Contents

[Introduction to Spark - Getting started 3](#_Toc88635880)

[What is Spark and what is its purpose? 3](#_Toc88635881)

[Speed 3](#_Toc88635882)

[Generality 3](#_Toc88635883)

[Ease of use 4](#_Toc88635884)

[Who uses Spark and why? 4](#_Toc88635885)

[Data scientist 4](#_Toc88635886)

[Engineers 4](#_Toc88635887)

[Everyone else 5](#_Toc88635888)

[Components of the Spark unified stack 5](#_Toc88635889)

[Spark Core 5](#_Toc88635890)

[Spark SQL 6](#_Toc88635891)

[Spark Streaming 6](#_Toc88635892)

[MLlib 6](#_Toc88635893)

[GraphX 6](#_Toc88635894)

[Brief History of Spark 6](#_Toc88635895)

[About 6](#_Toc88635896)

[Resilient Distributed Dataset (RDD) 7](#_Toc88635897)

[Types of operations 8](#_Toc88635898)

[Example of RDD flow. 8](#_Toc88635899)

[Downloading and installing Spark standalone 9](#_Toc88635900)

[Scala and Python overview 10](#_Toc88635901)

[Anonymous functions 10](#_Toc88635902)

[Launching and using Spark’s Scala and Python shell © 12](#_Toc88635903)

[Resilient Distributed Dataset and DataFrames 12](#_Toc88635904)

[Understand how to create parallelized collections and external datasets 12](#_Toc88635905)

[Parallelizing an existing collection 12](#_Toc88635906)

[Referencing a dataset 12](#_Toc88635907)

[Transformation from an existing RDD 13](#_Toc88635908)

[Creating an RDD 14](#_Toc88635909)

[RDD operations – Basics 15](#_Toc88635910)

[Work with Resilient Distributed Dataset (RDD) operations 15](#_Toc88635911)

[Direct Acyclic Graph (DAG) 15](#_Toc88635912)

[What happens when an action is executed? 16](#_Toc88635913)

[RDD operations – Transformations 20](#_Toc88635914)

[RDD operations – Actions 21](#_Toc88635915)

[Utilize shared variables and key-value pairs 22](#_Toc88635916)

[RDD persistence 22](#_Toc88635917)

[What storage level to choose? 23](#_Toc88635918)

[Shared variables and key-value pairs 24](#_Toc88635919)

[Programming with key-value pairs 25](#_Toc88635920)

[Example: 26](#_Toc88635921)

[Spark application programming 26](#_Toc88635922)

[Understand the purpose and usage of the SparkContext 26](#_Toc88635923)

[Initialize Spark with the various programming languages 27](#_Toc88635924)

[Scala 27](#_Toc88635925)

[Python 28](#_Toc88635926)

[Java 29](#_Toc88635927)

[Describe and run some Spark examples 30](#_Toc88635928)

[Passing functions to Spark. 30](#_Toc88635929)

[Business logic 31](#_Toc88635930)

[Running Spark examples 32](#_Toc88635931)

[Pass functions to Spark 33](#_Toc88635932)

[Create and run a Spark standalone application 33](#_Toc88635933)

[Submit applications to the cluster 35](#_Toc88635934)

[Introduction to Spark libraries 36](#_Toc88635935)

[Understand and use the various Spark libraries 36](#_Toc88635936)

[SparkSQL 37](#_Toc88635937)

[Spark Streaming 40](#_Toc88635938)

[MLib 42](#_Toc88635939)

[GraphX 43](#_Toc88635940)

[Spark configuration, monitoring, and tuning 44](#_Toc88635941)

[Understand components of the Spark cluster 44](#_Toc88635942)

[Driver 44](#_Toc88635943)

[Cluster Managers 45](#_Toc88635944)

[Configure Spark to modify the Spark properties, environmental variables, or logging properties 45](#_Toc88635945)

[Monitor Spark using the web UIs, metrics, and external instrumentation 46](#_Toc88635946)

[Understand performance tuning considerations 48](#_Toc88635947)

# Introduction to Spark - Getting started

## What is Spark and what is its purpose?

* Data is increasing in volume, velocity, variety.
* The need to have faster results from analytics becomes increasingly important.
* Apache Spark is a computing platform designed to be fast and general-purpose, and easy to use

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.

### Speed

* In-memory computations
* Faster than MapReduce for complex applications on disk

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.

Q) What gives Spark its speed advantage for complex applications?

* Various libraries provide Spark with additional functionality
* Spark extends the MapReduce model
* Spark can cover a wide range of workloads under one system
* Spark makes extensive use of in-memory computations
* All of the above

### Generality

* Covers a wide range of workloads on one system
* Batch applications (e.g. MapReduce)
* Iterative algorithms
* Interactive queries and streaming

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.

### Ease of use

* APIs for Scala, Python, Java
* Libraries for SQL, machine learning, streaming, and graph processing
* Runs on Hadoop clusters or as a standalone

You can also run interactive queries and process streaming data with your application. In a later slide, you'll see that there are several 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.

## Who uses Spark and why?

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.

* Parallel distributed processing, fault tolerance on commodity hardware, scalability, in-memory computing, high level APIs, etc.
* Saves time and money

### Data scientist

* + Analyze and model the data to obtain insight using ad-hoc analysis
  + Transforming the data into a useable format
  + Statistics, machine learning, SQL

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.

### Engineers

* + Develop a data processing system or application
  + Inspect and tune their applications
  + Programming with the Spark's API

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.

