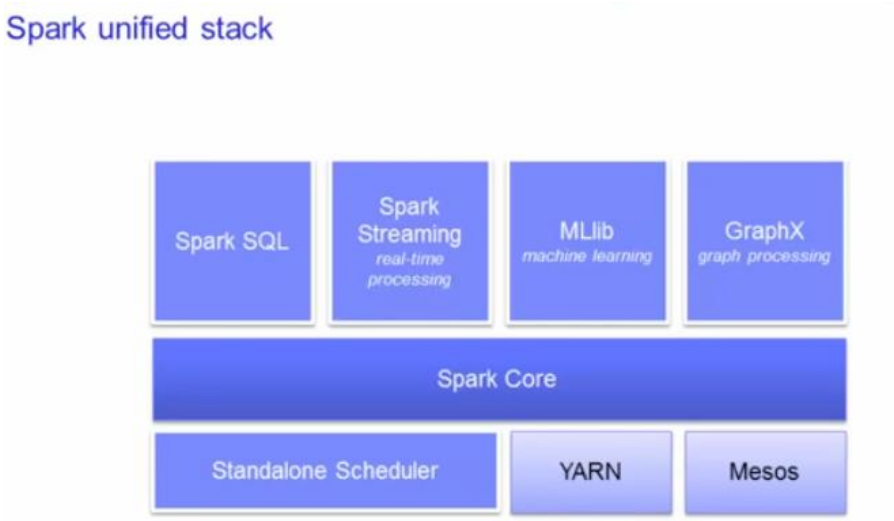
Like MapReduce, Spark provides parallel distributed processing, fault tolerance on commodity hardware, scalability, etc. Spark adds to the concept with aggressively cached in-memory distributed computing, low latency, high level APIs and stack of high level tools described on the next slide. This saves time and money.

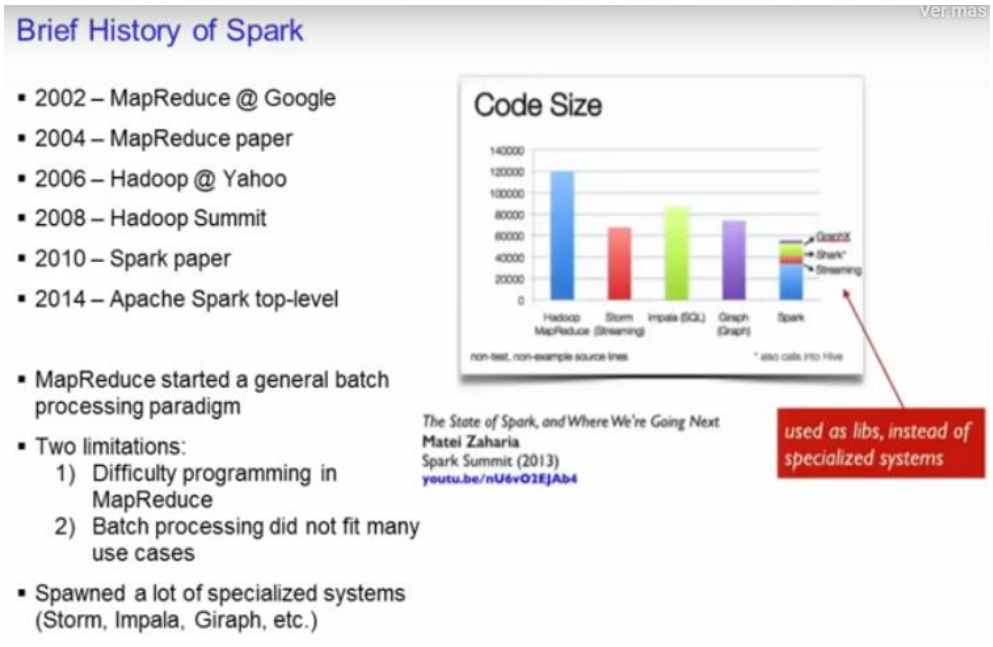
There are two groups that we can consider here who would want to use Spark: Data Scientists and Engineers. You may ask, but aren't they similar? In a sense, yes, they do have overlapping skill sets, but for our purpose, we'll define data scientist as those who need to analyze and model the data to obtain insight. They would have techniques to transform the data into something they can use for data analysis. They will use Spark for its ad-hoc analysis to run interactive queries that will give them results immediately. Data scientists may also have experience using SQL, statistics, machine learning and some programming, usually in Python, MatLab or R. Once the data scientists have obtained insights on the data and later someone determines that there's a need develop a production data processing application, a web application, or some system to act upon the insight, the person called upon to work on it would be the engineers.

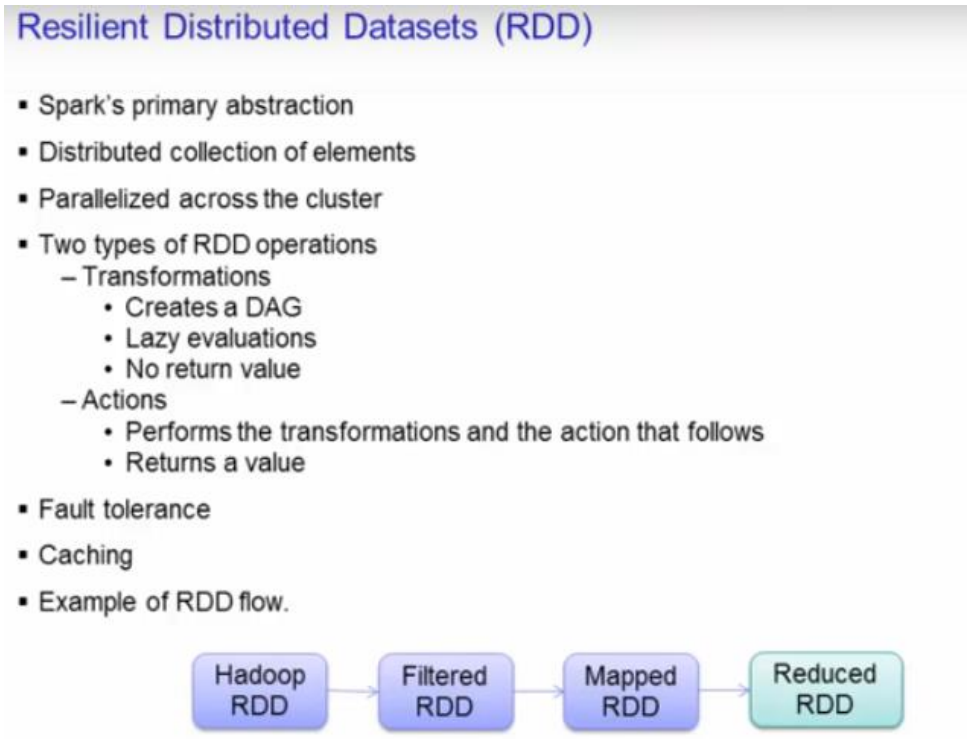
Engineers would use Spark's programming API to develop a system that implement business use cases. Spark parallelize these applications across the clusters while hiding the complexities of distributed systems programming and fault tolerance. Engineers can use Spark to monitor, inspect and tune applications.

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

is mature and reliable.







Here we have a picture of the Spark unified stack. As you can see, the Spark core is at the center of it all. The Spark core is a general-purpose system providing scheduling, distributing, and monitoring of the applications across a cluster. Then you have the components on top of the core that are designed to interoperate closely, letting the users combine them, just like they would any libraries in a software project. The benefit of such a stack is that all the higher layer components will inherit the improvements made at the lower layers. Example: Optimization to the Spark Core will speed up the SQL, the streaming, the machine learning and the graph processing libraries as well. The Spark core is designed to scale up from one to thousands of nodes. It can run over a variety of cluster managers including Hadoop YARN and Apache Mesos. Or simply, it can even run as a standalone with its own built-in scheduler.

Spark SQL is designed to work with the Spark via SQL and HiveQL (a Hive variant of SQL). Spark SQL allows developers to intermix SQL with Spark's programming language supported by Python, Scala, and Java.

Spark Streaming provides processing of live streams of data. The Spark Streaming API closely matches that of the Sparks Core's API, making it easy for developers to move between applications that processes data stored in memory vs arriving in real-time. It also provides the same degree of fault tolerance, throughput, and scalability that the Spark Core provides.

Machine learning, MLlib is the machine learning library that provides multiple types of machine learning algorithms. All of these algorithms are designed to scale out across the cluster as well.

GraphX is a graph processing library with APIs to manipulate graphs and performing graph-parallel computations.

Here's a brief history of Spark. I'm not going to spend too much time on this as you can easily find more information for yourself if you are interested. Basically, you can see that MapReduce started out over a decade ago. MapReduce was designed as a fault tolerant framework that ran on commodity systems. Spark comes out about a decade later with the similar framework to run data processing on commodity systems also using a fault tolerant framework. MapReduce started off as a general batch processing system, but there are two major limitations. 1) Difficulty in programming directly in MR and 2) Batch jobs do not fit many use cases. So this spawned specialized systems to handle other use cases. When you try to combine these third party systems in your applications, there are a lot of overhead.

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

Now let's get into the core of Spark. Spark's primary core abstraction is called Resilient Distributed Dataset or RDD. Essentially it is just a distributed collection of elements that is parallelized across the cluster. You can have two types of RDD operations. Transformations and Actions. Transformations are those that do not return a value. In fact, nothing is evaluated during the definition of these transformation statements. Spark just creates these Direct Acyclic Graphs or DAG, which will only be evaluated at runtime. We call this lazy evaluation.

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

Actions are when the transformations get evaluted along with the action that is called for that RDD. Actions return values. For example, you can do a count on a RDD, to get the number of elements within and that value is returned.

So you have an image of a base RDD shown here on the slide. The first step is loading the dataset from Hadoop. Then uou apply successive transformations on it such as filter, map, or reduce. Nothing actually happens until an action is called. The DAG is just updated each time until an action is called. This provides fault tolerance. For example, let's say a node goes offline. All it needs to do when it comes back online is to re-evaluate the graph to where it left off.

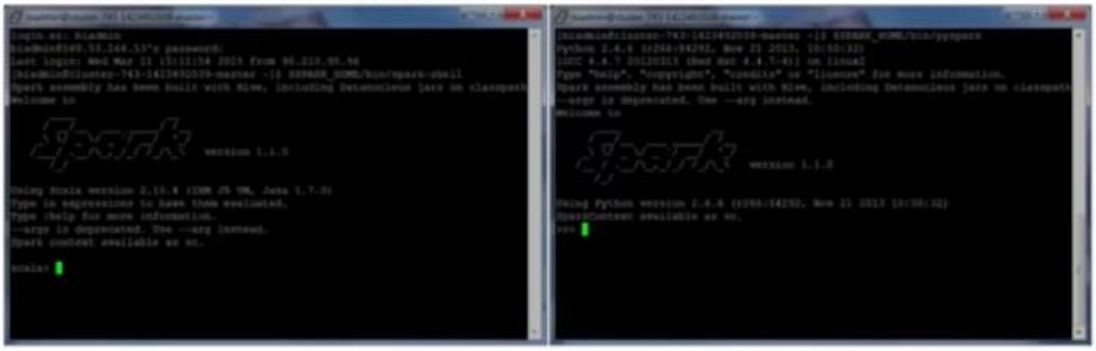
Caching is provided with Spark to enable the processing to happen in memory. If it does not fit in memory, it will spill to disk.

Downloading and installing Spark standalone

- Runs on both Windows and Unit-like systems (e.g Linux, Mac OS)
- To run locally on one machine, all you need is to have Java installed on your system PATH or the JAVA_HOME pointing to a valid Java installation.
- Visit this page to download: http://spark.apache.org/downloads.html
  - Select the Hadoop distribution you require under the "Pre-built packages"
  - Place a compiled version of Spark on each node on the cluster.
- Manually start the cluster by executing:
  - *./sbin/start-master.sh*
- Once started, the master will print out a spark://HOST:PORT URL for itself, which you can use to connect workers to it.
  - The default master's web UI is http://localhost:8080
- Check out Spark's website for more information
  - http://spark.apache.org/docs/latest/spark-standalone.html

Spark jobs and shell

- Spark jobs can be written in Scala, Python, or Java.
- Spark shells for Scala and Python
- APIs are available for all three.
- Must adhere to the appropriate versions for each Spark release.
- Spark's native language is Scala, so it is natural to write Spark applications using Scala.
- The course will cover code examples from Scala, Python and Java.



Brief overview of Scala

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

Scala - anonymous functions

- Functions without a name created for one-time use to pass to another function
- Left side of the right arrow => is where the argument resides (no arguments in the example)
- Right side of the arrow is the body of the function (the println statement)

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

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

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

Spark runs on Windows or Unix-like systems. The lab exercises that are part of this course will be running with IBM BigInsights. The information on this slide shows you how you can install the standalone version yourself with all the default values that come along with it.

To install Spark, simply download the pre-built version of Spark and place a compiled version of Spark on each node of your cluster. Then you can manually start the cluster by executing ./sbin/start-master.sh . In the lab, the start script is different, and you will see this in the lab exercise guide.

Again, the default URL to the master UI is on port 8080. The lab exercise environment will be different and you will see this in the guide as well.

You can check out Spark's website for more information:

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

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

Spark itself is written in the Scala language, so it is natural to use Scala to write Spark applications. This course will cover code examples from by Scala, Python, and Java.

Java 8 actually supports the functional programming style to include lambdas, which concisely captures the functionality that are executed by the Spark engine. This bridges the gap between Java and Scala for developing applications on Spark. Java 6 and 7 is supported, but would require more work and an additional library to get the same amount of functionality as you would using Scala or Python.

Here we have a brief overview of Scala. Everything in Scala is an object. The primitive types that is defined by Java such as int or boolean are objects in Scala. Functions are objects in Scala and will play an important role in how applications are written for Spark.

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

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

This slide and the next is just to serve as a very brief overview of Scala. If you wish to learn more about Scala, check out its website for tutorials and guide. Throughout this course, you will see examples in Scala that will have explanations on what it does. Remember, the focus of this course is on the context of Spark. It is not intended to teach Scala, Python or Java.

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

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

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

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

Having completed this lesson, you should now understand what Spark is all about and why you would want to use it. You should be able to list and describe the components in the Spark stack as well as understand the basics of Spark's primary abstraction, the RDDs.

You also saw how to download and install Spark's standalone if you wish to, or you can use the provided lab environment. You got a brief overview of Scala and saw how to launch and use the two Spark Shells.

You have completed this lesson. Go on to the first lab exercise and then proceed to the next lesson in this course.

>> Lab:

Spark is built around speed and the ease of use. In these labs you will see for yourself how easy it is to get started using Spark.

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset or RDD. In a subsequent lab exercise, you will learn more about the details of RDD. RDDs have actions, which return values, and transformations, which return pointers to new RDD.

This set of labs uses Skills Network (SN) Labs to provide an interactive environment to develop applications and analyze data. It is available in either Scala or Python shells. Scala runs on the Java VM and is thus a good way to use existing Java libraries. In this lab exercise, we will set up our environment in preparation for the later labs.

Run the following cells to get the lab data.

