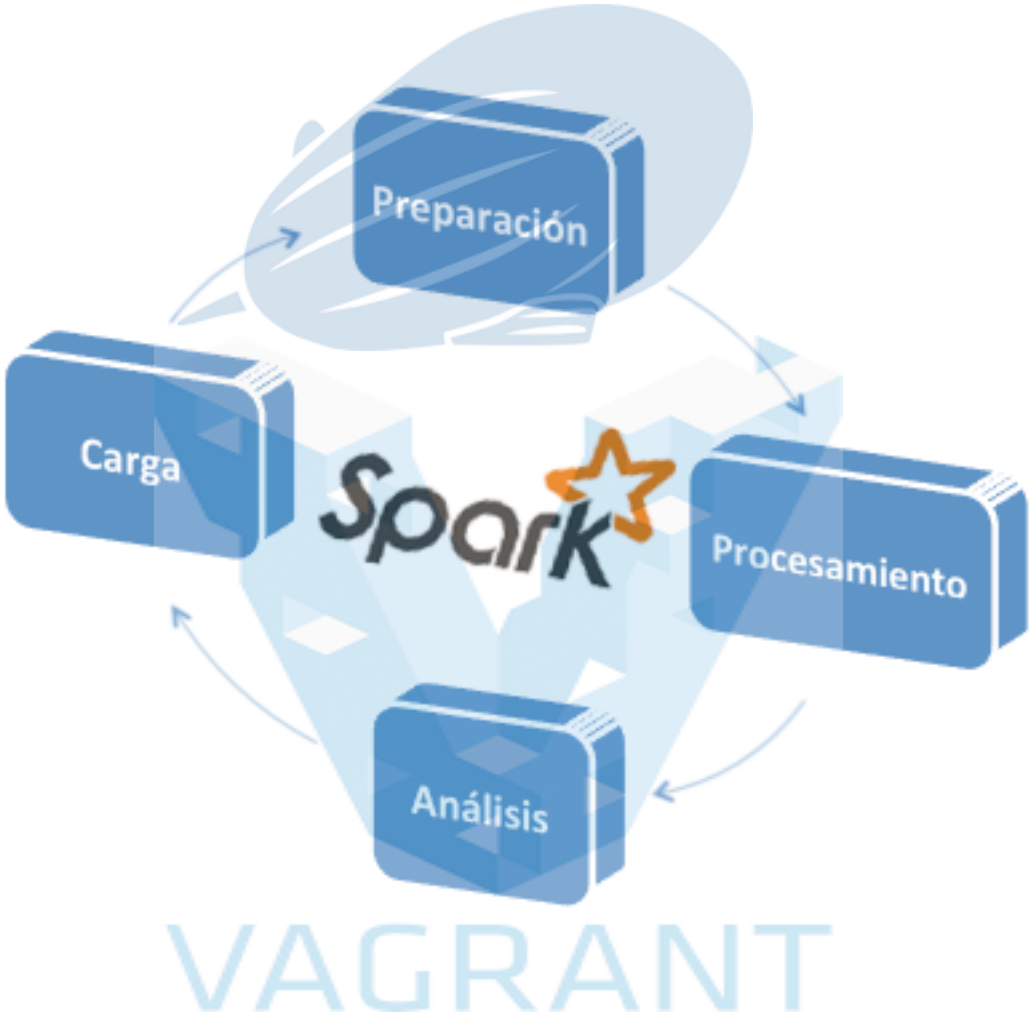
Introducción a Spark



Fco. Javier Lahoz Sevilla

Introducción a Spark

- **Resumen**: Introducción a Spark, Vagrant y Zeppelin viendo con ejemplos prácticos los distintos componentes del Framework de Spark con Scala, Python y R.