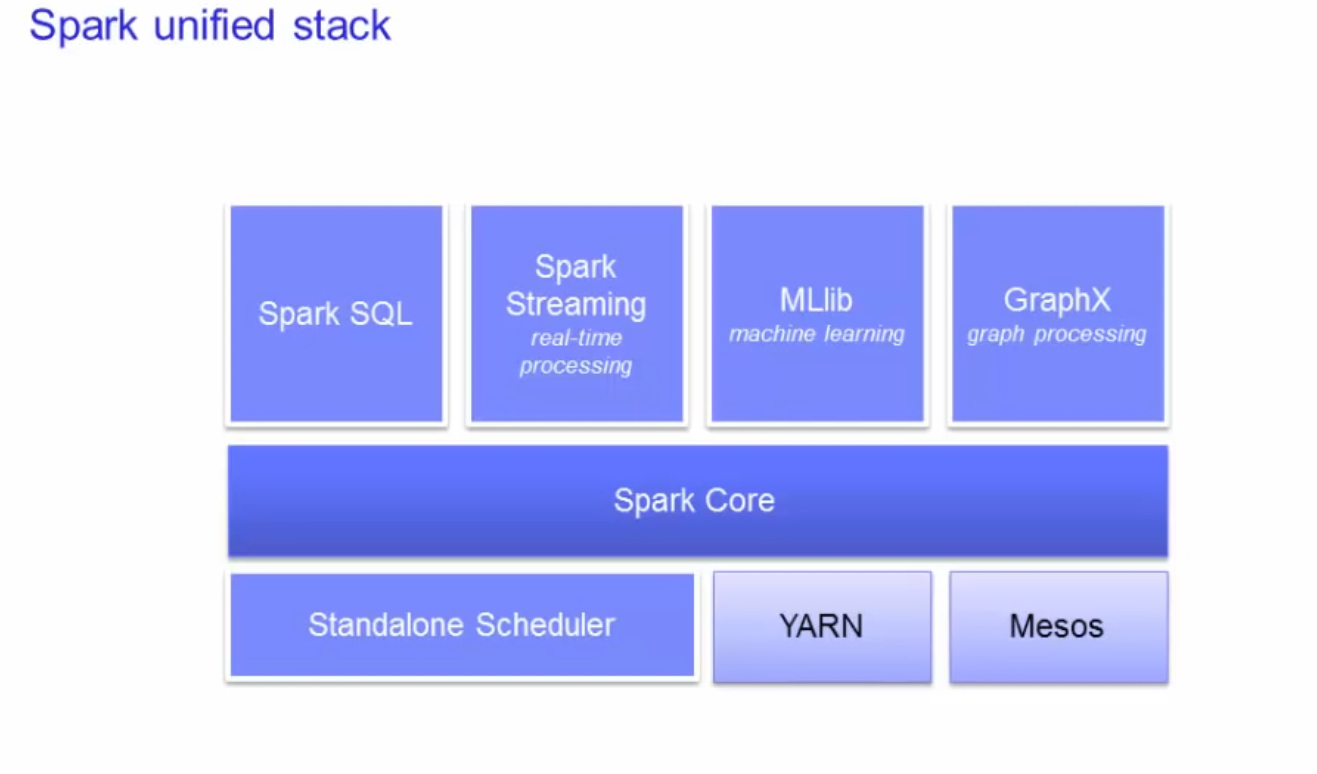
Q) For what purpose would an Engineer use Spark? Select all that apply.

* Analyzing data to obtain insights
* Programming with Spark’s API
* Transforming data into a useable form for analysis
* Developing a data processing system
* Tuning an application for a business use case

### Everyone else

* + Ease of use
  + Wide variety of functionality
  + Mature and reliable

## Components of the Spark unified stack



Here we have a picture of the Spark unified stack.

### Spark Core

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

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

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.

### MLlib

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

### GraphX

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

## Brief History of Spark

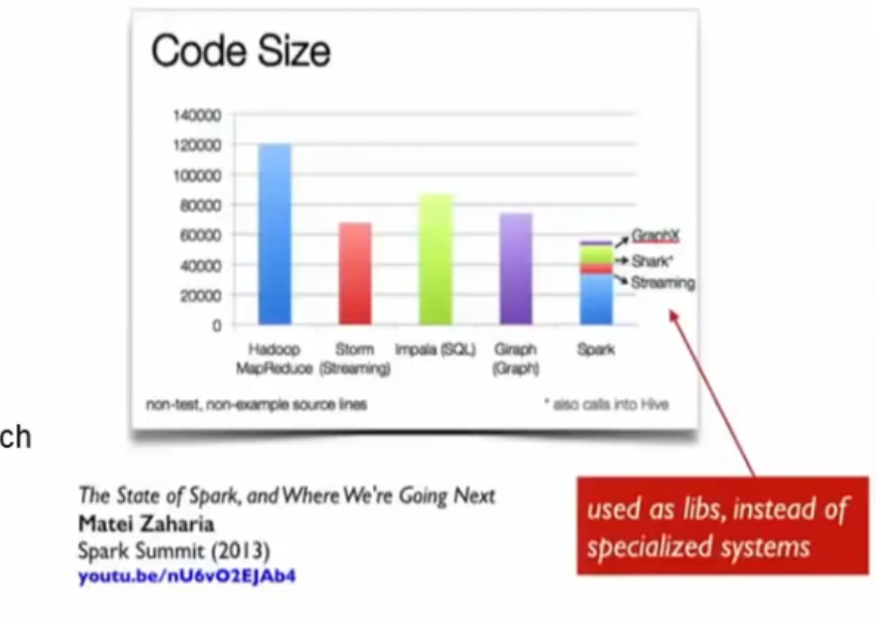
Here's a brief history of Spark. I'm not going to spend too much time on this as you can easily find more information for yourself if you are interested.

* 2002 — MapReduce @ Google Code Size
  + 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.
* 2004 — MapReduce paper
* 2006 — Hadoop @ Yahoo
* 2008 — Hadoop Summit
* 2010 — Spark paper
  + Spark comes out about a decade later with the similar framework to run data processing on commodity systems also using a fault tolerant framework.
* 2014 — Apache Spark top-level

### About

* MapReduce started a general batch processing paradigm
* Two limitations:
  + Difficulty programming in MapReduce
  + Batch processing did not fit many use cases
* Spawned a lot of specialized systems
  + Storm
  + Impala
  + Giraph
  + etc.

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.



## Resilient Distributed Dataset (RDD)

Q) Which of the following statements are true of the Resilient Distributed Dataset (RDD)? Select all that apply.

* There are three types of RDD operations.
* RDDs allow Spark to reconstruct transformations
* RDDs only add a small amount of code due to tight integration
* RDD action operations do not return a value
* RDD is a distributed collection of elements parallelized across the cluster.
* Spark's primary abstraction
* Distributed collection of elements
* Parallelized across the cluster
* Fault tolerance
  + allows Spark to reconstruct the transformations used to build the lineage to get back the lost data
* Caching
  + 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.

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.

### Types of operations

#### Transformations

* Creates a DAG
* Lazy evaluations
* No return value

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

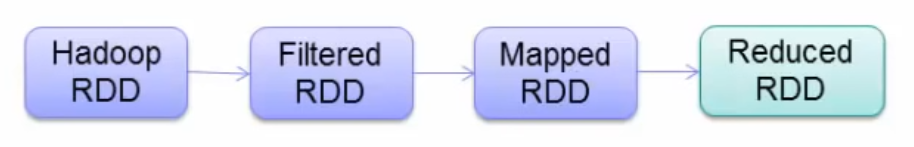
#### Actions

* Performs the transformations and the action that follows
* Returns a value

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

### Example of RDD flow.

Hadoop RDD => Filtered RDD => Mapped RDD => Reduced RDD



So, you have an image of a base RDD shown here on the slide. The first step is loading the dataset from Hadoop. Then you apply successive transformations on it such as filter, map, or reduce. Nothing 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.

## Downloading and installing Spark standalone

* Visit this page to download: [hitp://spark.apache.org/downloads html](hitp://spark.apache.org/downloads%20html)
  + 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 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.

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 backgrounds 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 functionalities 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.