[1]:
```
# download the data from the IBM server
# this may take ~30 seconds depending on your internet speed
!wget --quiet https://cocl.us/BD0211EN_Data
print("Data Downloaded!")
```

```
Data Downloaded!
```

Let's unzip the data that we just downloaded into a directory dedicated for this course. Let's choose the directory **/resources/jupyter/labs/BD0211EN/**.

[*]:
```
# this may take ~30 seconds depending on your internet speed
!unzip -q -o -d /resources/jupyterlab/labs/BD0211EN/ BD0211EN_Data
print("Data Extracted!")
```

The data is in a folder called **LabData**. Let's list all the files in the data that we just downloaded and extracted.

[ ]:
```
# list the extracted files
!ls -1 /resources/jupyterlab/labs/BD0211EN/LabData
```

Should have:

- followers.txt
- notebook.log
- nyctaxi100.csv
- nyctaxi.csv
- nyctaxisub.csv
- nycweather.csv
- pom.xml
- README.md
- taxistreams.py
- users.txt

Starting with Spark

Let's first import the tools that we need to use Spark in this SN Labs.

```
!pip install findspark
!pip install pyspark
import findspark
import pyspark
findspark.init()
sc = pyspark.SparkContext.getOrCreate()
```

```
Collecting findspark
  Downloading https://files.pythonhosted.org/packages/b1/c8/e6e1f6
3-none-any.whl
Installing collected packages: findspark
Successfully installed findspark-1.3.0
```

The notebooks provide code assist. For example, type in "sc." followed by th

Add in the path to the *README.md* file in **LabData**.

[5]:
```
readme = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/README.md")
```

Let's perform some RDD actions on this text file. Count the number of items in the RDD using this command:

[ ]:
```
readme.count()
```

You should see that this RDD action returned a value of 103.

Let's run another action. Run this command to find the first item in the RDD:

[ ]:
```
readme.first()
```

[14]:
```
'# Apache Spark'
```

Now let's try a transformation. Use the filter transformation to return a new RDD with a subset of the items in the file. Type in this command:

[15]:
```
linesWithSpark = readme.filter(lambda line: "Spark" in line)
```

You can even chain together transformations and actions. To find out how many lines contains the word "Spark", type in:

[16]:
```
linesWithSpark = readme.filter(lambda line: "Spark" in line)
readme.filter(lambda line: "Spark" in line).count()
```

[16]:
```
18
```

[17]:
```
readme.map(lambda line: len(line.split())).reduce(lambda a, b: a if (a > b) else b)
```

[17]:
```
14
```

There are two parts to this. The first maps a line to an integer value, the number of words in that line. In the second part reduce is called to find the line with the most words in it. The arguments to map and reduce are Python anonymous functions (lambdas), but you can use any top level Python functions. In the next step, you'll define a max function to illustrate this feature.

Define the max function. You will need to type this in:

[18]:
```
def max(a, b):
 if a > b:
    return a
 else:
    return b
```

Now run the following with the max function:

[19]:
```
readme.map(lambda line: len(line.split())).reduce(max)
```

[19]:
```
14
```

Spark has a MapReduce data flow pattern. We can use this to do a word count on the readme file.

[20]:
```
wordCounts = readme.flatMap(lambda line: line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)
```

Here we combined the flatMap, map, and the reduceByKey functions to do a word count of each word in the readme file.

To collect the word counts, use the *collect* action.

```
[21]: wordCounts.collect()
```

```
[21]: [('#', 1),
       ('Apache', 1),
       ('Spark', 14),
       ('is', 6),
       ('It', 2),
       ('provides', 1),
       ('high-level', 1),
       ('APIs', 1),
       ('in', 5),
       ('Scala,', 1),
       ('Java,', 1),
       ('an', 3),
       ('optimized', 1),
```

Using Spark caching

In this short section, you'll see how Spark caching can be used to pull data sets into a cluster-wide in-memory cache. This is very useful for accessing repeated data, such as querying a small "hot" dataset or when running an iterative algorithm. Both Python and Scala use the same commands.

As a simple example, let's mark our linesWithSpark dataset to be cached and then invoke the first count operation to tell Spark to cache it. Remember that transformation operations such as cache does not get processed until some action like count() is called. Once you run the second count() operation, you should notice a small increase in speed.

```
2]: print(linesWithSpark.count())
```

```
18
```

```
3]: from timeit import Timer
    def count():
        return linesWithSpark.count()
    t = Timer(lambda: count())
```

```
*]: print(t.timeit(number=50))
```

```
12.572217341978103
```

It may seem silly to cache such a small file, but for larger data sets across tens or hundreds of nodes, this would still work. The second linesWithSpark.count() action runs against the cache and would perform significantly better for large datasets.

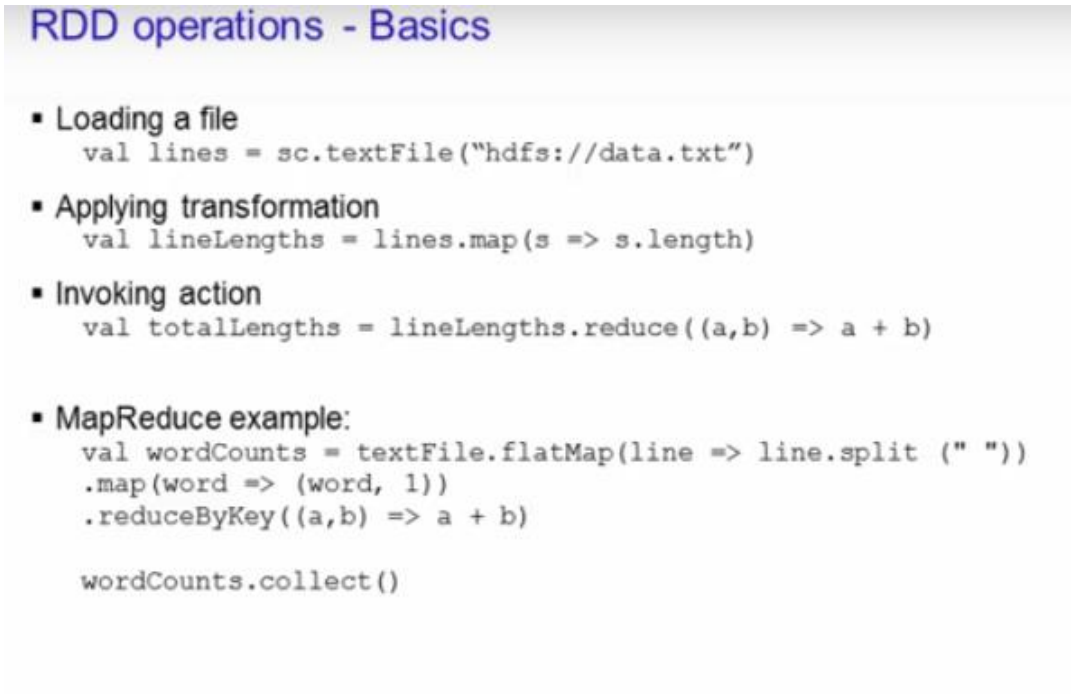
MODULE 2- RESILIENT DISTRIBUTED DATASET AND DATAFRAMES

Resilient Distributed Dataset (RDD)

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

Creating an RDD

- Launch the Spark shell
  ./bin/spark-shell
- Create some data
  val data = 1 to 10000
- Parallelize that data (creating the RDD)
  val distData = sc.parallelize(data)
- Perform additional transformations or invoke an action on it.
  distData.filter(…)
- Alternatively, create an RDD from an external dataset
  - val readmeFile = sc.textFile("Readme.md")



Hi. Welcome to the Spark Fundamentals course. This lesson will cover Resilient Distributed Dataset.

After completing this lesson, you should be able to understand and describe Spark's primary data abstraction, the RDD. You should know how to create parallelized collections from internal and external datasets. You should be able to use RDD opeartions such as Transformations and Actions. Finally, I will also show you how to take advantage of Spark's shared variables and key-value pairs.

Resilient Distributed Dataset (RDD) is Spark's primary abstraction. RDD is a fault tolerant collection of elements that can be parallelized. In other words, they can be made to be operated on in parallel. They are also immutable. These are the fundamental primary units of data in Spark.

When RDDs are created, a direct acyclic graph (DAG) is created. This type of operation is called transformations. Transformations makes updates to that graph, but nothing actually happens until some action is called. Actions are another type of operations. We'll talk more about this shortly. The notion here is that the graphs can be replayed on nodes that need to get back to the state it was before it went offline - thus providing fault tolerance.

The elements of the RDD can be operated on in parallel across the cluster. Remember, transformations return a pointer to the RDD created and actions return values that comes from the action.

There are three methods for creating a RDD. You can parallelize an existing collection. This means that the data already resides within Spark and can now be operated on in parallel.

As an example, if you have an array of data, you can create a RDD out of it by calling the parallelized method. This method returns a pointer to the RDD. So this new distributed dataset can now be operated upon in parallel throughout the cluster.

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

The third method to create a RDD is from transforming an existing RDD to create a new RDD. In other words, let's say you have the array of data that you parallelized earlier. Now you want to filter out strings that are shorter than 20 characters. A new RDD is created using the filter method.

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

Here is a quick example of how to create an RDD from an existing collection of data. In the examples throughout the course, unless otherwise indicated, we're going to be using Scala to show how Spark works. In the lab exercises, you will get to work with Python and Java as well. So the first thing is to launch the Spark shell. This command is located under the $SPARK_HOME/bin directory. In the lab environment, SPARK_HOME is the path to where Spark was installed.

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

We will cover more on the SparkContext, the sc object that is invoking the parallelized function, in our programming lesson, so for now, just know that when you initialize a shell, the SparkContext, sc, is initialized for you to use.

The parallelize method returns a pointer to the RDD. Remember, transformations operations such as parallelize, only returns a pointer to the RDD. It actually won't create that RDD until some action is invoked on it. With this new RDD, you can perform additional transformations or actions on it such as the filter transformation.

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

Here we go over some basic operations. You have seen how to load a file from an external dataset. This time, however, we are loading a file from the hdfs. Loading the file creates a RDD, which is only a pointer to the file. The dataset is not loaded into memory yet.

Nothing will happen until some action is called. The transformation basically updates the direct acyclic graph (DAG).

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

A common example is a MapReduce word count. You first split up the file by words and then map each word into a key value pair with the word as the key, and the value of 1. Then you reduce by the key, which adds up all the value of the same key, effectively, counting the number of occurrences of that key. Finally, you call the collect() function, which is an action, to have it print out all the words and its occurrences.

Again, you will get to work with this in the lab exercises.

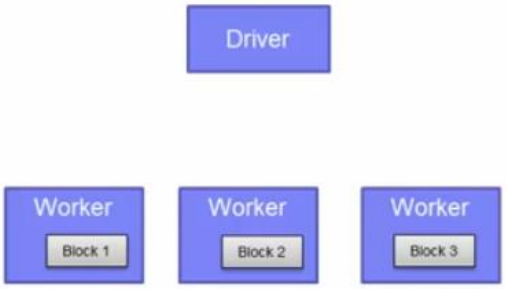
Direct Acyclic Graph (DAG)

- View the DAG
  linesLength.toDebugString
- Sample DAG

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

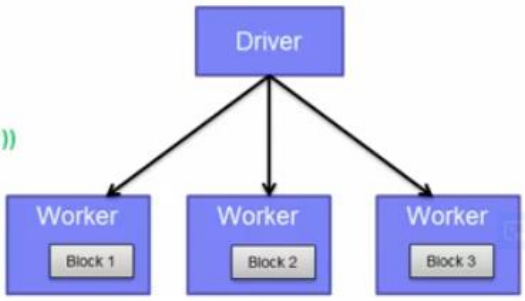
What happens when an action is executed?

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



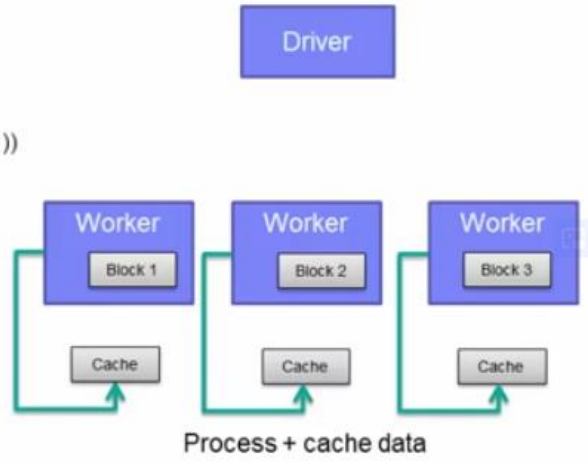
The data is partitioned into different blocks

