SPARK FUNDAMENTALS I

COURSE SYLLABUS

1. Introduction to Spark – Getting started
2. Resilient Distributed Dataset and DataFrames
3. Spark Application programming
4. Introduction to Spark libraries
5. Spark configuration, monitoring and tuning

BIG DATA

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.

DESCRIPTION

Module 1 - Introduction to Spark - Getting started

- What is Spark and what is its purpose?
- Components of the Spark unified stack
- Resilient Distributed Dataset (RDD)
- Downloading and installing Spark standalone
- Scala and Python overview
- Launching and using Spark's Scala and Python shell ©

Module 2 - Resilient Distributed Dataset and DataFrames

- Understand how to create parallelized collections and external datasets
- Work with Resilient Distributed Dataset (RDD) operations
- Utilize shared variables and key-value pairs

Module 3 - Spark application programming

- Understand the purpose and usage of the SparkContext
- Initialize Spark with the various programming languages
- Describe and run some Spark examples
- Pass functions to Spark
- Create and run a Spark standalone application
- Submit applications to the cluster

Module 4 - Introduction to Spark libraries

- Understand and use the various Spark libraries

Module 5 - Spark configuration, monitoring and tuning

- Understand components of the Spark cluster
- Configure Spark to modify the Spark properties, environmental variables, or logging properties
- Monitor Spark using the web UIs, metrics, and external instrumentation
- Understand performance tuning considerations

LEARNING OBJECTIVES

- the purpose of Spark and understand why and when you would use Spark.
- how to list and describe the components of the Spark unified stack.
- the basics of the Resilient Distributed Dataset, Spark's primary data abstraction.
- how to download and install Spark standalone.
- an overview of Scala and Python.

MODULE 1 – INTRODUCTION TO SPARK

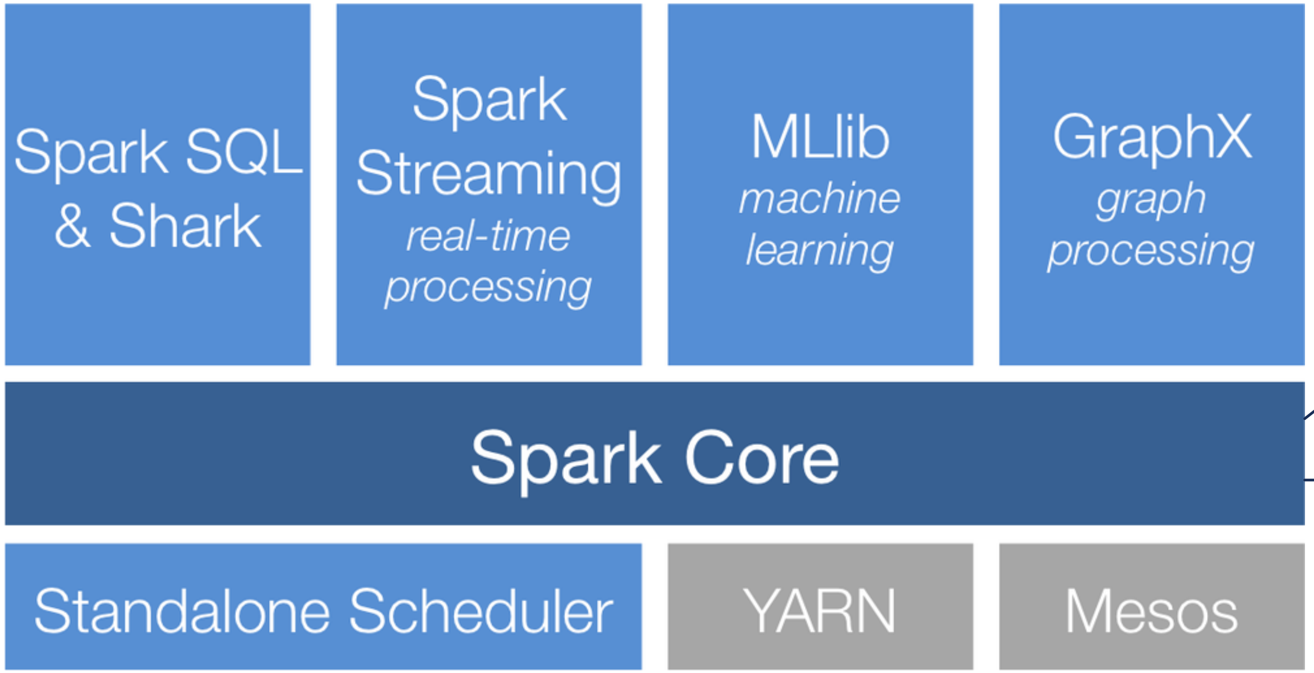
In this lesson:

- Learn about the purpose of Spark
- Learn about the components of the Spark unified stack
- Understand the basics of Resilient Distributed Dataset (RDD)
- Learn how to download and install Spark standalone
- Be given a brief overview of Scala and Python
- Learn how to launch and use Spark's Scala and Python shells

INTRODUCTION TO SPARK – PART I

- The ease of use with Spark enables you to quickly pick it up using simple APIs for Scala, Python and Java.
- 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.
- Why to use spark and it for? Spark provides parallel distributed processing, fault tolerance on commodity hardware, scalability, etc. Spark adds the concept with aggressively cached in-memory distributed computing, low latency, high level APIs and stack of high level tools. This save time and money.

INTRODUCTION TO SPARK – PART 2

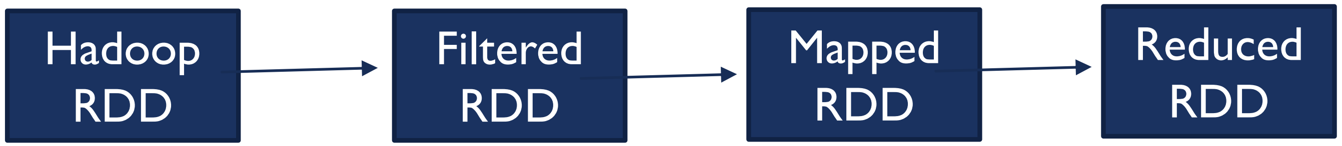


The Spark core is designed to scale up from one to thousands of nodes.

Spark's primary core abstraction is called RDD, it is just a distributed collection of elements that is parallelized across the Cluster. There are two types of RDD:

- Transformations
- Actions

Example of RDD flow:



LESSON SUMMARY

- Explain the purpose of Spark
- List and describe the components of the Spark unified stack
- Understand the basics of RDDs
- Downloading and installing Spark standalone
- Scala and Python overview
- Launch and use Spark's Scala and Python Shell

MODULE 2 – RESILIENT DISTRIBUTED DATASET AND DATAFRAMES

- **In this lesson:**
- Describe Spark's primary data abstraction
- Understand how to create parallelized collections and external datasets
- Work with Resilient Distributed Dataset (RDD) operations and DataFrames
- Utilize shared variables and key-value pairs

RDD - PART I

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

Creating an RDD

- Launch the Spark shell
  ./bin/spark-shell
- Create some data
  val data = 1 to 10000
- Parallelize that data (creating the RDD)
  val distData = sc.parallelize(data)

pyspark

- Perform additional transformations or invoke an action on it.
  distData.filter(…)
- Alternatively, create an RDD from an external dataset
  - val readmeFile = sc.textFile("Readme.md")

RDD - PART I

RDD operations - Basics

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

- MapReduce example:
```
val wordCounts = textFile.flatMap(line => line.split (" "))
.map(word => (word, 1))
.reduceByKey((a,b) => a + b)

wordCounts.collect()
```

RDD - PART 2

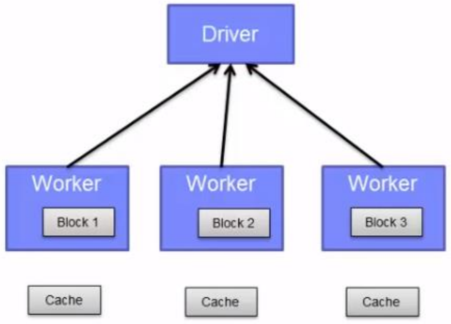
Direct Acyclic Graph (DAG)

- View the DAG
  linesLength.toDebugString
- Sample DAG

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

What happens when an action is executed?

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



Send the data back
to the driver

RDD operations - Transformations

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

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

- Actions returns values

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

RDD - PART 3

RDD persistence

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

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

Spark website

Which storage level to choose?