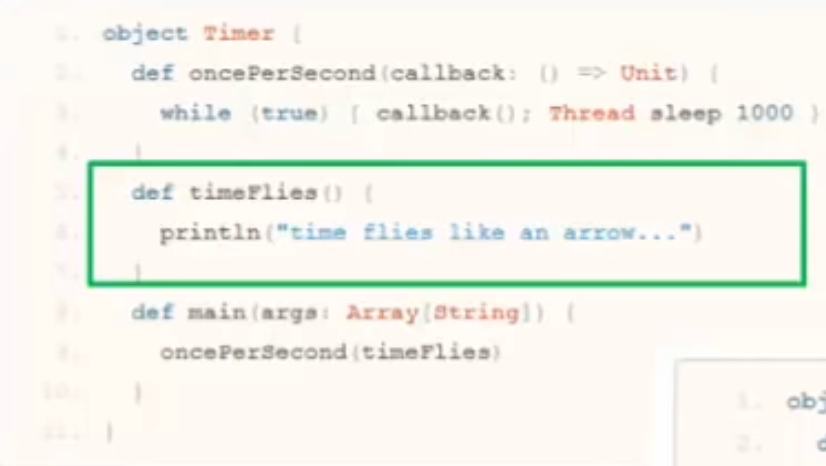
### Scala and Python overview

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. 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.

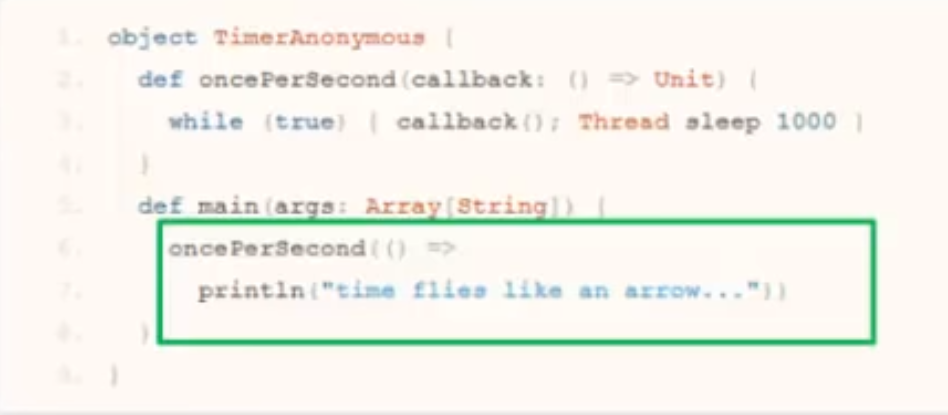
* Everything is an Object:
  + Primitive types such as numbers or Boolean
  + Functions
* Numbers are objects
  + -—1+2\*3/4 > (1).+(((2).\*(3))./(x)))
  + Where the +, \*, / are valid identifiers in Scala
* Functions are objects
  + Pass functions as arguments
  + Store them in variables
  + Return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

### 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)



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. Runs on both Windows and Unit-like systems (e.g Linux, Mac OS)



## Launching and using Spark’s Scala and Python shell ©

* Spark's shell provides a simple way to learn the API
* Powerful tool to analyze data interactively.
* The Scala shell runs on the Java VM
  + A good way to use existing Java libraries
* Scala:
  + To launch the Scala shell:
  + ./bin/spark-shell
  + To read in a text file:
  + scala> val textFile = sc.textFile("README.md")
* Python:
  + To launch the Python shell:
    - ./bin/pyspark
  + To readin a text file:
    - >>> textFile = sc.textFile ("README.md")

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.

# Resilient Distributed Dataset and DataFrames

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.

## Understand how to create parallelized collections and external datasets

* Fault-tolerant collection of elements that can be operated on in parallel.
* Immutable

### Parallelizing an existing collection

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.

### Referencing a dataset

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.

### Transformation from an existing RDD

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.

Q) What happens when an action is executed?

* Executors prepare the data for operation in parallel
* Data is partitioned into different blocks across the cluster
* The driver sends code to be executed on each block
* A cache is created for storing partial results in memory
* All of the above

Two types of RDD operations

* Transformations
  + 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 happens until some action is called.
  + Remember, transformations return a pointer to the RDD created
* Actions
  + 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, actions return values that comes from the action.
* Dataset from any storage supported by Hadoop HDFS
  + Cassandra
  + HBase
  + Amazon S3
  + etc.
* Types of files supported:
  + Text files
  + SequenceFiles
  + Hadoop InputFormat

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

### Creating an RDD

Q) Which of the following methods can be used to create a Resilient Distributed Dataset (RDD)? Select all that apply.

* Creating a directed acyclic graph (DAG)
* Parallelizing an existing Spark collection
* Referencing a Hadoop-supported dataset
* Using data that resides in Spark
* Transforming an existing RDD to form a new one

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.

* Launch the Spark shell
  + ./bin/spark-shell
  + 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.
* Create some data
  + val data = 1 to 10000
  + Once the shell is up, create some data with values from 1 to 10,000.
* Parallelize that data (creating the RDD)
  + val distData = sc.parallelize(data)
  + 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 won’t create that RDD until some action is invoked on it
* Perform additional transformations or invoke an action on it.
  + distData.filter(...)
  + With this new RDD, you can perform additional transformations or actions on it such as the filter transformation
* Alternatively, create an RDD from an external dataset
  + val readmeFile = sc.textFile(“Readme.md”)
  + 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.

### RDD operations – Basics

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 a file
  + val lines = sc.textFile(“hdfs://data.txt”)
  + 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.
* Applying transformation
  + val lineLengths = lines.map(s => s.length)
  + 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.
* Invoking action
  + val totalLengths = lineLengths.reduce((a,b) => a + b)
  + 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.

#### MapReduce example:

A common example is a MapReduce word count.

* val wordCounts = textFile.flatMap(line => line.split (" "))
  + .map(word => (word, 1))
    - 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.
  + .reduceByKey((a,b) => a + b)
    - 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.
* wordCounts.collect()
  + Finally, you call the collect() function, which is an action, to have it print out all the words and its occurrences.

## Work with Resilient Distributed Dataset (RDD) operations

### Direct Acyclic Graph (DAG)

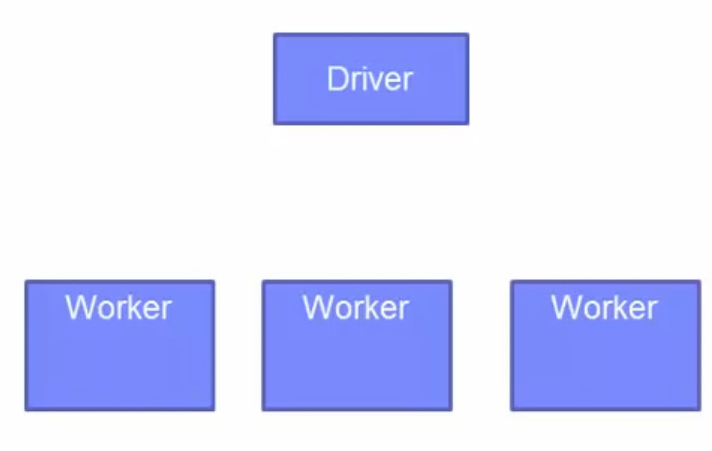
* View the DAG
  + linesLength.toDebugString
* Sample DAG
  + res5: String =
  + MappedRDD[4] at map at <console>:16 (3 partitions)
  + MappedRDD[3] at map at <console>:16 (3 partitions)
  + FilteredRDD[2] at filter at <console>:14 (3 partitions)
  + MappedRDD[1] at textFile at <console>:12 (3 partitions)
  + HadoopRDD[0] at textFile at <console>:12 (3 partitions)

You heard me mention direct acyclic graph several times now. On this slide, you will see how to view the DAG of any 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 must 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.