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

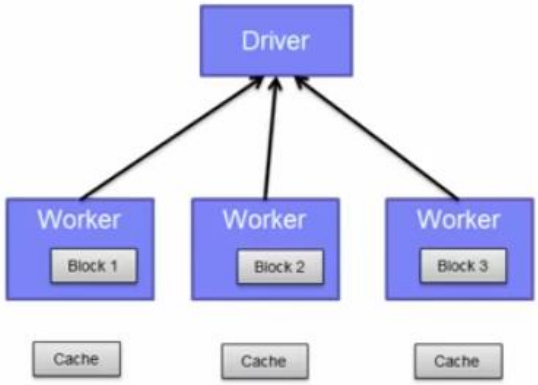


Driver sends the code to be executed on each block

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

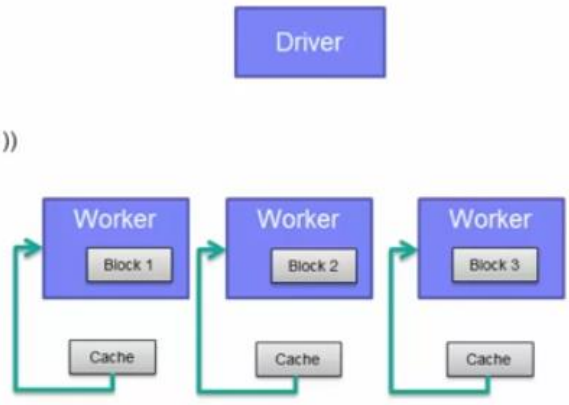



```
// Creating the RDD
val logFile = sc.textFile("hdfs://...")
// Transformations
val errors = logFile.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
//Caching
messages.cache()
// Actions
messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()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



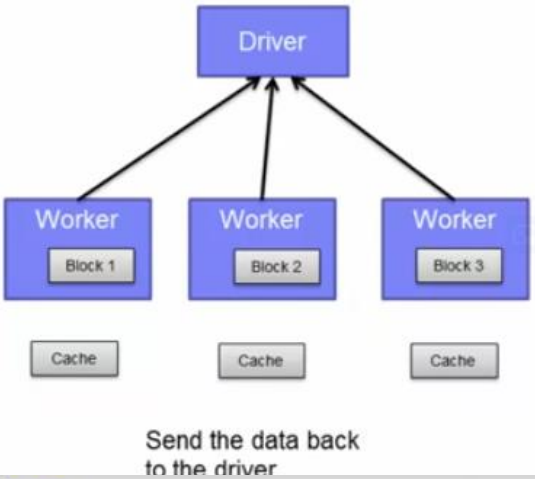
Send the data back
to the driver

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



Process from cache

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



RDD operations - Transformations

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

Transformation	Meaning
map(func)	Return a new dataset formed by passing each element of the source through a function *func*.
filter(func)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items. So func should return a Seq rather than a single item
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
reduceByKey(func)	When called on a dataset of (K, V) pairs, returns a dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K,V) pairs sorted by keys in ascending or descending order.

RDD operations - Actions

- Actions returns values

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

You heard me mention direct acyclic graph several times now. On this slide, you will see how to view the DAG of any particular RDD. A DAG is essentially a graph of the business logic and does not get executed until an action is called -- often called lazy evaluation.
To view the DAG of a RDD after a series of transformation, use the toDebugString method as you see here on the slide. It will display the series of transformation that Spark will go through once an action is called. You read it from the bottom up. In the sample DAG shown on the slide, you can see that it starts as a textFile and goes through a series of transformation such as map and filter, followed by more map operations. Remember, that it is this behavior that allows for fault tolerance. If a node goes offline and comes back on, all it has to do is just grab a copy of this from a neighboring node and rebuild the graph back to where it was before it went offline.
In the next several slides, you will see at a high level what happens when an action is executed.
Let's look at the code first. The goal here is to analyze some log files. The first line you load the log from the hadoop file system. The next two lines you filter out the messages within the log errors. Before you invoke some action on it, you tell it to cache the filtered dataset - it doesn't actually cache it yet as nothing has been done up until this point.
Then you do more filters to get specific error messages relating to mysql and php followed by the count action to find out how many errors were related to each of those filters.
Now let's walk through each of the steps. The first thing that happens when you load in the text file is the data is partitioned into different blocks across the cluster.
Then the driver sends the code to be executed on each block. In the example, it would be the various transformations and actions that will be sent out to the workers. Actually, it is the executor on each workers that is going to be performing the work on each block. You will see a bit more on executors in a later lesson.
Then the executors read the HDFS blocks to prepare the data for the operations in parallel.
After a series of transformations, you want to cache the results up until that point into memory. A cache is created.
After the first action completes, the results are sent back to the driver. In this case, we're looking for messages that relate to mysql. This is then returned back to the driver.
To process the second action, Spark will use the data on the cache -- it doesn't need to

go to the HDFS data again. It just reads it from the cache and processes the data from there.

Finally the results go back to the driver and we have completed a full cycle.

So a quick recap. This is a subset of some of the transformations available. The full list of them can be found on Spark's website.

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

I'm not going to read through this as you can do so yourself. I'll just point out some things I think are important.

The flatMap function is similar to map, but each input can be mapped to 0 or more output items. What this means is that the returned pointer of the func method, should return a sequence of objects, rather than a single item. It would mean that the flatMap would flatten a list of lists for the operations that follows. Basically this would be used for MapReduce operations where you might have a text file and each time a line is read in, you split that line up by spaces to get individual keywords. Each of those lines ultimately is flatten so that you can perform the map operation on it to map each keyword to the value of one.

The join function combines two sets of key value pairs and return a set of keys to a pair of values from the two initial set. For example, you have a K,V pair and a K,W pair. When you join them together, you will get a K, (V,W) set.

The reduceByKey function aggregates on each key by using the given reduce function. This is something you would use in a WordCount to sum up the values for each word to count its occurrences.

Action returns values. Again, you can find more information on Spark's website. This is just a subset.

The collect function returns all the elements of the dataset as an array of the driver program.

This is usually useful after a filter or another operation that returns a significantly small subset of data to make sure your filter function works correctly.

The count function returns the number of elements in a dataset and can also be used to check and test transformations.

The take(n) function returns an array with the first n elements. Note that this is currently not executed in parallel. The driver computes all the elements.

The foreach(func) function run a function func on each element of the dataset.

RDD persistence

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

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

Which storage level to choose?

- If your RDDs fit comfortably with the default storage level (MEMORY_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using MEMORY_ONLY_SER and selecting a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access.
- Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.
- Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). *All* the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.
- In environments with high amounts of memory or multiple applications, the experimental OFF_HEAP mode has several advantages:
  - It allows multiple executors to share the same pool of memory in Tachyon.
  - It significantly reduces garbage collection costs.
  - Cached data is not lost if individual executors crash.

Shared variables and key-value pairs

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

Scala: key-value pairs

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

Python: key-value pairs

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

Java: key-value pairs

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

Programming with key-value pairs

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

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

Now I want to get a bit into RDD persistence. You have seen this used already. That is the cache function. The cache function is actually the default of the persist function with the MEMORY_ONLY storage.

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

There are two methods to invoke RDD persistence. persist() and cache(). The persist() method allows you to specify a different storage level of caching. For example, you can choose to persist the data set on disk, persist it in memory but as serialized objects to save

space, etc. Again the cache() method is just the default way of using persistence by storing deserialized objects in memory.

The table here shows the storage levels and what it means. Basically, you can choose to store in memory or memory and disk. If a partition does not fit in the specified cache location,

then it will be recomputed on the fly. You can also decide to serialized the objects before storing this. This is space efficient, but will require the RDD to deserialized before it can be read, so it takes up more CPU workload. There's also the option to replicate each partition on two cluster nodes. Finally, there is an experimental storage level storing the serialized object in Tachyon. This level reduces garbage collection overhead and allows the executors to be smaller and to share a pool of memory. You can read more about this on Spark's website.

A lot of text on this page, but don't worry. It can be used as a reference when you have to decide the type of storage level.

There are tradeoffs between the different storage levels. You should analyze your current situation to decide which level works best. You can find this information here on Spark's website.

Basically if your RDD fits within the default storage level, by all means, use that.

It is the fastest option to fully take advantage of Spark's design. If not, you can serialized the RDD and use the MEMORY_ONLY_SER level. Just be sure to choose a fast serialization library to make the objects more space efficient and still reasonably fast to access.

Don't spill to disk unless the functions that compute your datasets are expensive or it requires a large amount of space.

If you want fast recovery, use the replicated storage levels. All levels are fully fault tolerant, but would still require the recomputing of the data. If you have a replicated copy, you can continue to work while Spark is reconstruction a lost partition.

Finally, use Tachyon if your environment has high amounts of memory or multiple applications.

It allows you to share the same pool of memory and significantly reduces garbage collection costs. Also, the cached data is not lost if the individual executors crash.

On these last two slides, I'll talk about Spark's shared variables and the type of operations you can do on key-value pairs.

Spark provides two limited types of shared variables for common usage patterns: broadcast variables and accumulators. Normally, when a function is passed from the driver to a worker, a separate copy of the variables are used for each worker. Broadcast variables allow each machine to work with a read-only variable cached on each machine. Spark attempts

to distribute broadcast variables using efficient algorithms. As an example, broadcast variables

can be used to give every node a copy of a large dataset efficiently.

The other shared variables are accumulators. These are used for counters in sums that works

well in parallel. These variables can only be added through an associated operation.

Only the driver can read the accumulators value, not the tasks. The tasks can only add to it. Spark supports numeric types but programmers can add support for new types. As an example,

you can use accumulator variables to implement counters or sums, as in MapReduce.

Last, but not least, key-value pairs are available in Scala, Python and Java. In Scala, you create

a key-value pair RDD by typing val pair = ('a' , 'b'). To access each element, invoke the ._ notation. This is not zero-index, so the ._1 will return the value in the first index and ._2 will return the value in the second index. Java is also very similar to Scala

where it is not zero-index. You create the Tuple2 object in Java to create a key-value pair. In Python, it is a zero-index notation, so the value of the first index resides in index 0 and the second index is 1.

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

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

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

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

I want to spend a little bit of time here to explain some of the syntax that you see on the slide. Note that in the first reduceByKey example with the a,b => a + b. This simply means that for the values of the same key, add them up together. In the example on the bottom of the slide, reduceByKey(_+_) uses the shorthand for anonymous function taking two parameters (a and b in our case) and adding them together, or multiplying, or any other operations for that matter.

Another thing I want to point is that for the goal of brevity, all the functions are concatenated on one line. When you actually code it yourself, you may want split each of the functions up. For example, do the flatMap operation and return that to a RDD. Then from that RDD, do the map operation to create another RDD. Then finally, from that last RDD, invoke the reduceByKey method. That would yield multiple lines, but you would be able to test each of the transformation to see if it worked properly.

So having completed this lesson, you should now be able to describe pretty well, RDDs.

You should also understand how to create RDDs using various methods including from existing datasets, external datasets such as a textFile or from HDFS, or even just from existing RDDs.

You saw various RDD operations and saw how to work with shared variables and key-value pairs.

Next steps. Complete lab exercise #2 Working with RDD operations. Then proceed to the next lesson in this course.

>>Lab (Scala):

Run the following lines of code to get the data

```
// download the required module to run shell commands within the notebook
import sys.process._
```

If you completed the **Getting Started** lab, then you should have the data downloaded and unzipped in the */resources/jupyterlab /labs/BD0211EN/LabData/* directory. Otherwise, please uncomment **the last two lines of code** in each of the following cells to download and unzip the data.

```
// download the data from the IBM Server
// this may take ~30 seconds depending on your internet speed

wget --quiet https://cocl.us/BD0211EN_Data" !
println("Data Downloaded!")
```

```
Name: Syntax Error.
Message:
StackTrace:
```

The data is in a folder called **LabData**. Let's list all the files in the data that we just downloaded and extracted.

```
// list the extracted files
"ls -1 /resources/jupyterlab/labs/BD0211EN/LabData" !
```

```
README.md
followers.txt
notebook.log
nyctaxi.csv
nyctaxi100.csv
nyctaxisub.csv
nycweather.csv
pom.xml
taxistreams.py
users.txt
warning: there was one feature warning; re-run with -feature for details
0
```

Now we are going to create an RDD file from the file README. This is created using the spark context ".textFile" just as in the previous lab. As we know the initial operation is a transformation, so nothing actually happens. We're just telling it that we want to create a readme RDD.

Run the code in the following cell. This was an RDD transformation, thus it returned a pointer to a RDD, which we have named as readme.

```
val readme = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/README.md")
```

Run the code in the following cell. This was an RDD transformation, thus it returned a pointer to a RDD, which we have named as readme.

```
val readme = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/README.md")
```

```
readme = /resources/jupyterlab/labs/BD0211EN/LabData/README.md MapPartitionsRDD[1] at textFile at <console>:30
/resources/jupyterlab/labs/BD0211EN/LabData/README.md MapPartitionsRDD[1] at textFile at <console>:30
```

Let's perform some RDD actions on this text file. Count the number of items in the RDD using this command:

```
readme.count()
```

```
98
```

Let's run another action. Run this command to find the first item in the RDD:

```
readme.first()
```

```
# Apache Spark
```

Now let's try a transformation. Use the filter transformation to return a new RDD with a subset of the items in the file. Type in this command:

```
val linesWithSpark = readme.filter(line => line.contains("Spark"))
linesWithSpark.count()
```

```
linesWithSpark = MapPartitionsRDD[2] at filter at <console>:30
18
```

Again, this returned a pointer to a RDD with the results of the filter transformation.

You can even chain together transformations and actions. To find out how many lines contains the word "Spark", type in:

```
readme.filter(line => line.contains("Spark")).count()
```

```
18
```

More on RDD Operations

This section builds upon the previous section. In this section, you will see that RDD can be used for more complex computations. You will find the line from that readme file with the most words in it.

```
readme.map(line => line.split(" ").size).
                   reduce((a, b) => if (a > b) a else b)
```

```
14
```

There are two parts to this. The first maps a line to an integer value, the number of words in that line. In the second part reduce is called to find the line with the most words in it. The arguments to map and reduce are Scala function literals (closures), but you can use any language feature or Scala/Java library.

In the next step, you use the Math.max() function to show that you can indeed use a Java library instead. Import in the java.lang.Math library:

```
import java.lang.Math
```

Now run with the max function:

```
readme.map(line => line.split(" ").size).
        reduce((a, b) => Math.max(a, b))
```

```
14
```

Spark has a MapReduce data flow pattern. We can use this to do a word count on the readme file.

```
val wordCounts = readme.flatMap(line => line.split(" ")).
                        map(word => (word, 1)).
                        reduceByKey((a,b) => a + b)
```

```
wordCounts = ShuffledRDD[8] at reduceByKey at <console>:33
ShuffledRDD[8] at reduceByKey at <console>:33
```

Here we combined the flatMap, map, and the reduceByKey functions to do a word count of each word in the readme file.

To collect the word counts, use the collect action.

It should be noted that the collect function brings all of the data into the driver node. For a small dataset, this isacceptable but, for a large dataset this can cause an Out Of Memory error. It is recommended to use collect() for testing only. The safer approach is to use the take() function e.g. take(n).foreach(println)

```
wordCounts.collect().foreach(println)
```

```
(package,1)
(For,2)
(Programs,1)
(processing.,1)
(Because,1)
(The,1)
(cluster.,1)
(its,1)
([run,1)
(APIs,1)
(have,1)
```

You can also do:

println(wordCounts.collect().mkString("\n"))

println(wordCounts.collect().deep)

In the cell below, determine what is the most frequent CHARACTER in the README, and how many times was it used?

```
// WRITE YOUR CODE BELOW
val wordCounts = readme.flatMap(line => line.split(" ")).
                        map(word => (word, 1)).
                        reduceByKey((a,b) => a + b).
                        reduce((a, b) => if (a._2 > b._2) a else b)

println(wordCounts)
```

```
(,67)
wordCounts = ("",67)
("",67)
```

Analysing a log file

First, let's analyze a log file in the current directory.

```
val logFile = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/notebook.log")
```

```
logFile = /resources/jupyterlab/labs/BD0211EN/LabData/notebook.log MapPartitionsRDD[13] at textFile at <console>:31
/resources/jupyterlab/labs/BD0211EN/LabData/notebook.log MapPartitionsRDD[13] at textFile at <console>:31
```

Filter out the lines that contains INFO (or ERROR, if the particular log has it)

```
val info = logFile.filter(line => line.contains("INFO"))
```

```
info = MapPartitionsRDD[14] at filter at <console>:31
MapPartitionsRDD[14] at filter at <console>:31
```

Count the lines:

```
info.count()
```

```
13438
```

Count the lines with Spark in it by combining transformation and action.

```
info.filter(line => line.contains("spark")).count()
```

```
156
```

Fetch those lines as an array of Strings

```
info.filter(line => line.contains("spark")).collect() foreach println
```

```
15/10/14 14:29:23 INFO Remoting: Remoting started; listening on addresses :[akka.tc
15/10/14 14:29:23 INFO Utils: Successfully started service 'sparkDriver' on port 53
15/10/14 14:29:23 INFO DiskBlockManager: Created local directory at /tmp/spark-fe15
-c142f2f1-ebb6-4612-945b-0a67d156230a
```

Remember that we went over the DAG. It is what provides the fault tolerance in Spark. Nodes can re-compute its state by borrowing the DAG from a neighboring node. You can view the graph of an RDD using the toDebugString command.