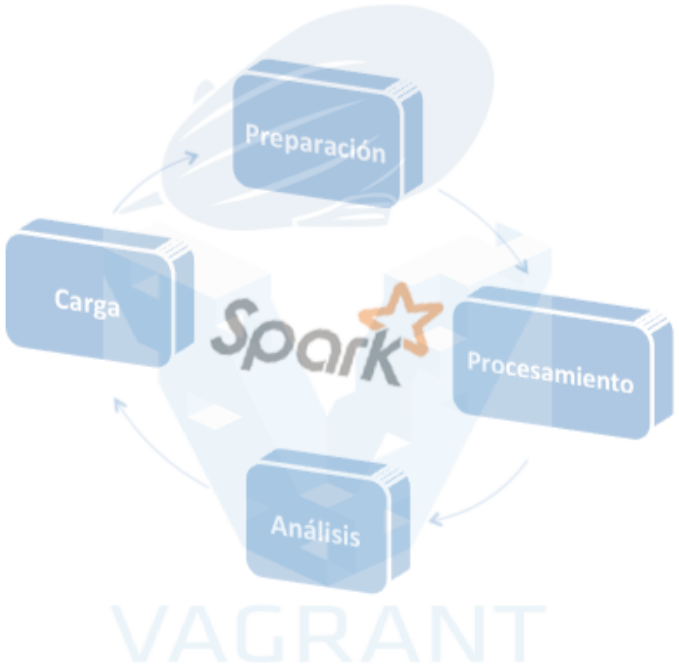
- **Nivel**: Intermedio.

- **Audiencia**:
  - Developers
  - Data Engineers
  - Data Sciences

- **Requisitos**: Conocimientos básicos de Linux y Hadoop

- **Recursos**:
  - Oracle Virtual Box
  - Vagrant
  - Emulador Linux (MobaXterm, Putty)
  - **SSOO que permita virtualización de 64 bits**

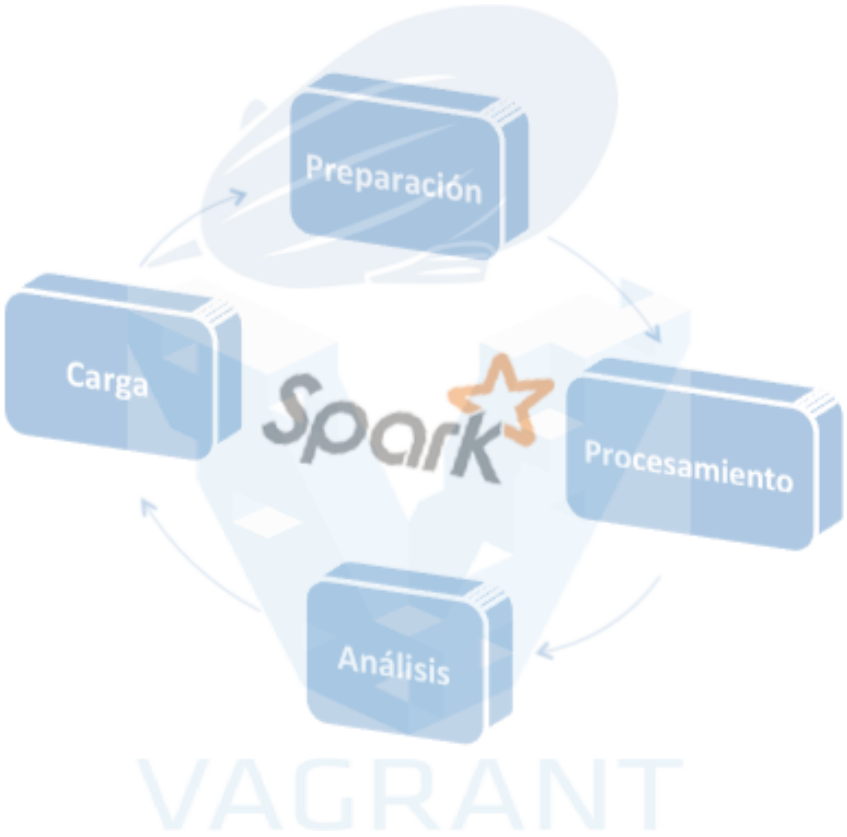
- **Objetivos del curso:** Introducirse en el mundo de Spark y de los sistemas virtualizados.