### What happens when an action is executed?

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.

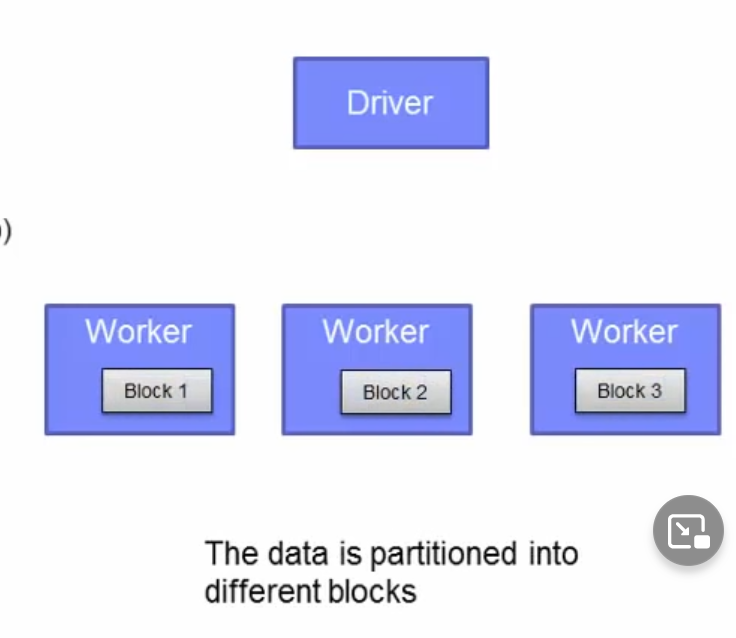
1. The first line you load the log from the hadoop file system.
2. The next two lines you filter out the messages within the log errors.
3. Before you invoke some action on it, you tell it to cache the filtered dataset - it doesn't cache it yet as nothing has been done up until this point.
4. Then you 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.

#### Create RDD

val logFile = sc.textFile(“hdfs://...”)

The first thing that happens when you load in the text file is the data is partitioned into different blocks across the cluster.

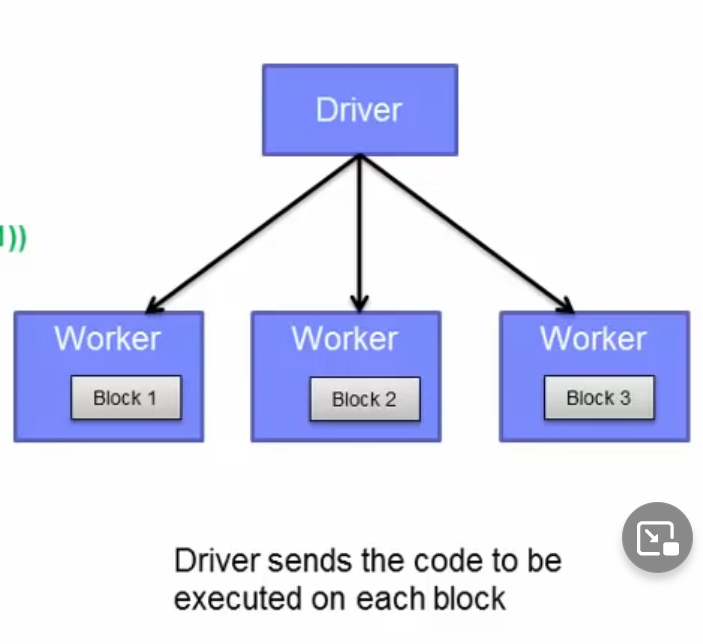


#### Transformations

val errors = logFile.filter(\_.startsWith(“ERROR’))

val messages = errors.map(\_.split(“\t’)).map(r => r(1))

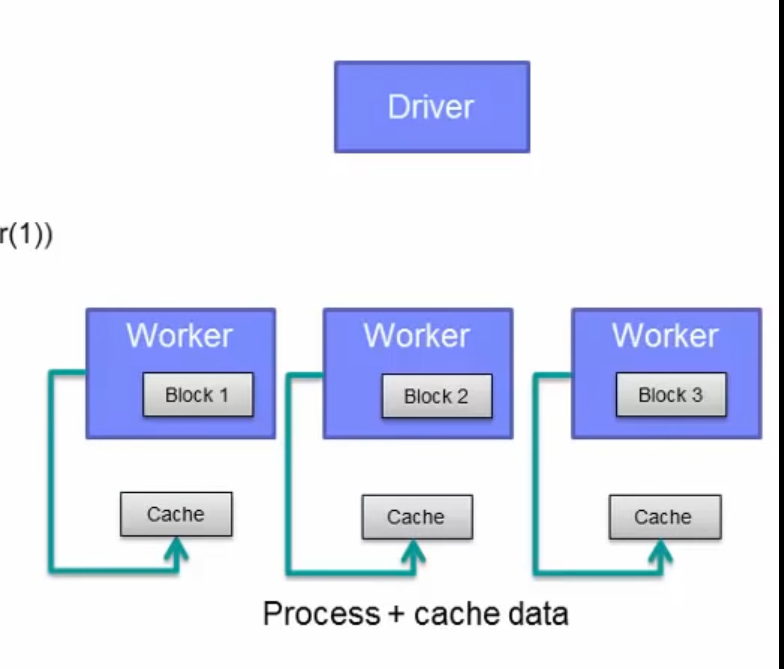
Then the driver sends the code to be executed on each block. In the example, it would be the various transformations and actions that will be sent out to the workers. Actually, it is the executor on each worker that is going to be performing the work on each block. You will see a bit more on executors in a later lesson.