```
println(info.toDebugString)
```

```
(1) MapPartitionsRDD[14] at filter at <console>:31 []
 |  /resources/jupyterlab/labs/BD0211EN/LabData/notebook.log MapPartitionsRDD[13] at textFile at <console>:31 []
 |  /resources/jupyterlab/labs/BD0211EN/LabData/notebook.log HadoopRDD[12] at textFile at <console>:31 []
```

Joining RDDs

Next, you are going to create RDDs for the README and the POM file in the current directory.

```
val readmeFile = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/README.md")
val pom = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/pom.xml")
```

```
readmeFile = /resources/jupyterlab/labs/BD0211EN/LabData/README.md MapPartitionsRDD[18] at textFile at <console>:31
pom = /resources/jupyterlab/labs/BD0211EN/LabData/pom.xml MapPartitionsRDD[20] at textFile at <console>:32
/resources/jupyterlab/labs/BD0211EN/LabData/pom.xml MapPartitionsRDD[20] at textFile at <console>:32
```

How many Spark keywords are in each file?

```
println(readmeFile.filter(line => line.contains("Spark")).count())
println(pom.filter(line => line.contains("Spark")).count())
```

```
18
2
```

Now do a WordCount on each RDD so that the results are (K,V) pairs of (word,count)

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

val pomCount = pom.
                flatMap(line => line.split(" ")).
                map(word => (word, 1)).
                reduceByKey(_ + _)
```

```
readmeCount = ShuffledRDD[25] at reduceByKey at <console>:35
pomCount = ShuffledRDD[28] at reduceByKey at <console>:40
ShuffledRDD[28] at reduceByKey at <console>:40
```

To see the array for either of them, just call the collect function on it.

```
println("Readme Count\n")
readmeCount.collect() foreach println
```

```
Readme Count

(package,1)
(For,2)
(Programs,1)
(processing.,1)
(Because,1)
(The,1)
(cluster.,1)
(its,1)
([run,1)
```

Now let's join these two RDDs together to get a collective set. The join function combines the two datasets (K,V) and (K,W) together and get (K, (V,W)). Let's join these two counts together and then cache it.

```
val joined = readmeCount.join(pomCount)
joined.cache()
```

```
joined = MapPartitionsRDD[31] at join at <console>:32
MapPartitionsRDD[31] at join at <console>:32
```

Let's see what's in the joined RDD.

```
joined.collect.foreach(println)
```

```
(file,(1,3))
(are,(1,1))
```

Let's combine the values together to get the total count. The operations in this command tells Spark to combine the values from (K,V) and (K,W) to give us(K, V+W). The ._ notation is a way to access the value on that particular index of the key value pair.

```
val joinedSum = joined.map(k => (k._1, (k._2)._1 + (k._2)._2))
joinedSum.collect() foreach println
```

```
(file,4)
(are,2)
(be,3)
(at,3)
(or,6)
(of,7)
(this,4)
(following,3)
```

Shared variables

Broadcast variables allow the programmer to keep a read-only variable cached on each worker node rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. After the broadcast variable is created, it should be used instead of the value v in any functions run on the cluster so that v is not shipped to the nodes more than once. In addition, the object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

Let's create a broadcast variable:

```
val broadcastVar = sc.broadcast(Array(1,2,3))
```

```
broadcastVar = Broadcast(27)
Broadcast(27)
```

To get the value, type in:

```
broadcastVar.value
```

```
Array(1, 2, 3)
```

Accumulators are variables that can only be added throu
parallel. Spark natively supports numeric type accumulat
Only the driver can read the values of the accumulators.

Create the accumulator variable. Type in:

```
val accum = sc.accumulator(0)
```

```
accum = 0
warning: there were two deprecation warnings; re-run with -deprecation for details
0
```

Next parallelize an array of four integers and run it through a loop to add each integer value to the accumulator variable. Type in:

```
sc.parallelize(Array(1,2,3,4)).foreach(x => accum += x)
```

To get the current value of the accumulator variable, type in:

```
accum.value
```

```
10
```

Key-value pairs

You have already seen a bit about key-value pairs in the Joining RDD section. Here is a brief example of how to create a key-value pair and access its values. Remember that certain operations such as map and reduce only works on key-value pairs.

Create a key-value pair of two characters. Type in:

```
val pair = ('a', 'b')
```

```
pair = (a,b)
(a,b)
```

Sample Application

In this section, you will be using a subset of a data for taxi trips that will determine the top 10 medallion numbers based on the number of trips.

```
val taxi = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/nyctaxi.csv")
```

```
taxi = /resources/jupyterlab/labs/BD0211EN/LabData/nyctaxi.csv MapPartitionsRDD[35] at textFile at <console>:31
/resources/jupyterlab/labs/BD0211EN/LabData/nyctaxi.csv MapPartitionsRDD[35] at textFile at <console>:31
```

To view the five rows of content, invoke the take function. Type in:

```
taxi.take(5).foreach(println)
```

```
"_id","_rev","dropoff_datetime","dropoff_latitude","dropoff_longitude","hack_license","medallion","passenger_count","pickup
datetime","pickup_latitude","pickup_longitude","rate_code","store_and_fwd_flag","trip_distance","trip_time_in_secs","vendor
id"
"29b3f4a30dea6688d4c289c9672cb996","1-ddfdec8050c7ef4dc694eeeda6c4625e","2013-01-11 22:03:00",+4.07033460000000E+001,-7.401
4200000000E+001,"A93D1F7F8998FFB75EEF477EB6077516","68BC16A99E915E44ADA7E639B4DD5F59",2,"2013-01-11 21:48:00",+4.0676067000
```

To parse out the values, including the medallion numbers, you need to first create a new RDD by splitting the lines of the RDD using the comma as the delimiter. Type in:

```
val taxiParse = taxi.map(line=>line.split(","))
```

```
taxiParse = MapPartitionsRDD[36] at map at <console>:31
MapPartitionsRDD[36] at map at <console>:31
```

Now create the key-value pairs where the key is the medallion number and the value is 1. We use this model to later sum up all the keys to find out the number of trips a particular taxi took and in particular, will be able to see which taxi took the most trips. Map each of the medallions to the value of one. Type in:

```
val taxiMedKey = taxiParse.map(vals=>(vals(6), 1))
```

```
taxiMedKey = MapPartitionsRDD[37] at map at <console>:31
MapPartitionsRDD[37] at map at <console>:31
```

vals(6) corresponds to the column where the medallion key is located

Finally, the values are swapped so they can be ordered in descending order and the results are presented correctly.

```
for (pair <-taxiMedCounts.map(_.swap).top(10)) println("Taxi Medallion %s had %s Trips".format(pair._2, pair._1))
```

```
Taxi Medallion "FE4C521F3C1AC6F2598DEF00DDD43029" had 415 Trips
Taxi Medallion "F5BB809E7858A669C9A1E8A12A3CCF81" had 411 Trips
Taxi Medallion "8CE240F0796D072D5DCFE06A364FB5A0" had 406 Trips
Taxi Medallion "0310297769C8B049C0EA8E87C697F755" had 402 Trips
Taxi Medallion "B6585890F68EE02702F32DECDEABC2A8" had 399 Trips
Taxi Medallion "33955A2FCAF62C6E91A11AE97D96C99A" had 395 Trips
Taxi Medallion "4F7C132D3130970CFA892CC858F5ECB5" had 391 Trips
Taxi Medallion "78833E177D45E4BC520222FFBBAC5B77" had 383 Trips
Taxi Medallion "E097412FE23295A691BEEE56F28FB9E2" had 380 Trips
Taxi Medallion "C14289566BAAD9AEDD0751E5E9C73FBD" had 377 Trips
```

While each step above was processed one line at a time, you can just as well process everything on one line:

```
val taxiMedCountsOneLine = taxi.map(line=>line.split(',')).map(vals=>(vals(6),1)).reduceByKey(_ + _)
```

Run the same line as above to print the taxiMedCountsOneLine RDD.

Run the same line as above to print the taxiMedCountsOneLine RDD.

```
for (pair <-taxiMedCountsOneLine.map(_.swap).top(10)) println("Taxi Medallion %s had %s Trips".format(pair._2, pair._1))
```

Let's cache the taxiMedCountsOneLine to see the difference caching makes. Run it with the logs set to INFO and you can see the output of the time it takes to execute each line. First, let's cache the RDD

```
taxiMedCountsOneLine.cache()
```

Next, you have to invoke an action for it to actually cache the RDD. Note the time it takes here (either empirically using the INFO log or just notice the time it takes)

```
taxiMedCountsOneLine.count()
```

Run it again to see the difference.

```
taxiMedCountsOneLine.count()
```

The bigger the dataset, the more noticeable the difference will be. In a sample file such as ours, the difference may be negligible.

>>Lab (Python):

Analyzing a log file

First let's download the tools that we need to use Spark in SN Labs.

```
!pip install findspark
!pip install pyspark
import findspark
import pyspark
findspark.init()
sc = pyspark.SparkContext.getOrCreate()
```

```
Collecting findspark
  Downloading https://files.pythonhosted.org/packages/b1/c8/e6e1f6a3
3-none-any.whl
```

[5]:
```
## download the data from the IBM server
## this may take ~30 seconds depending on your interent speed

!wget --quiet https://cocl.us/BD0211EN_Data
print("Data Downloaded!")
```

```
Data Downloaded!
```

[6]:
```
## unzip the folder's content into "resources" directory
## this may take ~30 seconds depending on your internet speed

!unzip -q -o -d /resources/jupyterlab/labs/BD0211EN/ BD0211EN_Data
print("Data Extracted!")
```

```
Data Extracted!
```

[4]:
```
# list the extracted files
!ls -1 /resources/jupyterlab/labs/BD0211EN/LabData/
```

```
README.md
```

Now, let's create an RDD by loading the log file that we analyze in the Scala version of this lab.

```
logFile = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/notebook.log")
```

```
# WRITE YOUR CODE BELOW

info=logFile.filter(lambda line:"INFO" in line)
print(info)
```

```
PythonRDD[4] at RDD at PythonRDD.scala:53
```

Count the lines:

```
# WRITE YOUR CODE BELOW
count=info.count()
print(count)
```

```
13438
```

Count the lines with "spark" in it by combining transformation and action.

```
# WRITE YOUR CODE BELOW
info.filter(lambda line:"spark" in line).count()
```

```
156
```

Fetch those lines as an array of Strings

```
# WRITE YOUR CODE BELOW
info.filter(lambda line:"spark" in line).collect()
```

```
['15/10/14 14:29:23 INFO Remoting: Remoting started; li
 "15/10/14 14:29:23 INFO Utils: Successfully started se
 '15/10/14 14:29:23 INFO DiskBlockManager: Created loca
```

View the graph of an RDD using this command:

```
print(info.toDebugString())
```

```
b'(2) PythonRDD[4] at RDD at PythonRDD.scala:53 [
e at NativeMethodAccessorImpl.java:0 []\n |  /res
ccessorImpl.java:0 []'
```

Joining RDDs

Next, you are going to create RDDs for the same README and the POM files that we used in the Scala version.

```
readmeFile = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/README.md")
pomFile = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/pom.xml")
```

How many Spark keywords are in each file?

```
print(readmeFile.filter(lambda line: "Spark" in line).count())
print(pomFile.filter(lambda line: "Spark" in line).count())
```

```
18
2
```

Now do a WordCount on each RDD so that the results are (K,V) pairs of (word,count)

```
readmeCount = readmeFile.                    \
    flatMap(lambda line: line.split("   ")).   \
    map(lambda word: (word, 1)).             \
    reduceByKey(lambda a, b: a + b)

pomCount = pomFile.                          \
    flatMap(lambda line: line.split("   ")).   \
    map(lambda word: (word, 1)).               \
    reduceByKey(lambda a, b: a + b)
```

To see the array for either of them, just call the collect function on it.

```
print("Readme Count\n")
print(readmeCount.collect())
```

To see the array for either of them, just call the collect function on it.

```
print("Readme Count\n")
print(readmeCount.collect())
```

```
Readme Count

[('', 43), ('Spark is a fast and general cluster computing system for
g.', 1), ('guide, on the [project web page](http://spark.apache.org/d
[Apache Maven](http://maven.apache.org/).', 1), (' build/mvn -DskipTe
```

```
print("Pom Count\n")
print(pomCount.collect())
```

```
Pom Count

[('<?xml version="1.0" encoding="UTF-8"?>', 1), ('  ~ Licens
r license agreements.  See the NOTICE file distributed with'
', 1), (' http://www.apache.org/licenses/LICENSE-2.0', 1), (
limitations under the License.', 1), ('  -->', 1), ('', 841)
/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.a
ersion>4.0.0</modelVersion>', 1), ('  <parent>', 1), (' <gro
', 1), (' <version>1.6.0-SNAPSHOT</version>', 1), ('  <prope
s>', 1), ('  <packaging>jar</packaging>', 1), ('  <dependenc
</dependency>', 24), ('<artifactId>spark-streaming_${scala.b
```

The join function combines the two datasets (K,V) and (K,W) together and get (K, (V,W)). Let's join these two counts together.

```
joined = readmeCount.join(pomCount)
```

Print the value to the console

```
joined.collect()
```

```
[('', (43, 841))]
```

Let's combine the values together to get the total count

```
joinedSum = joined.map(lambda k: (k[0], (k[1][0]+k[1][1])))
```

To check if it is correct, print the first five elements from the joined and the joinedSum RDD

```
print("Joined Individial\n")
print(joined.take(5))

print("\n\nJoined Sum\n")
print(joinedSum.take(5))
```

```
Joined Individial
```

Shared variables

Normally, when a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all the variables used in the function. These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program. Supporting general, read-write shared variables across tasks would be inefficient. However, Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators.

Broadcast variables

Broadcast variables are useful for when you have a large dataset that you want to use across all the worker nodes. A read-only variable is cached on each machine rather than shipping a copy of it with tasks. Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark

```
broadcastVar = sc.broadcast([1,2,3])
```

To get the value, type in:

```
broadcastVar.value
```

```
[1, 2, 3]
```

Accumulators

Accumulators are variables that can only be added through an associative operation. It is used to implement counters and sum efficiently in parallel. Spark natively supports numeric type accumulators and standard mutable collections. Programmers can extend these for new types. Only the driver can read the values of the accumulators. The workers can only invoke it to increment the value.

Create the accumulator variable. Type in:

```
accum = sc.accumulator(0)
```

Next parallelize an array of four integers and run it through a loop to add each integer value to the accumulator variable. Type in:

```
rdd = sc.parallelize([1,2,3,4])
def f(x):
    global accum
    accum += x
```

Next, iterate through each element of the rdd and apply the function f on it:

```
rdd.foreach(f)
```

To get the current value of the accumulator variable, type in:

```
accum.value
```

```
10
```

You should get a value of 10.

This command can only be invoked on the driver side. The worker nodes can only increment the accumulator.

Key-value pairs

You have already seen a bit about key-value pairs in the Joining RDD section.

Create a key-value pair of two characters. Type in:

```
pair = ('a', 'b')
```

To access the value of the first index use [0] and [1] method for the 2nd.

```
print(pair[0])
print(pair[1])
```

```
a
b
```

>>Lab (Dadtaframes):

A DataFrame is two-dimensional. Columns can be of different data types. DataFrames accept many data inputs including series and other DataFrames. You can pass indexes (row labels) and columns (column labels). Indexes can be numbers, dates, or strings/tuples.