Fco. Javier Lahoz Sevilla

Introducción a Spark

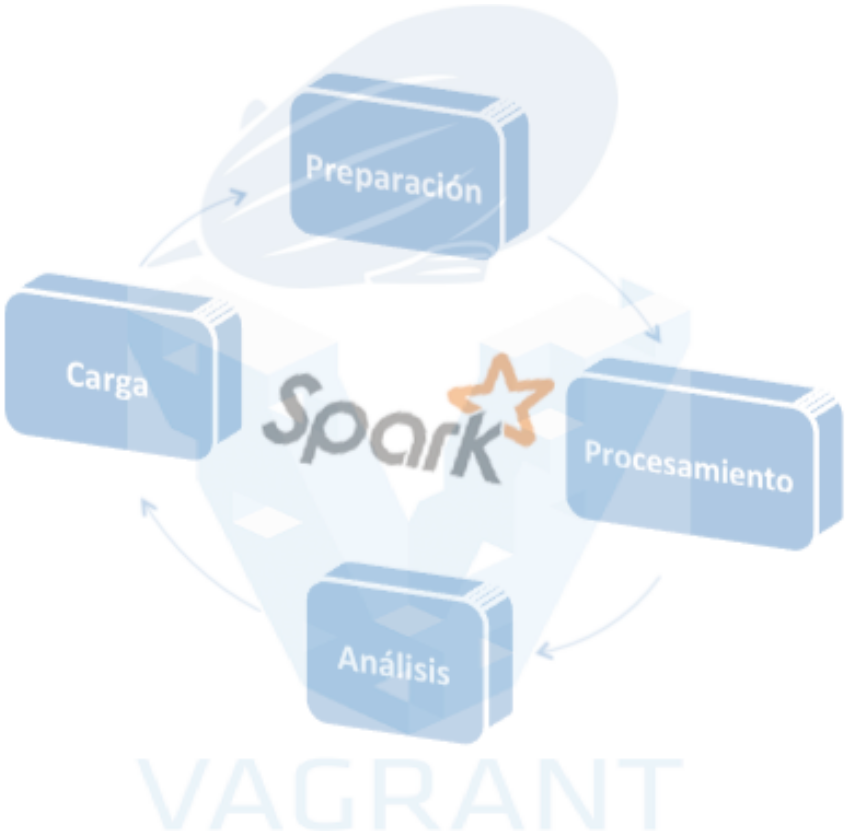
- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Introdución a Vagrant

- Vagrant nos permite administrar máquinas virtuales de una forma simple y sencilla.
- Las máquinas virtuales pueden ejecutarse en distintos sistemas operativos.
- Vagrant se puede utilizar con varios proveedores:
  - VirtualBox
  - VMWare
  - AWS
- Nos permite usar box ya empaquetados o crear nuestros propios entornos.

Introdución a Vagrant

- Vagrant para desarrolladores:
  - Permite aislar dependencias y configurar entornos.
  - Podemos desplegar entornos de una forma rápida reduciendo tiempos.
  - Existen muchos box previamente empaquetados (Ubuntu, Centos, Debian).

- Vagrant para ingenieros de explotación:
  - Da entornos desechables para el testeo de despliegues de infraestrucutas.
  - Despliegue mediante shell script de entornos con diversos proveedores.
  - Integrable con soluciones como Chef o Puppet.

Introdución a Vagrant

Comenzaremos descargando el box y la configuración de Vagrant:

- https://s3-eu-west-1.amazonaws.com/curso-spark/curso-spark.box
- https://s3-eu-west-1.amazonaws.com/curso-spark/sha1_file
- https://s3-eu-west-1.amazonaws.com/curso-spark/Vagrantfile

Una vez descargado ya podemos arrancarlo:

- **vagrant box add** sparkbox /path/to/curso-spark.box
- **vagrant init** sparkbox
- Sustituimos el fichero Vagrantfile con el proporcionado
- Arrancamos la máquina virtual: **vagrant up**
- Nos conectamos a la máquina: **vagrant ssh**
- También podemos pararla o destruirla: **vagrant halt/destroy**

En el fichero README del HOME de la máquina virtual se detalla el sistema operativo y sw desplegado además de los fuentes y dataset que se utilizarán a lo largo del curso.