#### Caching

messages.cache()

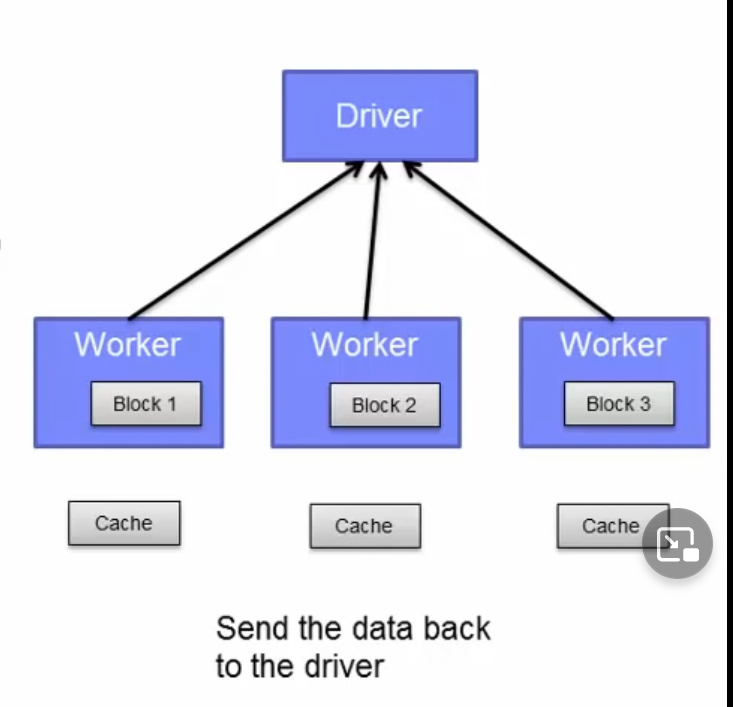
Then the executors read the HDFS blocks to prepare the data for the operations in parallel. After a series of transformations, you want to cache the results up until that point into memory. A cache is created.



#### Actions

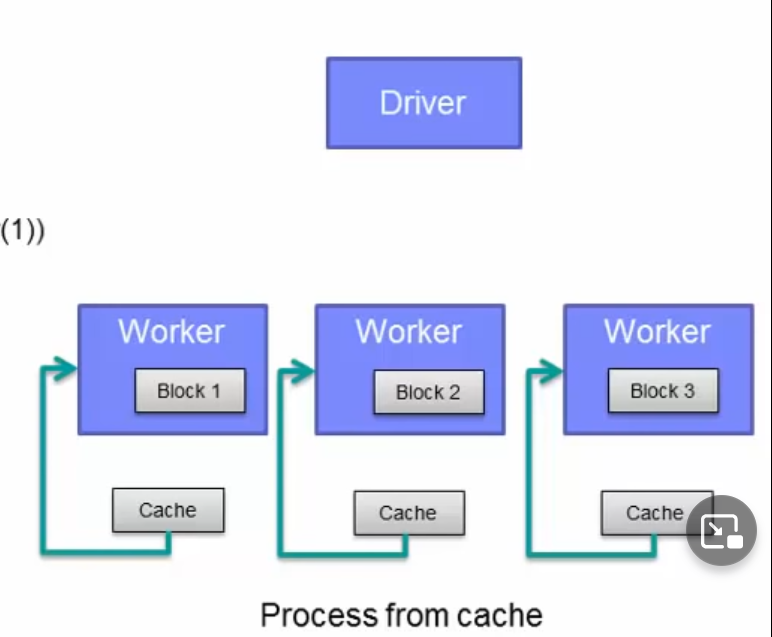
messages .filter(\_.contains(“mysql”)).count()

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.

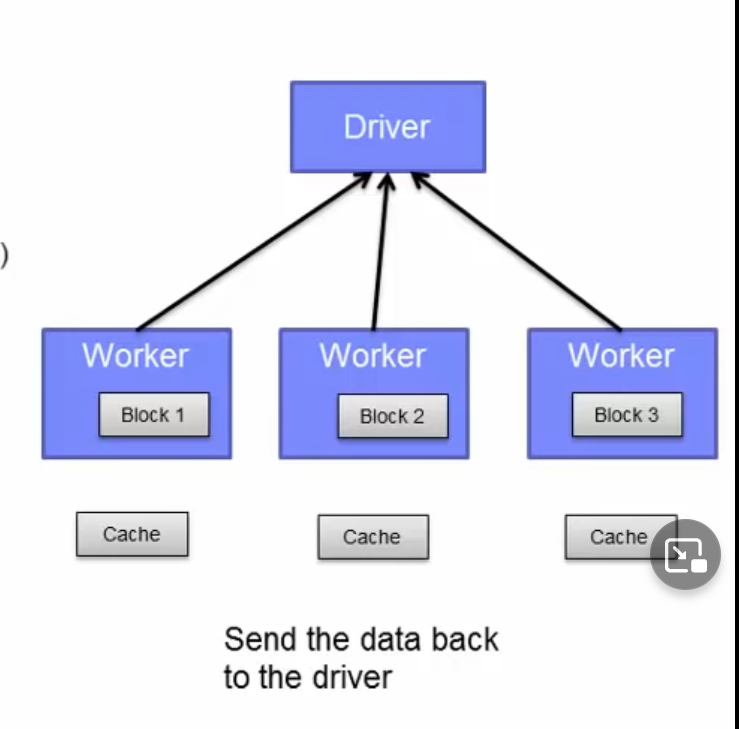


messages filter(\_.contains(“php”)).count()

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.



### 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

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

#### 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

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.

#### 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.

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.

#### 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

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.

#### 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

#### 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.

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.

#### count()

Return the number of elements in a dataset.

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

#### first()

Return the first element of the dataset

#### take(n)

Returns an array with the first n elements of the dataset.

#### foreach(func)

Run a function func on each element of the dataset.

## Utilize shared variables and key-value pairs

Q) Which of the following statements is true of RDD persistence? Select all that apply.

* Persistence through caching provides fault tolerance
* Future actions can be performed significantly faster
* Each partition is replicated on two cluster nodes
* RDD persistence always improves space efficiency
* By default, objects that are too big for memory are stored on the disk

### 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()
    - 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.
  + cache()
    - essentially just persist with MEMORY\_ONLY storage
    - Again the cache() method is just the default way of using persistence by storing deserialized objects in memory.

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 the default of the persist function with the MEMORY\_ONLY storage. One of the key capabilities 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.

#### MEMORY\_ONLY

Store as deserialized Java objects in the JVM. Ifthe 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

Like 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.

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 serialize 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

### What 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 executor’s crash. A lot of text on this page, but don't worry. It can be used as a reference when you must 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 use that. It is the fastest option to fully take advantage of Spark's design.

If not, you can serialize 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 executor’s 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.

### Shared variables and key-value pairs

* When a function is passed from the driver to a worker, normally a separate copy of the variables is used.

#### Broadcast variables

* Read-only copy on each machine
* Distribute broadcast variables using efficient broadcast algorithms

Normally, when a function is passed from the driver to a worker, a separate copy of the variables is 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.

#### 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.

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

#### Scala: key-value pairs:

val pair = (‘a’, ‘b’)  
pair.\_1 // will return ‘a’  
pair.\_2 // will return ‘b’

In Scala, you create a key-value pair RDD by typing val pair = ('a' , 'b'). To access each element, invoke the dot\_ 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.

#### Python: key-value pairs:

pair = (‘a’, ‘b’)  
pair[0] # will return ‘a’   
pair[1] # will return ‘b’

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.

#### Java: key-value pairs:

Tuple2 pair = new Tuple2(‘a’, ‘b’);  
pair. 1 // will return ‘a’  
pair. 2 // will return ‘b’

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.