Pandas is a library used for data manipulation and analysis. Pandas offers data structures and operations for creating and manipulating Data Series and DataFrame objects. Data can be imported from various data sources, e.g., Numpy arrays, Python dictionaries and CSV files. Pandas allows you to manipulate, organize and display the data.

In this short notebook, we will load and explore the mtcars dataset. Specifically, this tutorial covers:

1. Loading data in memory
2. Creating SQLContext
3. Creating Spark DataFrame
4. Group data by columns
5. Operating on columns
6. Running SQL Queries from a Spark DataFrame

[1]:
```
import pandas as pd
mtcars = pd.read_csv('https://cocl.us/BD0211EN_mtcars')
```

[2]:
```
mtcars.head()
```

[2]:

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Initialize SQLContext

To work with dataframes we need an SQLContext which is created using `SQLContext(sc)`. SQLContext uses SparkContext which has been already created in Data Scientist Workbench, named `sc`.

But first, let's import the tools that we need to use Spark in SN Labs.

```
!pip install findspark
!pip install pyspark
import findspark
import pyspark
findspark.init()
sc = pyspark.SparkContext.getOrCreate()
```

```
Requirement already satisfied: findspark in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (1.3.0)
Requirement already satisfied: pyspark in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (2.4.5)
Requirement already satisfied: py4j==0.10.7 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from pyspark) (0.10.7)
```

```
sqlContext = SQLContext(sc)
```

Creating Spark DataFrames

With SQLContext and a loaded local DataFrame, we create a Spark DataFrame:

```
sdf = sqlContext.createDataFrame(mtcars)
sdf.printSchema()
```

Displays the content of the DataFrame

```
sdf.show(5)
```

Selecting columns

```
sdf.select('mpg').show(5)
```

Filtering Data

Filter the DataFrame to only retain rows with `mpg` less than 18

Operating on Columns

SparkR also provides a number of functions that can be directly applied to columns for data processing and aggregation. The example below shows the use of basic arithmetic functions to convert lb to metric ton.

```
sdf.withColumn('wtTon', sdf['wt'] * 0.45).show(6)
```

```
sdf.show(6)
```

Grouping, Aggregation

Spark DataFrames support a number of commonly used functions to aggregate data after grouping. For example we can compute the average weight of cars by their cylinders as shown below:

```
sdf.groupby(['cyl'])\
.agg({"wt": "AVG"})\
.show(5)
```

```
# We can also sort the output from the aggregation to get the most common cars
car_counts = sdf.groupby(['cyl'])\
.agg({"wt": "count"})\
.sort("count(wt)", ascending=False)\
.show(5)
```

Running SQL Queries from Spark DataFrames

A Spark DataFrame can also be registered as a temporary table in Spark SQL and registering a DataFrame as a table allows you to run SQL queries over its data. The `sql` function enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
# Register this DataFrame as a table.
sdf.registerTempTable("cars")

# SQL statements can be run by using the sql method
highgearcars = sqlContext.sql("SELECT gear FROM cars WHERE cyl >= 4 AND cyl <= 9")
highgearcars.show(6)
```

MODULE 3 – SPARK APPLICATION PROGRAMMING

Linking with Spark - Scala

- Spark applications requires certain dependencies.
- Must have a compatible Scala version to write applications.
  - e.g Spark 1.1.1 uses Scala 2.10.
- To write a Spark application, you need to add a Maven dependency on Spark.
  - Spark is available through Maven Central at:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.1.1
```

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

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

Linking with Spark - Python

- Spark 1.1.1 works with Python 2.6 or higher (but not Python 3)
- Uses the standard CPython interpreter, so C libraries like NumPy can be used.
- To run Spark applications in Python, use the *bin/spark-submit* script located in the Spark directory.
  - Load Spark's Java/Scala libraries
  - Allow you to submit applications to a cluster
- If you wish to access HDFS, you need to use a build of PySpark linking to your version of HDFS.
- Import some Spark classes

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

Linking with Spark - Java

- Spark 1.1.1 works with Java 6 and higher.
  - Java 8 supports lambda expressions
- Add a dependency on Spark
  - Available through Maven Central at:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.1.1
```

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

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

- Import some Spark classes

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

Initializing Spark - Scala

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

Initializing Spark - Python

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

Initializing Spark - Java

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

Hi. Welcome to the Spark Fundamentals course. This lesson will cover Spark application programming.

After completing this lesson, you should be able to understand the purpose and usage of the SparkContext.This lesson will show you how to can get started by programming your own Spark application. First you will see how to link to Spark. Next you will see how to run some Spark examples. You should also be able to understand how to pass functions to Spark and be able to create and run a Spark standalone application. Finally, you should be able to submit applications to the Spark cluster.

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

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

To link with Spark using Scala, you must have a compatible version of Scala with the Spark you choose to use. For example, Spark 1.1.1 uses Scala 2.10, so make sure that you have Scala 2.10 if you wish to write applications for Spark 1.1.1.

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

In the lab environment, this is already set up for you. The information on this page shows how you would set up for your own Spark cluster.

Spark 1.1.1 works with Python 2.6 or higher, but not Python 3. It uses the standard CPython interpreter, so C libraries like NumPy can be used.

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

Spark 1.1.1 works with Java 6 and higher. If you are using Java 8, Spark supports lambda expressions for concisely writing functions. Otherwise, you can use the org.apache.spark.api.java.function package with older Java versions.

As with Scala, you need to a dependency on Spark, which is available through Maven Central.

If you wish to access an HDFS cluster, you must add the dependency there as well. Last, but not least, you need to import some Spark classes.

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

For example, val conf = new SparkConf().setAppName(appName).setMaster(master).

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

16.

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

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

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

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

Passing functions to Spark

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

```
(x: Int) => x + 1
```

  - Static methods in a global singleton object

```
object MyFunctions {
     def func1 (s: String): String = {…}
}
myRdd.map(MyFunctions.func1)
```

  - Passing by reference
    - To avoid sending the entire object, consider copying the function to a local variable.
    - Example:

```
val field = "Hello"
```

    - Avoid:

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

    - Consider:

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

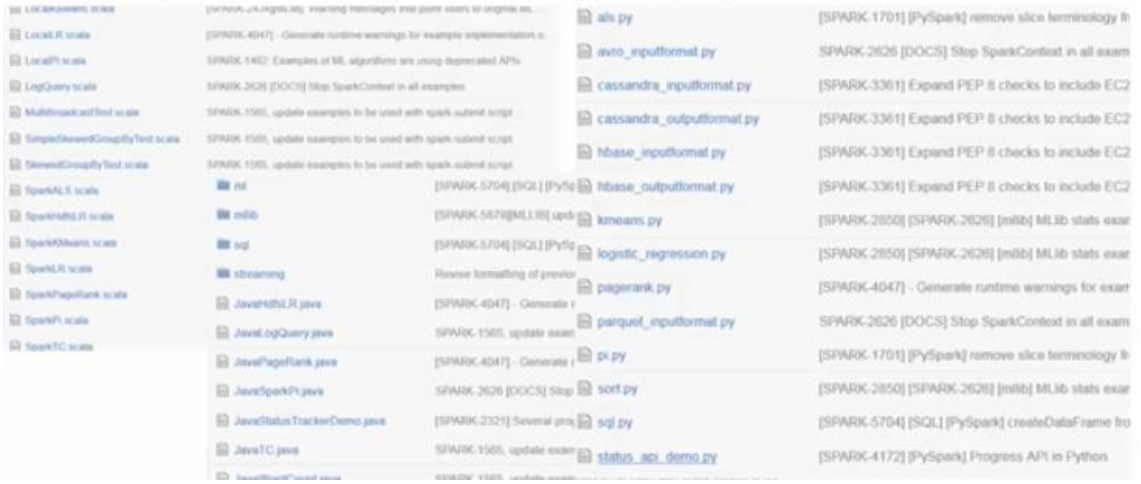
Programming the business logic

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

```
package org.apache.spark.examples

import org.apache.spark._


object HdfsTest {

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

Running Spark examples

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



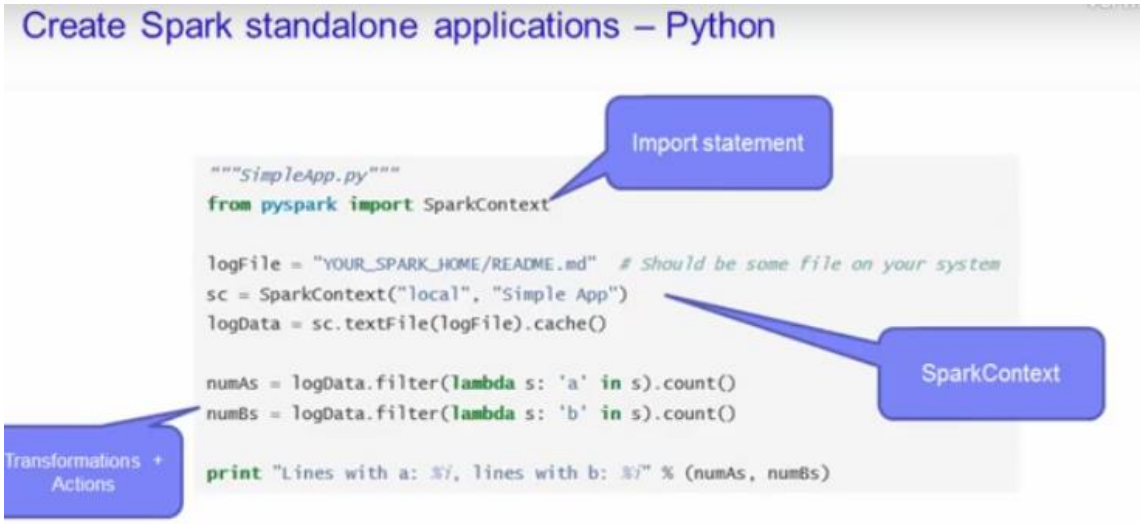
Passing functions to Spark. I wanted to touch a little bit on this before we move further. This is important to understand as you begin to think about the business logic of your application.

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

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

The first method to do this is using an anonymous function syntax. You saw briefly what an anonymous
function is in the first lesson. This is useful for short pieces of code. For example, here we define the anonymous function that takes in a particular parameter x of type Int and add one to it. Essentially, anonymous functions are useful for one-time use of the function. In other words, you don't need to explicitly define the function to use it. You define it as you go. Again, the left side of the => are the parameters or the arguments. The right side of the => is the body of the function.
Another method to pass functions around Spark is to use static methods in a global singleton object. This means that you can create a global object, in the example, it is the object MyFunctions.
Inside that object, you basically define the function func1. When the driver requires that function, it only needs to send out the object -- the worker will be able to access it. In this case, when the driver sends out the instructions to the worker, it just has to send out the singleton object.
It is possible to pass reference to a method in a class instance, as opposed to a singleton object. This would require sending the object that contains the class along with the method. To avoid this consider copying it to a local variable within the function instead of accessing it externally.
Example, say you have a field with the string Hello. You want to avoid calling that directly inside a function as shown on the slide as x => field + x.
Instead, assign it to a local variable so that only the reference is passed along and not the entire object shown val field_ = this.field.
For an example such as this, it may seem trivial, but imagine if the field object is not a simple text Hello, but is something much larger, say a large log file. In that case, passing by reference will have greater value by saving a lot of storage by not having to pass the entire file.
Back to our regularly scheduled program. At this point, you should know how to link dependencies with Spark and also know how to initialize the SparkContext. I also touched a little bit on passing functions with Spark to give you a better view of how you can program your business logic. This course will not focus too much on how to program business logics, but there are examples available for you to see how it is done. The purpose is to show you how you can create an application using a simple, but effective example which demonstrates Spark's capabilities.
Once you have the beginning of your application ready by creating the SparkContext object, you can start to program in the business logic using Spark's API available in Scala, Java, or Python. You create the RDD from an external dataset or from an existing RDD. You use transformations
and actions to compute the business logic. You can take advantage of RDD persistence, broadcast variables and/or accumulators to improve the performance of your jobs.
Here's a sample Scala application. You have your import statement. After the beginning of the object, you see that the SparkConf is created with the application name. Then a SparkContext is created. The several lines of code after is creating the RDD from a text file and then performing the Hdfs test on it to see how long the iteration through the file takes. Finally, at the end, you stop the SparkContext by calling the stop() function.
Again, just a simple example to show how you would create a Spark application. You will get to practice this in the lab exercise.
I mentioned that there are examples available which shows the various usage of Spark. Depending
on your programming language preference, there are examples in all three languages that work
with Spark. You can view the source code of the examples on the Spark website or within

the Spark distribution itself. I provided some screenshots here to show you some of the examples available.
On the slide, I also listed the step to run these examples. To run Scala or Java examples, you would execute the run-example script under the Spark's home/bin directory. So for example,
to run the SparkPi application, execute run-example SparkPi, where SparkPi would be the name of
the application. Substitute that with a different application name to run that other application.
To run the sample Python applications, use the spark-submit command and provide the path to the application.