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

RDD - PART 3

Shared variables and key-value pairs

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

Programming with key-value pairs

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

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

Scala: key-value pairs

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

Python: key-value pairs

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

Java: key-value pairs

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

LESSON SUMMARY

- Describe Spark`s primary data abstraction
- Understand how to create parallelized collections and external datasets
- Work with RDD operations
- Utilize shared variables and key-value pairs

MODULE 3 – SPARK APPLICATION PROGRAMMING

In this lesson:

- Understand the purpose and usage of the SparkContext
- Initialize Spark with the various programming languages
- Describe and run some Spark examples
- Pass functions to Spark
- Create and run a Spark standalone application
- Submit applications to the cluster

APPLICATION PROGRAMMING – PART I

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 (like pyspark!) - (Import some Spark classes: from pyspark impor SparkContext, SparkConf)

when typing pyspark in the terminal it automatically creates a sparkContext!

Initializing Spark - Python

- Build a SparkConf object that contains information about your application

```
conf = SparkConf().setAppName(appName).setMaster(master)
```

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

```
sc = SparkContext(conf=conf)
```

APPLICATION PROGRAMMING – PART 2

Passing functions to Spark

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

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

  - Static methods in a global singleton object

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

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

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

    - Avoid:

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

    - Consider:

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

Programming the business logic:

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

Running Spark examples:
In Py: ./bin/spark-submit examples/src/main/python/pi.py

APPLICATION PROGRAMMING – PART 3

Create Spark standalone applications – Python

```python
"""SimpleApp.py"""
from pyspark import SparkContext

logFile = "YOUR_SPARK_HOME/README.md"  # Should be some file on your system
sc = SparkContext("local", "Simple App")
logData = sc.textFile(logFile).cache()

numAs = logData.filter(lambda s: 'a' in s).count()
numBs = logData.filter(lambda s: 'b' in s).count()

print "Lines with a: %i, lines with b: %i" % (numAs, numBs)
```

Import statement

SparkContext

Transformations + Actions

Run standalone applications

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

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

Submit applications to the cluster

- 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
```

LESSON SUMMARY

- Understand the purpose and usage of the SparkContext
- Link with Spark using Scala, Python, and Java
- Initialize Spark using Scala, Python, and Java
- Describe and run some Spark examples
- Pass functions to Spark
- Create and run a Spark standalone application
- Submit applications to the cluster

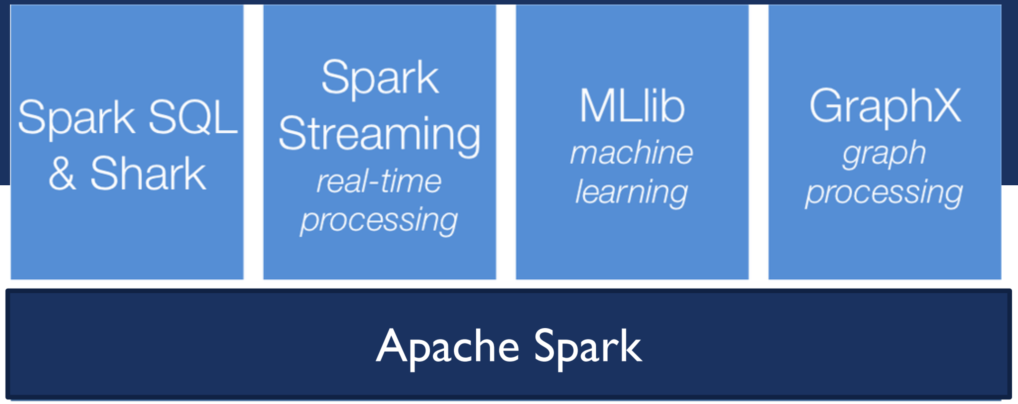
MODULE 4 – INTRODUCTION TO THE SPARK LIBRARIES

In this lesson:

- Understand and use the various Spark libraries

SPARK LIBRARIES – PART I

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



Spark SQL

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

Spark SQL – Getting started

- SQLContext
  - Created from a SparkContext

  Scala:

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

  Java:

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

  Python:

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

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

SPARK LIBRARIES – PART 2

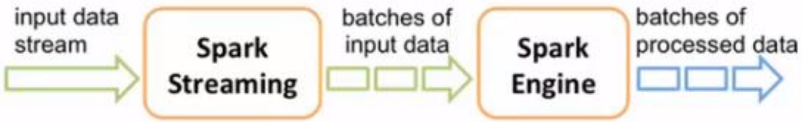
Spark Streaming

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

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

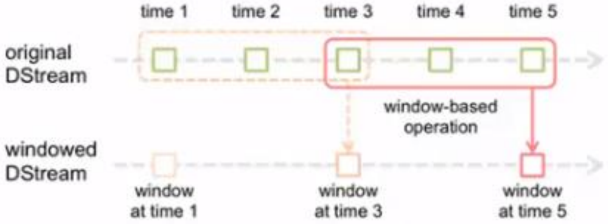
Spark Streaming - Internals

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



spark.apache.org

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



Spark Streaming – Getting started

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

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

- Create the StreamingContext object

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

- Create a DStream

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

- Split the lines into words

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

- Count the words

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

- Print to the console:

```
wordCounts.print()
```

Spark Streaming – Continued

- No real processing happens until you tell it:

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

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

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

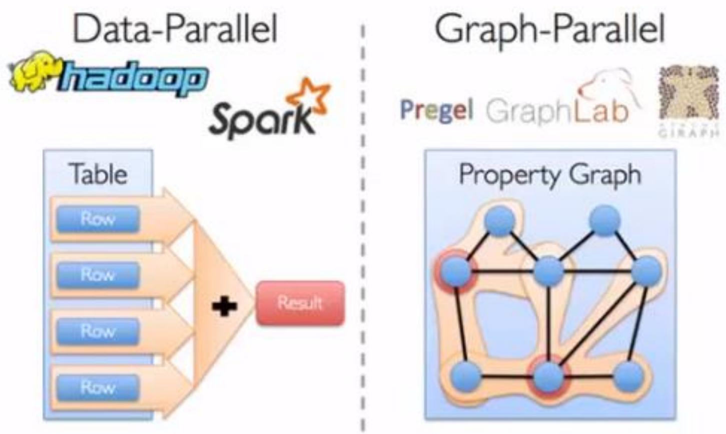
SPARK LIBRARIES – PART 3

MLlib

- 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

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.



MODULE 5 – SPARK CONFIGURATION, MONITORING AND TUNING

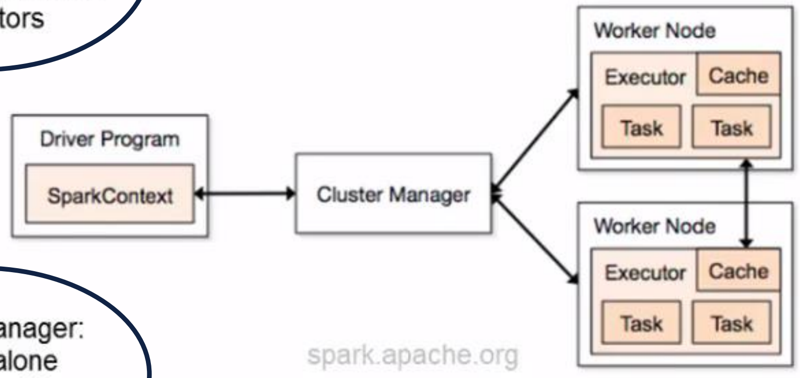
In this lesson:

- Understand components of the Spark cluster
- Configure Spark to modify the Spark properties, environmental variables, or logging properties
- Monitor Spark using the web UIs, metrics, and external instrumentation
- Understand performance tuning considerations

CONFIGURATION, MONITORING AND TUNING - PART I

Spark cluster overview

- Components
  - Driver
  - Cluster Master
  - Executors



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

Spark configuration

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

CONFIGURATION, MONITORING AND TUNING - PART I

Spark configuration – Spark properties

- Set application properties via the SparkConf object.

```
val conf = new SparkConf()
       .setMaster("local")
       .setAppName("CountingSheep")
       .set("spark.executor.memory", "1g")
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 UI

```
http://<driver>:4040
```

CONFIGURATION, MONITORING AND TUNING - PART 2

Spark monitoring

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

Spark monitoring – Web UI / history server

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

CONFIGURATION, MONITORING AND TUNING - PART 2

Spark tuning

- 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
```

Spark tuning – other considerations

- 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 parallelism
- Broadcasting large variables
- Serialized size of each tasks are located on the master.
  - Tasks > 20 KB worth optimizing

THANKS