### 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)
  + 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.
* Custom objects as key in key-value pair requires a custom equals() method with a matching hashCode() method.

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) if 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.

### Example:

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

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.

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 code it yourself, you may want split each of the functions up.

1. For example, do the flatMap operation and return that to a RDD.
2. Then from that RDD, do the map operation to create another RDD.
3. Then finally, from that last RDD, invoke the reduceByKey method.
4. That would yield multiple lines, but you would be able to test each of the transformation to see if it worked properly.

# Spark application programming

## Understand the purpose and usage of the SparkContext

* The main entry point for Spark functionality
* Represents the connection to a Spark cluster
* Create RDDs, accumulators, and broadcast variables on that cluster
* In the Spark shell, the SparkContext, sc, is automatically initialized for you to use
* In a Spark program, import some classes and implicit conversions into your program:

import org.apache.spark.SparkContext  
import org.apache.spark.SparkContext.\_  
import org.apache.spark.SparkConf

Q) What is SparkContext?

* A tool for linking to nodes
* A tool that provides fault tolerance
* The built-in shell for the Spark engine
* A programming language for applications
* An object that represents the connection to a 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.

## Initialize Spark with the various programming languages

### Scala

#### Link

* 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 aHDFS 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>

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.

#### Initialize

* 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)

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

### Python

#### Link

* 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

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.

#### Initialize

* 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 masterin 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)

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.

### Java

#### Link

* 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:
  + groupiId = 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

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. 4

#### Initialize

* 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 masterparameter > is a Spark, Mesos, or YARN cluster URL (or a special “local” string to run in local mode)
  + In production mode, do not hardcode masterin 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);

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.

## Describe and run some Spark examples

Q) Which of the following is a main component of a Spark application’s source code?

* Transformations and actions
* Import statements
* Business Logic
* SparkContext object
* All of the above

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.

### Passing functions to Spark.

Q) Which of the following methods can be used to pass functions to Spark? Select all that apply.

* Transformations and actions
* Passing by reference
* Static methods in a global singleton
* Import statements
* Anonymous function syntax
* Spark’s API relies on heavily passing functions in the driver program to run on the cluster
* Three methods

1. Anonymous function syntax
   1. (x: Int) => x + 1

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.

1. Static methods in a global singleton object
   1. object MyFunctions {
      1. def funcl (s: String): String = {...}
   2. }
   3. myRdd.map (MyFunctions. funcl)

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 must send out the singleton object.

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

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:

* 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)
  + }

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.

### 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

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.

#### Sample Scala application.

* 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(®))
    - 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()
  + }
* }

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.

### 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

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.