Create Spark standalone applications – Java

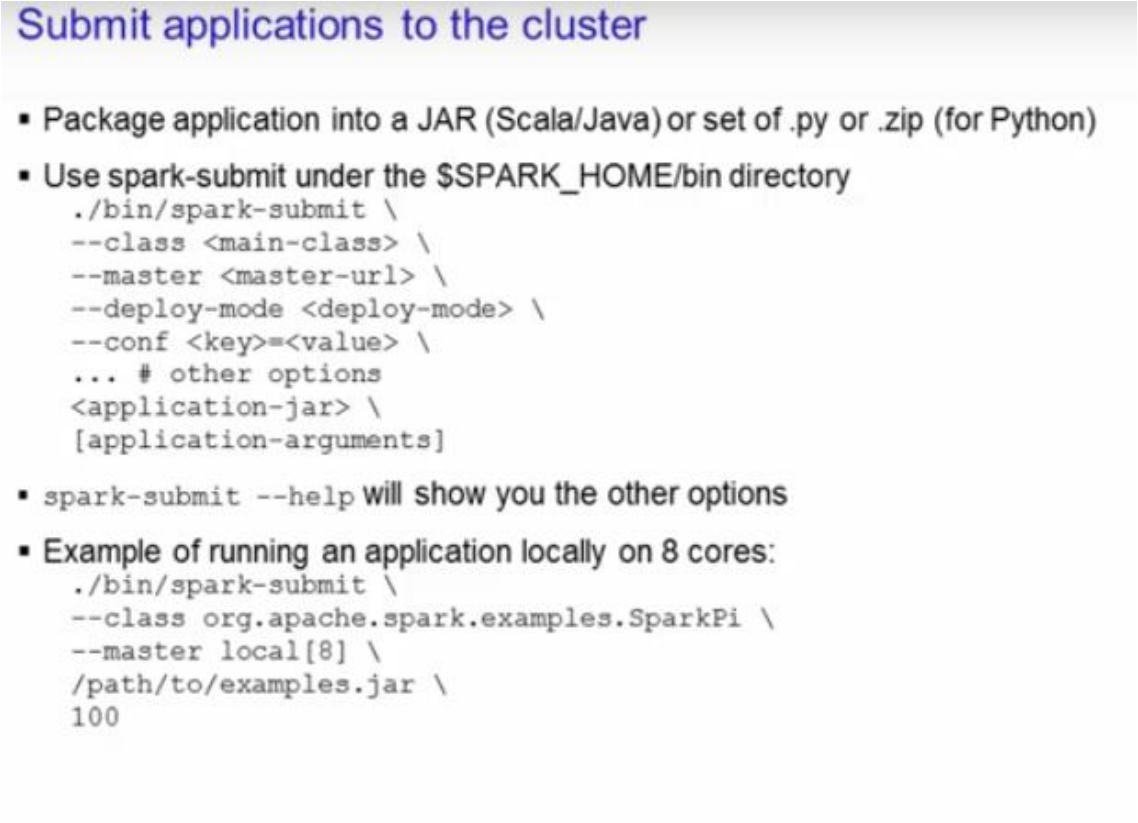

```java
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system
    SparkConf conf = new SparkConf().setAppName("Simple Application");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("a"); }
    }).count();

    long numBs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("b"); }
    }).count();

    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
  }
}
```

Run standalone applications

- Define the dependencies – can use any system builds (Ant, sbt, Maven)
- Example:
  - Scala → simple.sbt
  - Java → pom.xml
  - Python → --py-files argument (not needed for SimpleApp.py)
- Create the typical directory structure with the files

Scala using SBT :	Java using Maven:
./simple.sbt	./pom.xml
./src	./src
./src/main	./src/main
./src/main/scala	./src/main/java
./src/main/scala/SimpleApp.scala	./src/main/java/SimpleApp.java

- Create a JAR package containing the application's code.
  - Scala: sbt
  - Java: mvn
  - Python: submit-spark
- Use spark-submit to run the program



In the next three slides, you will see another example of creating a Spark application. First you will see how to do this in Scala. The next following sets of slides will show Python and Java.

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

if you wish to code this application.

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

In Python, this application does the exact same thing, that is, count the number of lines with 'a' in it, and the number of lines with 'b' in it. You use a SparkContext object to create the RDDs and cache it. Then you run the transformations and actions, follow by a print to the console. Nothing entirely new here, just a difference in syntax.

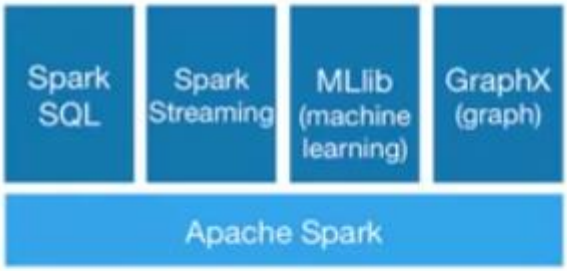
Similar to Scala and Python, in Java you need to get a JavaSparkContext. RDDs are represented by JavaRDD. Then you run the transformations and actions on them. The lambda expressions of Java 8 allows you to concisely write functions. Otherwise, you can use the classes in the org.apache.spark.api.java.function package for older versions of java. The business logic is the same as the previous two examples to count the number of a's and b's from the Readme file. Just a matter of difference in the syntax and library names.

Up until this point, you should know how to create a Spark application using any of the supported programming languages. Now you get to see how to run the application. You will need to first define the dependencies. Then you have to package the application together using system build tools such as Ant, sbt, or Maven. The examples here show how you would

do it using the various tools. You can use any tool for any of the programming languages.
For Scala, the example is shown using sbt, so you would have a simple.sbt file. In Java, the example shows using Maven so you would have the pom.xml file. In Python, if you need to have dependencies that requires third party libraries, then you can use the --py-files argument to handle that.
Again, shown here are examples of what a typical directory structure would look like for the tool that you choose.
Finally, once you have the JAR packaged created, run the spark-submit to execute the application.
In the lab exercise, you will get to practice this.
In short, you package up your application into a JAR for Scala or Java or a set of .py or .zip files for Python.
To submit your application to the Spark cluster, you use spark-submit command, which is located
under the $SPARK_HOME/bin directory.
The options shown on the slide are the commonly used options. To see other options, just invoke
spark-submit with the help argument.
Let's briefly go over what each of these options mean.
The class option is the main entry point to your class. If it is under a package name, you must provide the fully qualified name. The master URL is where your cluster is located. Remember that it is recommended approach to provide the master URL here, instead of hardcoding
it in your application code.
The deploy-mode is whether you want to deploy your driver on the worker nodes (cluster) or locally as an external client (client). The default deploy-mode is client.
The conf option is any configuration property you wish to set in key=value format.
The application jar is the file that you packaged up using one of the build tools.
Finally, if the application has any arguments, you would supply it after the jar file.
Here's an actual example of running a Spark application locally on 8 cores. The class is the org.apache.spark.examples.SparkPi. local[8] is saying to run it locally on 8 cores. The examples.jar is located on the given path with the argument 100 to be passed into the SparkPi application.
Having completed this lesson, you should now know how to create a standalone Spark application
and run it by submitting it to the Spark Cluster. You saw briefly on the different methods on how to pass functions in Spark. All three programming languages were shown in this lesson.
Next steps. Copmlete lab exercise #3, Creating a Spark application and then proceed to the next lesson in this course.

Spark libraries

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



spark.apache.org

Spark SQL

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

Spark SQL – Getting started

- SQLContext
  - Created from a SparkContext

  Scala:

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

  Java:

```
JavaSparkContext sc = ...; // An existing JavaSparkContext.
JavaSQLContext sqlContext = new
org.apache.spark.sql.api.java.JavaSQLContext(sc);
```

  Python:

```
from pyspark.sql import SQLContext sqlContext = SQLContext(sc)
```

- Import a library to convert an RDD to a SchemaRDD
  - Scala only: `import sqlContext.createSchemaRDD`
- SchemaRDD data sources:
  - Inferring the schema using reflection
  - Programmatic interface

Spark SQL – Inferring the schema using reflection

- The *case class* in Scala defines the schema of the table.

```
case class Person(name: String, age: Int)
```

- The arguments of the case class becomes the names of the columns.
- Create the RDD of the *Person* object

```
val people =
sc.textFile("examples/src/main/resources/people.txt")
.map(_.split(","))
.map(p => Person(p(0), p(1).trim.toInt))
```

- Register the RDD as a table

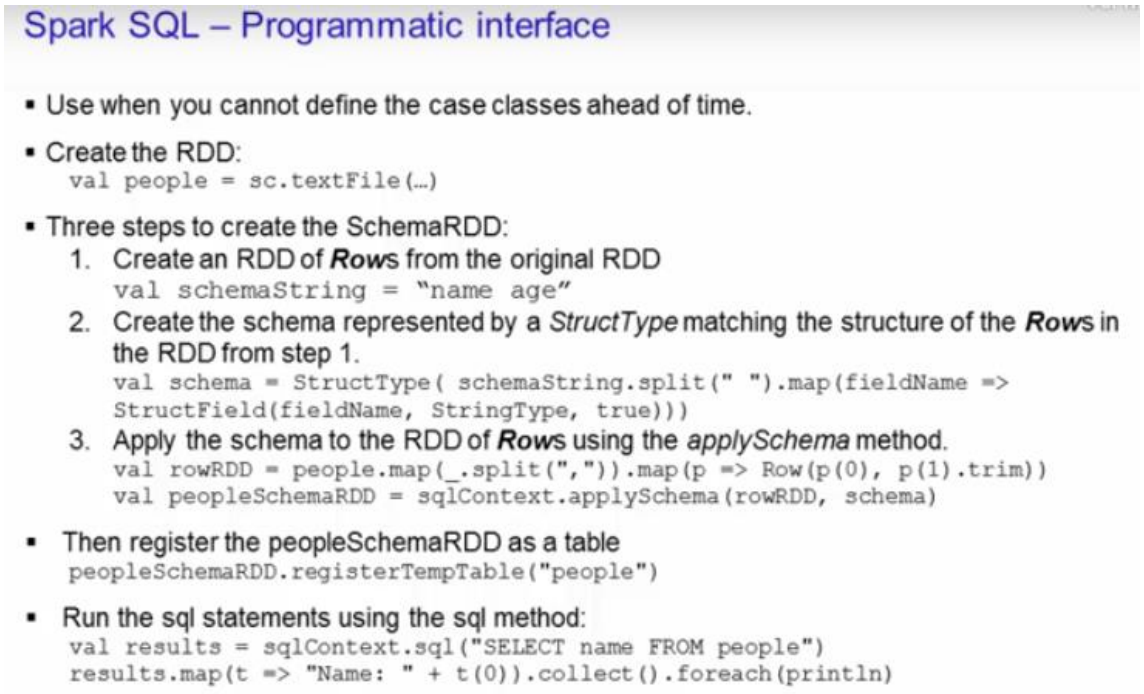
```
people.registerTempTable("people")
```

- Run SQL statements using the *sql* method provided by the SQLContext

```
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND
age <= 19")
```

- The results of the queries are SchemaRDD. Normal RDD operations also work on them

```
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```



Hi. Welcome to the Spark Fundamentals course. This lesson will cover the Introduction to the Spark libraries.
After completing this lesson, you should be able to understand and use the various Spark libraries including SparkSQL, Spark Streaming, MLlib and GraphX
Spark comes with libraries that you can utilize for specific use cases. These libraries are an extension of the Spark Core API. Any improvements made to the core will automatically take effect
with these libraries. One of the big benefits of Spark is that there is little overhead to use these libraries with Spark as they are tightly integrated. The rest of this lesson will cover on a high level, each of these libraries and their capabilities. The main focus will be on Scala with specific callouts to Java or Python if there are major differences.
The four libraries are Spark SQL, Spark Streaming, MLlib, and GraphX.
Spark SQL allows you to write relational queries that are expressed in either SQL, HiveQL, or Scala to be executed using Spark. Spark SQL has a new RDD called the SchemaRDD. The SchemaRDD consists of rows objects and a schema that describes the type of data in each column
in the row. You can think of this as a table in a traditional relational database.
You create a SchemaRDD from existing RDDs, a Parquet file, a JSON dataset, or using HiveQL to query against the data stored in Hive. You can write Spark SQL application using Scala, Java or Python.
The SQLContext is created from the SparkContext. You can see here that in Scala, the sqlContext
is created from the SparkContext. In Java, you create the JavaSQLContext from the JavaSparkContext.
In Python, you also do the same.
There is a new RDD, called the SchemaRDD that you use with Spark SQL. In Scala only, you have to import a library to convert an existing RDD to a SchemaRDD. For the others, you do not need to import a library to work with the Schema RDD.
So how do these SchemaRDDs get created? There are two ways you can do this.
The first method uses reflection to infer the schema of the RDD. This leads to a more concise code and works well when you already know the schema while writing your Spark application.

The second method uses a programmatic interface to construct a schema and then apply that to an existing RDD. This method gives you more control when you don't know the schema of the RDD until runtime. The next two slides will cover these two methods in more detail.
The first of the two methods used to determine the schema of the RDD is to use reflection. In this scenario, use the case class in Scala to define the schema of the table. The arguments of the case class are read using reflection and becomes the names of the columns.
Let's go over the code shown on the slide. First thing is to create the RDD of the person object.
You load the text file in using the textFile method. Then you invoke the map transformation to split the elements on a comma to get the individual columns of name and age. The final transformation creates the Person object based on the elements.
Next you register the people RDD that you just created by loading in the text file and performing the transformation as a table.
Once the RDD is a table, you use the sql method provided by SQLContext to run SQL statements.
The example here selects from the people table, the schemaRDD.
Finally, the results that comes out from the select statement is also a SchemaRDD.
That RDD, teenagers on our slide, can run normal RDD operations.
The programmatic interface is used when cannot define the case classes ahead of time. For example, when the structure of records is encoded in a string or a text dataset will be parsed and fields will be projected different for different users.
A schemaRDD is created with three steps.
The first is to create an RDD of Rows from the original RDD. In the example, we create a schemaString of name and age.
The second step is to create the schema using the RDD from step one. The schema is represented
by a StructType that takes the schemaString from the first step and splits it up into StructFields. In the example, the schema is created using the StructType by splitting the name and age schemaString and mapping it to the StructFields, name and age.
The third step convert records of the RDD of people to Row objects. Then you apply the schema to the row RDD.
Once you have your SchemaRDD, you register that RDD as a table and then you run sql statements
against it using the sql method.
The results are returned as the SchemaRDD and you can run any normal RDD operation on it. In the example, we select the name from the people table, which is the SchemaRDD. Then we print it out on the console.

Spark Streaming

- Scalable, high-throughput, fault-tolerant stream processing of live data streams
- Receives live input data and divides into small batches which are processed and returned as batches
- DStream – sequence of RDD
- Currently supports Scala and Java
- Basic Python support available in Spark 1.2.

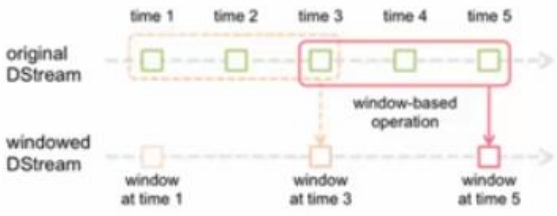
- Receives data from:
  - Kafka
  - Flume
  - HDFS / S3
  - Kinesis
  - Twitter
- Pushes data out to:
  - HDFS
  - Databases
  - Dashboard

Spark Streaming - Internals

- The input stream (DStream) goes into Spark Steaming
- Breaks up into batches
- Feeds into the Spark engine for processing
- Generate the final results in streams of batches

spark.apache.org

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



spark.apache.org

Spark Streaming – Getting started

- Scenario: Count the number of words coming in from the TCP socket.
- Import the Spark Streaming classes and some implicit conversions

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
```

- Create the StreamingContext object

```
val conf = new
SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

- Create a DStream

```
val lines = ssc.socketTextStream("localhost", 9999)
```

- Split the lines into words

```
val words = lines.flatMap(_.split(" "))
```

- Count the words

```
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
```

- Print to the console:

```
wordCounts.print()
```

Spark Streaming – Continued

- No real processing happens until you tell it:

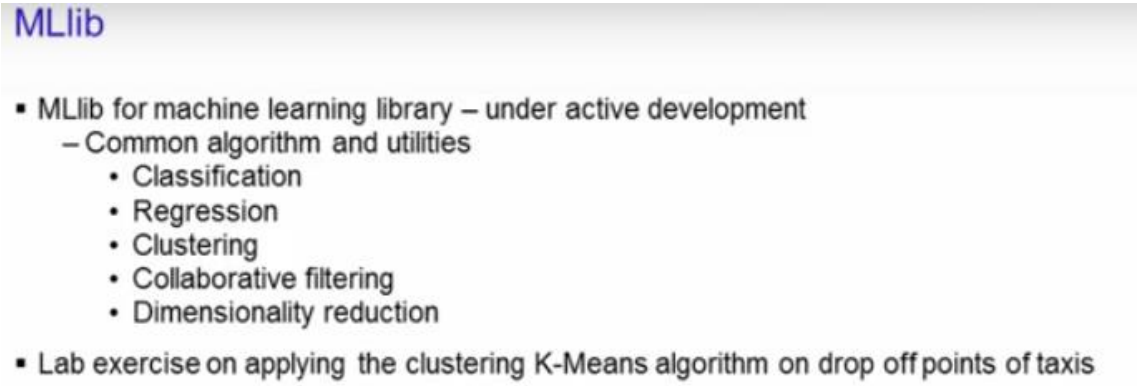
```
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

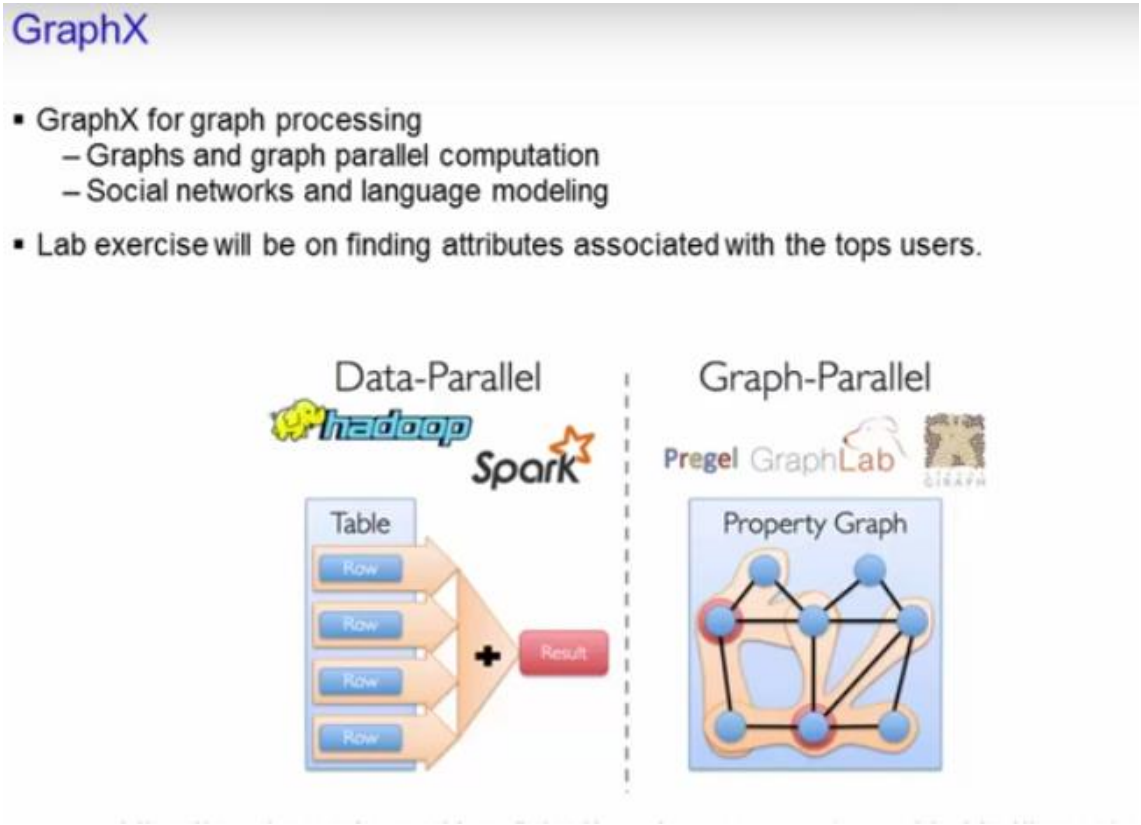
- The entire code and application can be found in the NetworkWordCount example
- Run the full example:
  - Run netcat to start the data stream
  - In a different terminal, run the application

```
./bin/run-example streaming.NetworkWordCount localhost 9999
```

Spark streaming gives you the capability to process live streaming data in small batches. Utilizing Spark's core, Spark Streaming is scalable, high-throughput and fault-tolerant.
You write Stream programs with DStreams, which is a sequence of RDDs from a stream of data.
There are various data sources that Spark Streaming receives from including, Kafka, Flume, HDFS, Kinesis, or Twitter. It pushes data out to HDFS, databases, or some sort of dashboard.
Spark Streaming supports Scala, Java and Python. Python was actually introduced with Spark 1.2. Python has all the transformations that Scala and Java have with DStreams, but it can only support text data types. Support for other sources such as Kafka and Flume will be available in future releases for Python.
Here's a quick view of how Spark Streaming works. First the input stream comes in to Spark Streaming. Then that data stream is broken up into batches of data that is fed into the Spark engine for processing. Once the data has been processed, it is sent out in batches.
Spark Stream support sliding window operations. In a windowed computation, every time the

window slides over a source of DStream, the source RDDs that falls within the window are combined and operated upon to produce the resulting RDD.
There are two parameters for a sliding window. The window length is the duration of the window
and the sliding interval is the interval in which the window operation is performed. Both of these must be in multiples of the batch interval of the source DStream.
In the diagram, the window length is 3 and the sliding interval is 2. To put it in a different perspective, say you wanted to generate word counts over last 30 seconds of data, every 10 seconds. To do this, you would apply the reduceByKeyAndWindow operation on the pairs of DStream of (Word,1) pairs over the last 30 seconds of data.
Here we a have a simple example. We want to count the number of words coming in from the TCP socket. First and foremost, you must import the appropriate libraries. Then, you would create the StreamingContext object. In it, you specify to use two threads with the batch interval of 1 second. Next, you create the DStream, which is a sequence of RDDs, that listens to the TCP socket on the given hostname and port. Each record of the DStream is a line of text. You split up the lines into individual words using the flatMap function.
The flatMap function is a one-to-many DStream operation that takes one line and creates a new DStream by generating multiple records (in our case, that would be the words on the line). Finally, with the words DStream, you can count the words using the map and reduce model. Then you print out the results to the console.
One thing to note is that when each element of the application is executed, the real processing doesn't actually happen yet. You have to explicitly tell it to start. Once the application begins, it will continue running until the computation terminates. The code snippets that you see here is from a full sample found in the NetworkWordCount. To run the full example, you must first start
up netcat, which is a small utility found in most Unix-like systems. This will act as a data source to give the application streams of data to work with. Then, on a different terminal window, run the application using the command shown here.





Here is a really short overview of the machine learning library. The MLlib library contains algorithms and utilities for classification, regression, clustering, collaborative filtering and dimensionality reduction. Essentially, you would use this for specific machine learning use cases that requires these algorithms. In the lab exercise, you will use the clustering K-Means algorithm on a set of taxi drop off points to figure out potentially where the best place to hail a cab would be.

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

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

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

The lab exercise goes through an example of loading in a text file and creating a graph from it to find attributes of the top users.

Having completed this lesson, you should be able to understand and use the various Spark libraries.

Complete lab exercise #4, creating applications using Spark SQL, MLib, Spark Streaming, and GraphX and then proceed to the next lesson in this course.

>>Lab:

Let's first download the data that we will be working with in this lab

```
]: // download module to run shell commands within this notebook
   import sys.process._
```

The data is in a folder called **LabData**. Let's list all the files in the data that we just downloaded and extracted.

```
// list the extracted files
"ls -1 /resources/jupyterlab/labs/BD0211EN/LabData/" !
```

```
Name: Compile Error
Message: <console>:27: error: value ! is not a member of String
       "ls -1 /resources/jupyterlab/labs/BD0211EN/LabData/" !
                                                            ^

StackTrace:
```

Let's take a look at the nycweather data. So run the following code:

```
val lines = scala.io.Source.fromFile("/resources/jupyterlab/labs/BD0211EN/LabData/nycweather.csv").mkString

println(lines)
```

```
"2013-01-01",1,0
"2013-01-02",-2,0
"2013-01-03",-2,0
```

There are three columns in the dataset, the date, the mean temperature in Celsius, and the precipitation for the day. Since we already know the schema, we will infer the schema using reflection.

You will first need to define the SparkSQL context. Do so by creating it from an existing SparkContext. Type in:

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

```
sqlContext = org.apache.spark.sql.SQLContext@5fb964b1
warning: there was one deprecation warning; re-run with -deprecation for details
org.apache.spark.sql.SQLContext@5fb964b1
```

Next, you need to import a library for creating a SchemaRDD. Type this:

Create a case class in Scala that defines the schema of the table. Type in:

```
case class Weather(date: String, temp: Int, precipitation: Double)
```

```
defined class Weather
```

Create the RDD of the Weather object:

```
val weather = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/nycweather.csv").map(_.split(",")). map(w => Weather(w(0), w(1).trim.toInt, w(2).trim.toDouble)).toDF()
```

```
weather = [date: string, temp: int ... 1 more field]
[date: string, temp: int ... 1 more field]
```

You first load in the file, and then you map it by splitting it up by the commas and then another mapping to get it into the Weather class.

Next you need to register the RDD as a table. Type in:

```
weather.registerTempTable("weather")
```

```
warning: there was one deprecation warning; re-run with -deprecation for details
```

At this point, you are ready to create and run some queries on the RDD. You want to get a list of the hottest dates with some precipitation. Type in:

```
val hottest_with_precip = sqlContext.sql("SELECT * FROM weather WHERE precipitation > 0.0 ORDER BY temp DESC")

hottest_with_precip.collect()
```

```
hottest_with_precip = [date: string, temp: int ... 1 more field]
```

Creating a Spark application using MLlib

In this section, Spark will be used to acquire the K-Means clustering for drop-off latitudes and longitudes of taxis for 3 clusters. The sample data contains a subset of taxi trips with hack license, medallion, pickup date/time, drop off date/time, pickup/drop off latitude/longitude, passenger count, trip distance, trip time and other information. As such, this may give a good indication of where to best to hail a cab.

Remember, this is only a subset of the file that you used in a previous exercise. If you ran this exercise on the full dataset, it would take a long time as we are only running on a test environment with limited resources.

Import the needed packages for K-Means algorithm and Vector packages:

```
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors
```

Create an RDD

```
val taxiFile = sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/nyctaxisub.csv")
```

```
taxiFile = /resources/jupyterlab/labs/BD0211EN/LabData/nyctaxisub.csv MapPartitionsRDD[14] at textFile at <console>:34
/resources/jupyterlab/labs/BD0211EN/LabData/nyctaxisub.csv MapPartitionsRDD[14] at textFile at <console>:34
```

Determine the number of rows in taxiFile.

```
taxiFile.count()
```

Cleanse the data.

```
val taxiData=taxiFile.filter(_.contains("2013")).
    filter(_.split(",")(3)!="" ).    //dropoff_latitude
    filter(_.split(",")(4)!="")      //dropoff_longitude
```

```
taxiData = MapPartitionsRDD[17] at filter at <console>:36
MapPartitionsRDD[17] at filter at <console>:36
```

The first filter limits the rows to those that occurred in the year 2013. This will also remove any header in the file. The third and fourth columns contain the drop off latitude and longitude. The transformation will throw exceptions if these values are empty.

Do another count to see what was removed.

```
taxiData.count()
```

In this case, if we had used the full set of data, it would have filtered out a great many more lines.

To fence the area roughly to New York City use this command:

```
val taxiFence=taxiData.
    filter(_.split(",")(3).toDouble>40.70).
    filter(_.split(",")(3).toDouble<40.86).
    filter(_.split(",")(4).toDouble>(-74.02)).
    filter(_.split(",")(4).toDouble<(-73.93))
```

```
taxiFence = MapPartitionsRDD[21] at filter at <console>:38
MapPartitionsRDD[21] at filter at <console>:38
```

Determine how many are left in taxiFence:

```
taxiFence.count()
```

```
206646
```

Create Vectors with the latitudes and longitudes that will be used as input to the K-Means algorithm.

```
val taxi=taxiFence.
    map{
        line=>Vectors.dense(
            line.split(',').slice(3,5).map(_ .toDouble)
        )
    }
```

```
taxi = MapPartitionsRDD[22] at map at <console>:35
MapPartitionsRDD[22] at map at <console>:35
```

```
val iterationCount=10
val clusterCount=3

val model=KMeans.train(taxi,clusterCount,iterationCount)
val clusterCenters=model.clusterCenters.map(_.toArray)

clusterCenters.foreach(lines=>println(lines(0),lines(1)))
```

```
(40.72458257012865,-73.99591140564232)
(40.78696784862365,-73.95717886418726)
(40.75683479531887,-73.98099751962766)
iterationCount = 10
clusterCount = 3
model = org.apache.spark.mllib.clustering.KMeansModel@900ad81
clusterCenters = Array(Array(40.72458257012865, -73.99591140564232), Array(40.78696784862365, -73.95717886418726), Array(40.75683479531887, -73.98099751962766))
Array(Array(40.72458257012865, -73.99591140564232), Array(40.78696784862365, -73.95717886418726), Array(40.75683479531887, -73.98099751962766))
```

Creating a Spark application using Spark Streaming

This section focuses on Spark Streams, an easy to build, scalable, stateful (e.g. sliding windows) stream processing library. Streaming jobs are written the same way Spark batch jobs are coded and support Java, Scala and Python. In this exercise, taxi trip data will be streamed using a socket connection and then analyzed to provide a summary of number of passengers by taxi vendor. This will be implemented in the Spark shell using Scala.

There are two relevant files for this section. The first one is the nyctaxi100.csv which will serve as the source of the stream. The other file is a python file, taxistreams.py, which will feed the csv file through a socket connection to simulate a stream.

```
import org.apache.log4j.Logger
import org.apache.log4j.Level
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)

import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
```

Create the StreamingContext by using the existing SparkContext (sc). It will be using a 1 second batch interval, which means the stream is divided to 1 second batches and each batch becomes a RDD. This is intentional to make it easier to read the data during execution.

```
val ssc = new StreamingContext(sc, Seconds(1))
```

Create the StreamingContext by using the existing SparkContext (sc). It will be using a 1 second batch interval, which means the stream is divided to 1 second batches and each batch becomes a RDD. This is intentional to make it easier to read the data during execution.

```
val ssc = new StreamingContext(sc, Seconds(1))
```

```
ssc = org.apache.spark.streaming.StreamingContext@1897f49d
org.apache.spark.streaming.StreamingContext@1897f49d
```

Create the socket stream that connects to the localhost socket 7777. This matches the port that the Python script is listening on. Each batch from the Stream be a lines RDD.

```
val lines = ssc.socketTextStream("localhost", 7777)
```

```
lines = org.apache.spark.streaming.dstream.SocketInputDStream@4aa9db38
org.apache.spark.streaming.dstream.SocketInputDStream@4aa9db38
```

Next, put in the business logic to split up the lines on each comma and mapping pass(15), which is the vendor, and pass(7), which is the passenger count. Then this is reduced by key resulting in a summary of number of passengers by vendor.

```
val pass = lines.map(_.split(",")).
    map(pass=>(pass(15), pass(7).toInt)).
    reduceByKey(_+_)
```

```
pass = org.apache.spark.streaming.dstream.ShuffledDStream@15f253f1
org.apache.spark.streaming.dstream.ShuffledDStream@15f253f1
```

Print out to the console:

```
pass.print()
```

The next two line starts the stream.

```
ssc.start()
ssc.awaitTermination()
```

It will take a few cycles for the connection to be recognized, and then the data is sent. In this case, 2 rows per second of taxi trip data is receive in a 1 second batch interval.