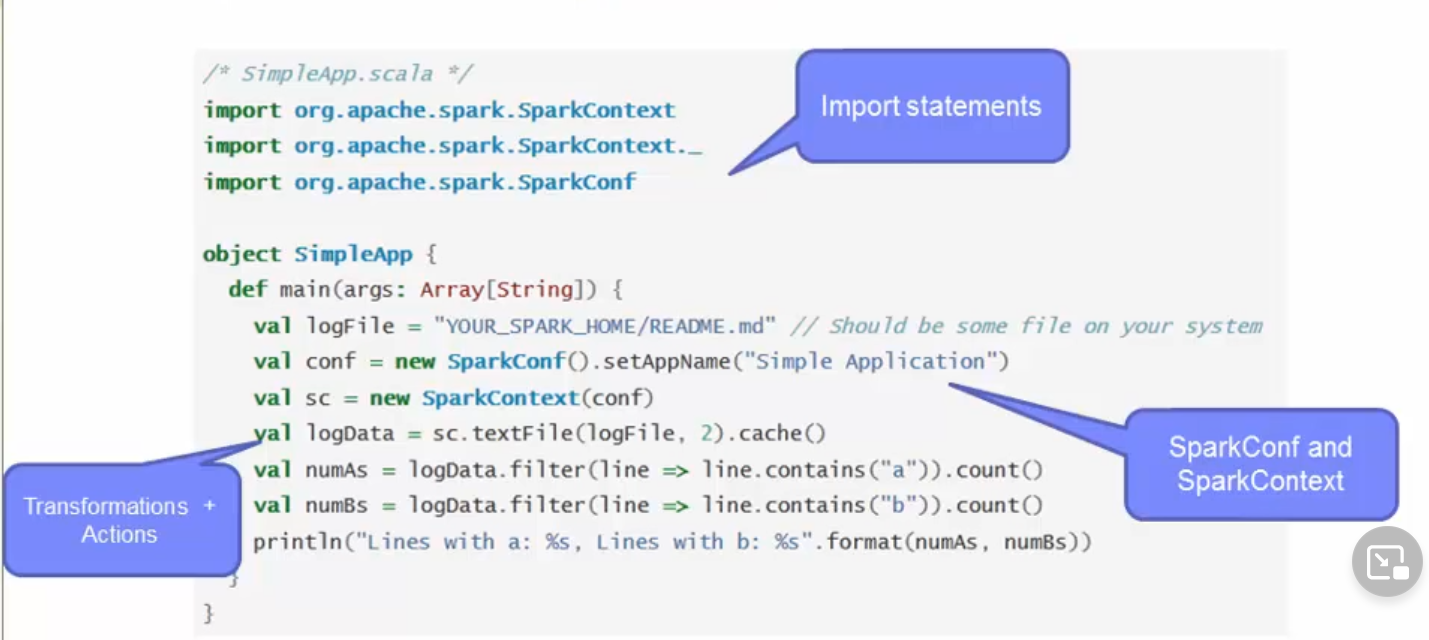
## Pass functions to Spark

## Create and run a Spark standalone application

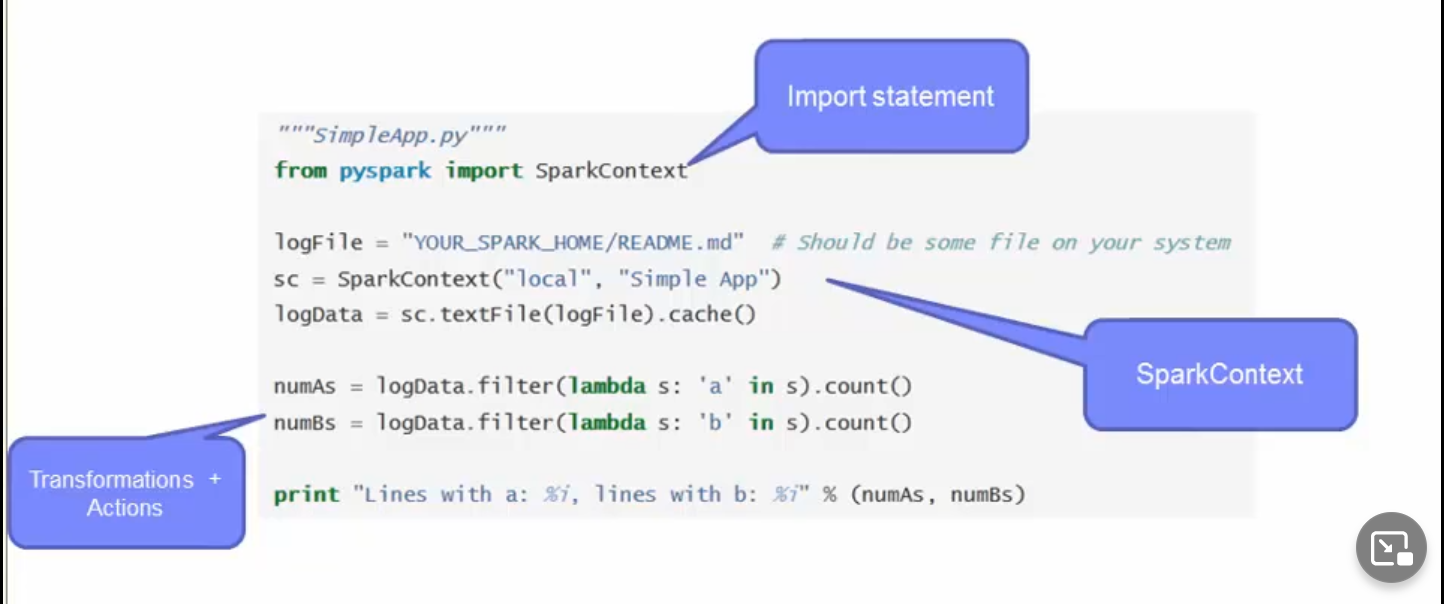
In the next three slides, you will see another example of creating a Spark application. First you will see how to do this in Scala. The next following sets of slides will show Python and Java. The application shown here counts the number of lines with 'a' and the number of lines with 'b'.

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.

1. First you must create a SparkConf to set up your application's name.
2. Then you create the SparkContext by passing in the SparkConf object.
3. 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.
4. 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.



## Submit applications to the cluster

* 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
  + ./simple.sbt
  + ./srce
  + ./src/main
  + ./src/main/scala
  + ./src/main/scala/SimpleApp.scala
* Java using Maven:
  + ./pom. xml
  + ./src
  + ./src/main
  + ./src/main/java
  + ./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

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.

* Package application into a JAR (Scala/Java) or set of .py or .zip (for Python)
* Use spark-submit under the $SPARK\_HOME/bin directory
  + ./bin/spark-submit \
  + --class <main-class> \
  + --master <master-url> \
  + --deploy-mode <deploy-mode> \
  + --conf <key>=<value> \
  + …# other options
  + <application-jar> \
  + [application-arguments]
* spark-submit --help will show you the other options
* Example of running an application locally on 8 cores:
  + ./bin/spark-submit \
  + --class org.apache.spark.examples.SparkPi \
  + --master local[8] \
  + /path/to/examples.jar \
  + 100

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.

# Introduction to Spark libraries

Q) Which of the following is NOT an example of a Spark library?

* MLlib
* GraphX
* Spark SQL
* Spark Streaming
* Hive

## Understand and use the various Spark libraries

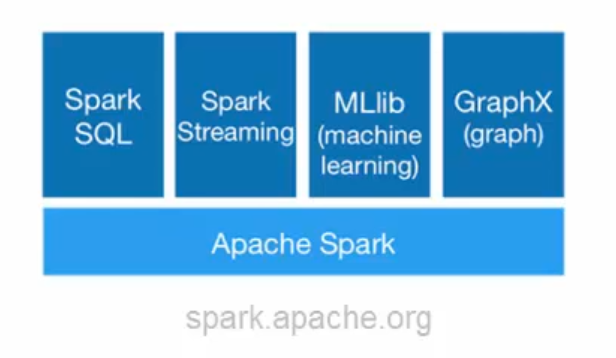
After completing this lesson, you should be able to understand and use the various Spark libraries including

* SparkSQL
* Spark Streaming
* MLlib
* 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.



### SparkSQL

* 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 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.

* 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:
  + 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

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 must 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.

#### SchemaRDD data sources

##### Inferring 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")
  + -mMap(\_.split(","))
  + -Map(p => Person(p(0), p(1).trim.toInt) )
* Register the RDD as a table
  + people.registerTempTable ("people")
* Run SQL statements using the sq/ 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)

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.

##### Programmatic interface

* Use when you cannot define the case classes ahead of time.
* Create the RDD:
  + val people = sc.textFile(...)

Three steps to create the SchemaRDD:

1. Create an RDD of Rows from the original RDD
   * val schemaString = “name age”
2. Create the schema represented by a StructType matching the structure of the Rows in the RDD from step 1.
   * val schema = StructType( schemaString.split(" ").map(fieldName =>
   * StructField(fieldName, StringType, true)))
3. Apply the schema to the RDD of Rows using the app/ySchema method.
   * val rowRDD = people.map(\_.split(",")).map(p => Row(p(0), p(1).trim))
   * val peopleSchemaRDD = sqlContext.applySchema(rowRDD, schema)

* Then register the peopleSchemaRDD as a table
  + peopleSchemaRDD. registerTempTable ("people")
* Run the sql statements using the sql method:
  + val results = sqlContext.sql("SELECT name FROM people")
  + results.map(t => "Name: " + t(0)).collect() .foreach(printin)

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

Q) From which of the following sources can Spark Streaming receive data? Select all that apply.

* Kafka
* JSON
* Parquet
* HDFS
* Hive

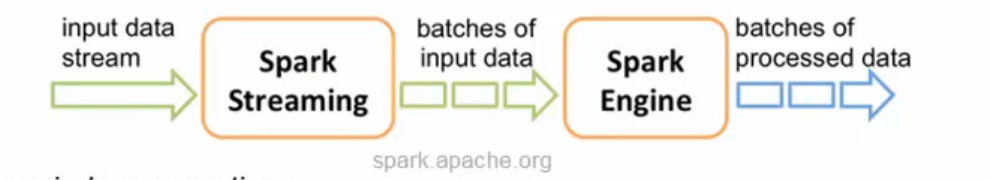
Q) In Spark Streaming, processing begins immediately when an element of the application is executed. True or false?

* 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

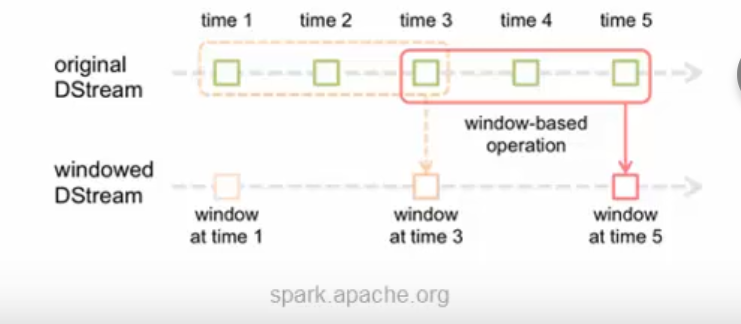
Spark streaming gives you the capability to process live streaming data in small batches. Utilizing Spark's core, Spark Streaming is scalable, high-throughput and fault-tolerant. You write Stream programs with DStreams, which is a sequence of RDDs from a stream of data. There are various data sources 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.