Creating a Spark application using GraphX

Users.txt is a set of users and followers is the relationship between the users. Take a look at the contents of these two files.

```
println("Users: ")
println(scala.io.Source.fromFile("/resources/jupyterlab/labs/BD0211EN/LabData/users.txt").mkString)

println("Followers: ")
println(scala.io.Source.fromFile("/resources/jupyterlab/labs/BD0211EN/LabData/followers.txt").mkString)
```

Import the GraphX package:

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

Create the users RDD and parse into tuples of user id and attribute list:

```
val users = (sc.textFile("/resources/jupyterlab/labs/BD0211EN/LabData/users.txt").map(line => line.split(",")).map(parts => (parts.head.toLong, parts.tail)))

users.take(5).foreach(println)
```

Parse the edge data, which is already in userId -> userId format

```
val followerGraph = GraphLoader.edgeListFile(sc, "/resources/jupyterlab/labs/BD0211EN/LabData/followers.txt")
```

Attach the user attributes

```
val graph = followerGraph.outerJoinVertices(users) {
    case (uid, deg, Some(attrList)) => attrList
    case (uid, deg, None) => Array.empty[String]
}
```

Restrict the graph to users with usernames and names:

```
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)
```

Compute the PageRank

```
val pagerankGraph = subgraph.pageRank(0.001)
```

Get the attributes of the top pagerank users

```
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices) {
    case (uid, attrList, Some(pr)) => (pr, attrList.toList)
    case (uid, attrList, None) => (0.0, attrList.toList)
}
```

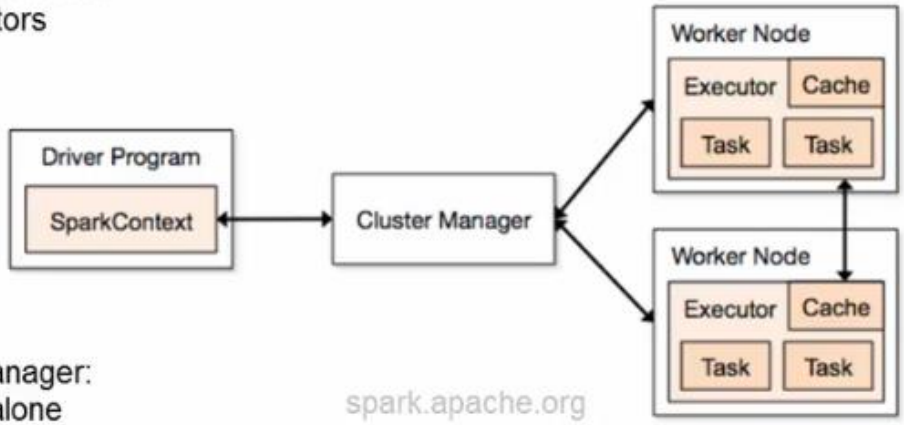
Print the line out:

```
println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString("\n"))
```

MODULE 5 – SPARK CONFIGURATION, MONITORING AND TUNING

Spark cluster overview

- Components
  - Driver
  - Cluster Master
  - Executors

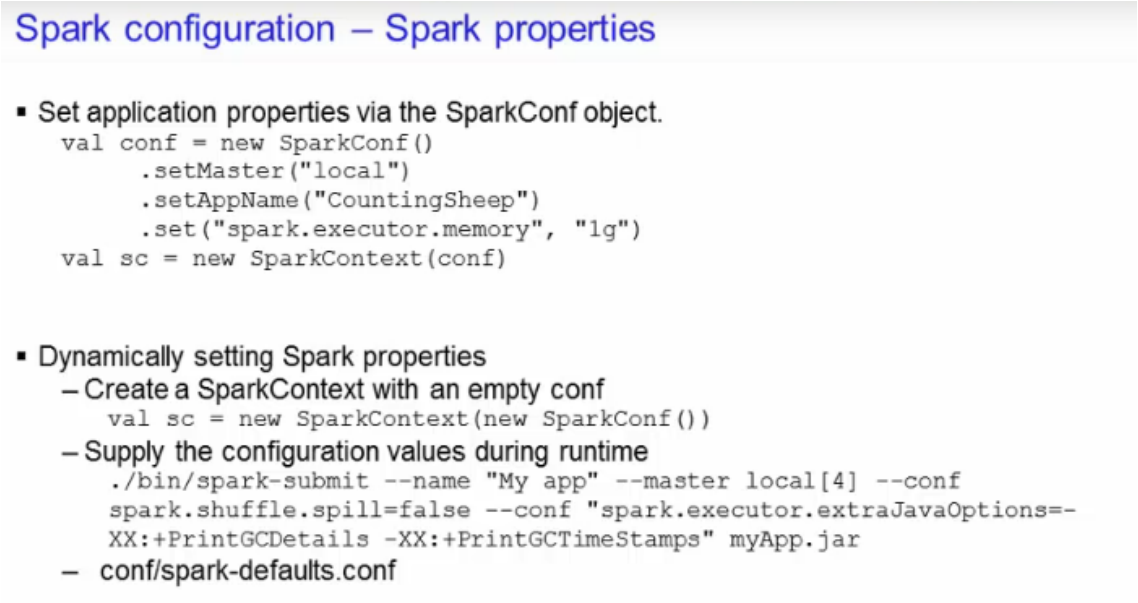


- Cluster manager:
  - Standalone
  - Apache Mesos
  - Hadoop YARN

spark.apache.org

Spark configuration

- Three locations for configuration:
  - Spark properties
  - Environment variables
    - `conf/spark-env.sh`
  - Logging
    - `log4j.properties`
- Override default configuration directory (`SPARK_HOME/conf`)
  - `SPARK_CONF_DIR`
    - spark-defaults.conf
    - spark-env.sh
    - log4j.properties
    - etc.
- Spark shell can be verbose
  - To view ERRORs only, changed the INFO value to ERROR in the log4j.properties
    - `$SPARK_HOME/conf/log4j.properties`



rt 1 (5:26)
Skip to a navigable version of this video's transcript.
5:18 / 5:26
Maximum Volume.
Skip to end of transcript.

Hi. Welcome to the Spark Fundamentals course. This lesson will cover Spark configuration, monitoring and tuning.

After completing this lesson, you should be able describe the cluster overview. Configure Spark by modifying Spark properties, environmental variables or logging properties. Monitor Spark and its applications using the Web UIs, metrics and various other external tools. Also covered in this lesson would be some performance tuning considerations.

There are three main components of a Spark cluster. You have the driver, where the SparkContext is located within the main program. To run on a cluster, you would need some sort of cluster manager. This could be either Spark's standalone cluster manager, Mesos or Yarn.

Then you have your worker nodes where the executor resides. The executors are the processes that run computations and store the data for the application. The SparkContext sends the application, defined as JAR or Python files to each executor. Finally, it sends the tasks for each executor to run.

Several things to understand about this architecture. Each application gets its own executor. The executor stays up for the entire duration of the application. The benefit of this is that the applications are isolated from each other, on the scheduling side and running on different JVMs. However, this means that you cannot share data across applications. You would need to externalize the data if you wish to share data between thedifferent applications.

Spark applications don't care about the underlying cluster manager. As long as it can acquire executors and communicate with each other, it can run on any cluster manager.

Because the driver program schedules tasks on the cluster, it should run close to the worker nodes on the same local network. If you like to send remote requests to the cluster,

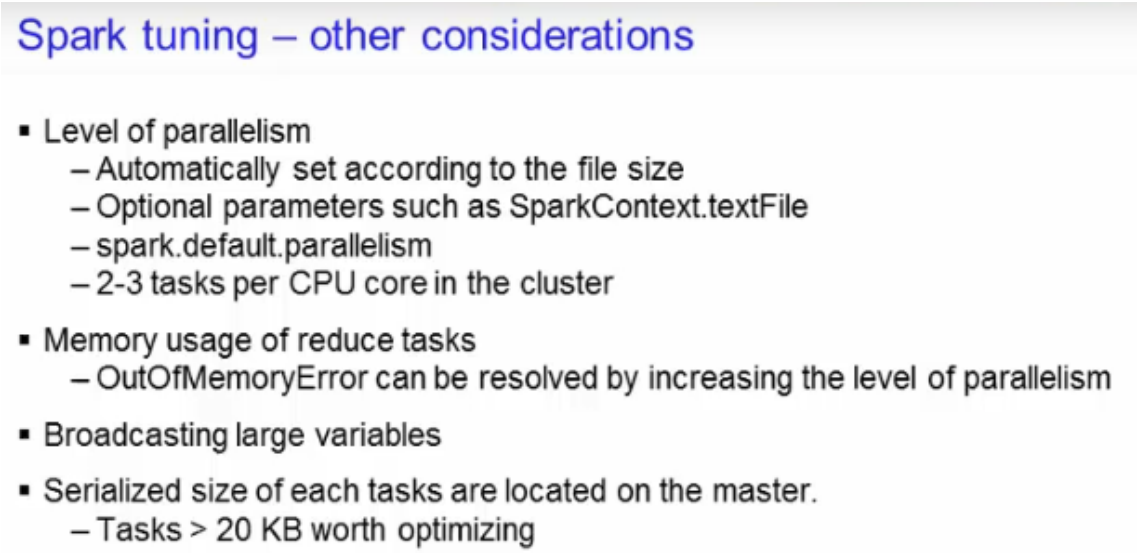
it is better to use a RPC and have it submit operations from nearby.
There are currently three supported cluster managers that we have mentioned before. Sparks
comes with a standalone manager that you can use to get up and running. You can use Apache
Mesos, a general cluster manager that can run and service Hadoop jobs. Finally, you
can also use Hadoop YARN, the resource manager in Hadoop 2. In the lab exercise, you will
be using BigInsights with Yarn to run your Spark applications.
Spark configuration. There are three main locations for Spark configuration.
You have the Spark properties, where the application parameters can be set using the SparkConf
object or through Java system properties.
Then you have the environment variables, which can be used to set per machine settings such
as IP address. This is done through the conf/spark-env.sh script on each node.
Finally, you also have your logging properties, which can be configured through log4j.properties.
You can choose to override the default configuration directory, which is currently under the SPARK_HOME/conf
directory. Set the SPARK_CONF_DIR environment variable and provide your custom configuration
files under that directory.
In the lab exercise, the Spark shell can be verbose, so if you wish, change it from INFO
to ERROR in the log4j.properties to reduce all the information being printed on the console.
There are two methods of setting Spark properties. The first method is by passing application
properties via the SparkConf object. As you know, the SparkConf variable is used to create
the SparkContext object. In the example shown on this slide, you set the master node as
local, the appName as "CountingSheep", and you allow 1GB for each of the executor processes.
The second method is to dynamically set the Spark properties. Spark allows you to pass
in an empty SparkConf when creating the SparkContext as shown on the slide.
You can then either supply the values during runtime by using the command line options
--master or the --conf. You can see the list of options using the --help when executing
the spark-submit script. On the slide here, you give the app name of
My App and telling it to run on the local system with four cores. You set the spark.shuffle.spill
to false and the various java options at the end. Finally you supply the application JAR
file after all the properties have been specified.
You can find a list of all the properties on the spark.apache.org website.
Another way to set Spark properties is to provide your settings inside the spark-defaults.conf
file. The spark-submit script will read in the configurations from this file. You can
view the Spark properties on the application web UI at the port 4040 by default.
One thing I'll add is that properties set directly on the SparkConf take highest precedence,
then flags passed to spark-submit or spark-shell is second and finally options in the spark-defaults.conf
file is the lowest priority.

Spark monitoring

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

Spark monitoring – Web UI / history server

- Port 4040
- Shows current application
- Contains the following information
  - A list of scheduler stages and tasks
  - A summary of RDD sizes and memory usage
  - Environmental information.
  - Information about the running executors
- Viewing the history (on Mesos or YARN): `./sbin/start-history-server.sh`
- Configure the history server to set
  - Memory allocated
  - JVM options
  - Public address for the server
  - Various properties



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

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

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

The Web UI is found on port 4040, by default, and shows the information for the current application while it is running.

The Web UI contains the following information. A list of scheduler stages and tasks.

A summary of RDD sizes and memory usage. Environmental information and information about the running executors.

To view the history of an application after it has ran, you can start up the history server.

The history server can be configured on the amount of memory allocated for it, the various JVM options, the public address for the server, and a number of properties.

You will see all of this in the lab exercise.

Spark programs can be bottlenecked by any resource in the cluster. Due to Spark's nature of the in-memory computations, data serialization and memory tuning are two areas that will improve performance. Data serialization is crucial for network performance and to reduce memory use. It is often the first thing you should look at when tuning Spark applications.

Spark provides two serialization libraries. Java serialization provides a lot more flexibility, but it is quiet slow and leads to large serialized objects. This is the default library that Spark uses to serialize objects. Kyro serialization is much quicker than Java, but does not support all Serializable types. It would require you to register these types in advance for best performance. To use Kyro serialization, you can set it using the SparkConf object.

With memory tuning, you have to consider three things. The amount of memory used by the objects (whether or not you want the entire object to fit in memory). The cost of accessing those objects and the overhead garbage collection.

You can determine how much memory your dataset requires by creating a RDD, put it into cache, and look at the SparkContext log on your driver program. Examining that log will show you how much memory your dataset uses.

Few tips to reduce the amount of memory used by each object. Try to avoid Java features that add overhead such as pointer based data structures and wrapper objects. If possible go with arrays or primitive types and try to avoid nested structures.

Serialized storage can also help to reduce memory usage. The downside would be that it will take longer to access the object because you have to deserialized it before you can use it.

You can collect statistics on the garbage collection to see how frequently it occurs and the amount of time spent on it. To do so, add the line to the SPARK_JAVA_OPTS environment variable.

The level of parallelism should be considered in order to fully utilize your cluster. It is automatically set to the file size of the task, but you can configure this through optional parameters such as in the SparkContext.textFile. You can also set the default level in the spark.default.parallelism config property. Generally, it is recommended to set 2-3 tasks per CPU core in the cluster.

Sometimes when your RDD does not fit in memory, you will get an OutOfMemoryError. In cases like this, often by increasing the level of parallelism will resolve this issue. By increasing the level, each set of task input will be smaller, so it can fit in memory.

Using Spark's capability to broadcast large variables greatly reduces the size of the serialized object. A good example would be if you have some type of static lookup table. Consider turning that into a broadcast variable so it does not need to be passed on to each of the worker nodes.

Spark prints the serialized size of each tasks in the master. Check that out to examine if any tasks are too large. If you see some tasks larger than 20KB, it's worth taking a look to see if you can optimize it further, such as creating broadcast variables.

Having completed this lesson, you should be able to describe the cluster overview. You

should also know where and how to set Spark configuration properties. You also saw how to monitor Spark using the UI, metrics or various external tools. Finally, you should understand some performance tuning considerations.

Next steps. Complete lab exercise #5 and then, congratulations, you have completed this course.