#### Internals

* Input stream (DStream goes into Spark streaming)
* Breaks up into batches
* Feeds into the spark engine for processing
* Generate the results in streams of batches



* Sliding window operations
  + Windowed omputations
    - Window length
    - Sliding interval
    - reducebKKeyAndWindow



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 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.

#### Example

* 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 ()

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 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.

* 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

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.

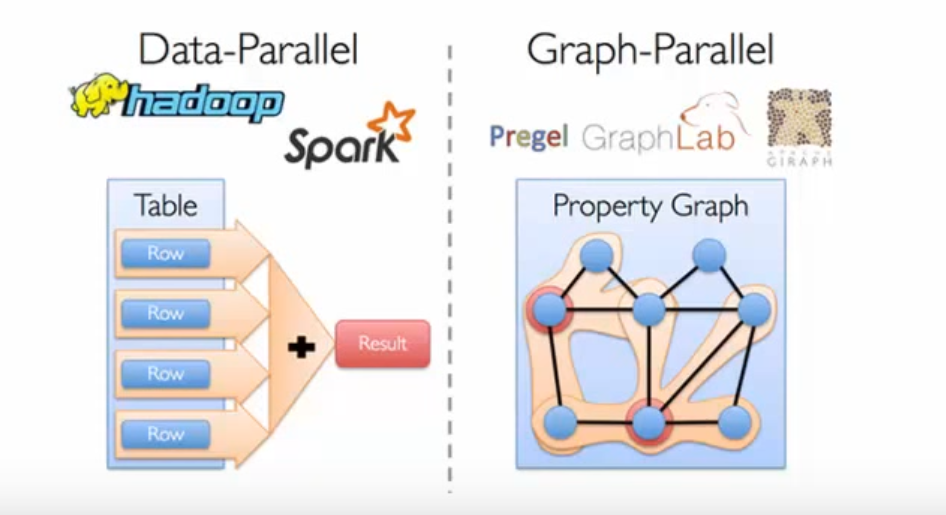
### MLib

* MLlib for machine learning library — under active development
* Common algorithm and utilities
  + Classification
  + Regression
  + Clustering
  + Collaborative filtering
  + Dimensionality reduction
* Lab exercise on applying the clustering K-Means algorithm on drop off points of taxis

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.

### GraphX

* GraphX for graph processing
  + Graphs and graph parallel computation
  + Social networks and language modeling
* Lab exercise will be on finding attributes associated with the tops users.



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.

# Spark configuration, monitoring, and tuning

Q) Which of the following is a main component of a Spark cluster? Select all that apply.

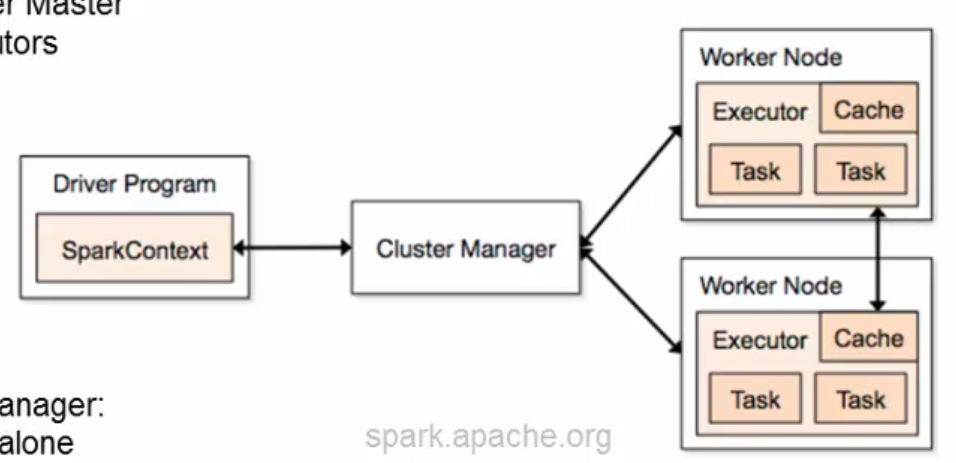
* Driver Program
* SparkContext
* Cluster Manager
* Worker node
* Cache

Q) What are the main locations for Spark configuration? Select all that apply.

* The SparkConf object
* The Spark Shell
* Executor Processes
* Environment variables
* Logging properties

## Understand components of the Spark cluster

There are three main components of a Spark cluster.



### Driver

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 will need to externalize the data if you wish to share data between the different applications. Spark applications don't care about the underlying cluster manager. If 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.

### Cluster Managers

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.

## Configure Spark to modify the Spark properties, environmental variables, or logging properties

* 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
* Set application properties via the SparkConf object.
  + val conf = new SparkConf ()
    - .setMaster ("local")
    - . setAppName ("CountingSheep”)
    - .set ("spark.executor.memory", "lg")
  + val sc = new SparkContext (conf)
* Dynamically setting Spark properties
  + Create a SparkContext with an empty conf
    - val sc = new SparkContext(new SparkConf ())
  + Supply the configuration values during runtime
    - ./bin/spark-submit --name "My app" --master local[4] --conf
    - spark.shuffle.spill=false --conf "spark.executor.extraJavaOptions=-
    - XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp. jar
  + conf/spark-defaults.conf
* Application web Ul
  + http://<driver>: 4040

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.

## Monitor Spark using the web UIs, metrics, and external instrumentation

Three ways to monitor Spark applications

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

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.

Web UI

* 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

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.

## Understand performance tuning considerations

Q) Which of the following techniques can improve Spark performance? Select all that apply.

* Scheduler Configuration
* Memory Tuning
* Data Serialization
* Using Broadcast variables
* Using nested structures

Data serialization

* + Java serialization
  + Kyro serialization
    - conf.set ("spark.serializer", "org.apache.spark.serializer.KryoSerializer”)

Memory tuning

* Amount of memory used by the objects
  + Avoid Java features that add overhead
  + Go with arrays or primitive types
  + Avoid nested structures when possible
* Cost of accessing those objects
  + Serialized RDD storage
* Overhead of garbage collection
  + Analyze the garbage collection
  + SPARK\_JAVA\_OPTS
  + verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps to your SPARK\_JAVA\_OPTS

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.

* Level of parallelism
  + Automatically set according to the file size
  + Optional parameters such as SparkContext.textFile
  + spark.default.parallelism
  + 2-3 tasks per CPU core in the cluster
* Memory usage of reduce tasks
  + OutOfMemoryError can be resolved by increasing the level of parallelisn
* Broadcasting large variables
* Serialized size of each tasks are located on the master.
  + Tasks > 20 KB worth optimizing

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