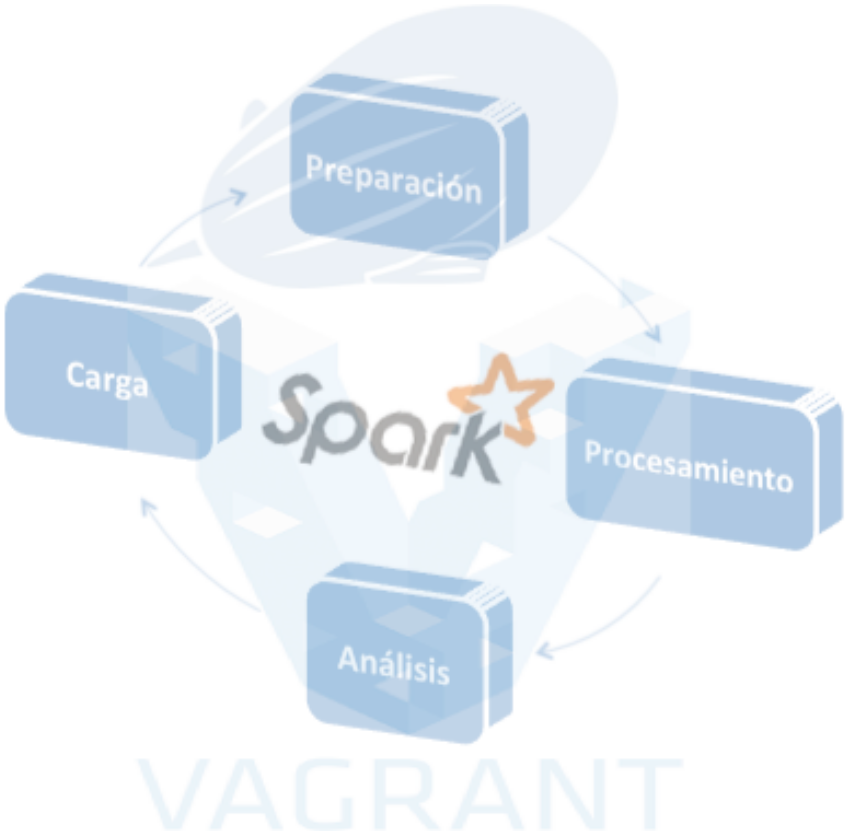
Podemos empezar arrancando Hadoop:

>$HADOOP_HOME/sbin/start-dfs.sh

>$HADOOP_HOME/sbin/start-yarn.sh

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Introducción a Spark

- **Spark** es un framework de computación distribuida en memoría desarrollado por Universidad de California en su AMPLab para, más tarde, ser donado la función de Apache. Spark requiere un cluster y un sistema de almacenamiento distribuido.

- Los clusters que Spark soportan son:
  - Standalone (nativo en Spark Cluster)
  - Hadoop **YARN**
  - Apache Mesos

- Para el almacenamiento distribuido incluye:
  - **HDFS**
  - S3
  - Cassandra

Vamos a ver brevemente las principales componentes:

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib
- GraphX

Es el fundamento de todo el framework. Proporciona tareas distribuidas basadas en RDDs que son colecciones lógicas de datos particionadas a lo largo del cluster que se puede implementar en Scala, Java, Python y R.

Vamos a ver brevemente las principales componentes:

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib
- GraphX

Es un componente por encima de Spark Core que introduce una nueva abstracción llamada DataFrames que proporciona soporte para datos estructurados y semi estructurados en lenguaje SQL.

Vamos a ver brevemente las principales componentes:

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib
- GraphX

Aprovecha las capacidades del core de Spark para el procesamiento analítico en streaming mediante micro-batching que permite operar en lotes de datos. Este diseño permite reutilizar los procesos batch en streaming.

Vamos a ver brevemente las principales componentes:

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib
- GraphX

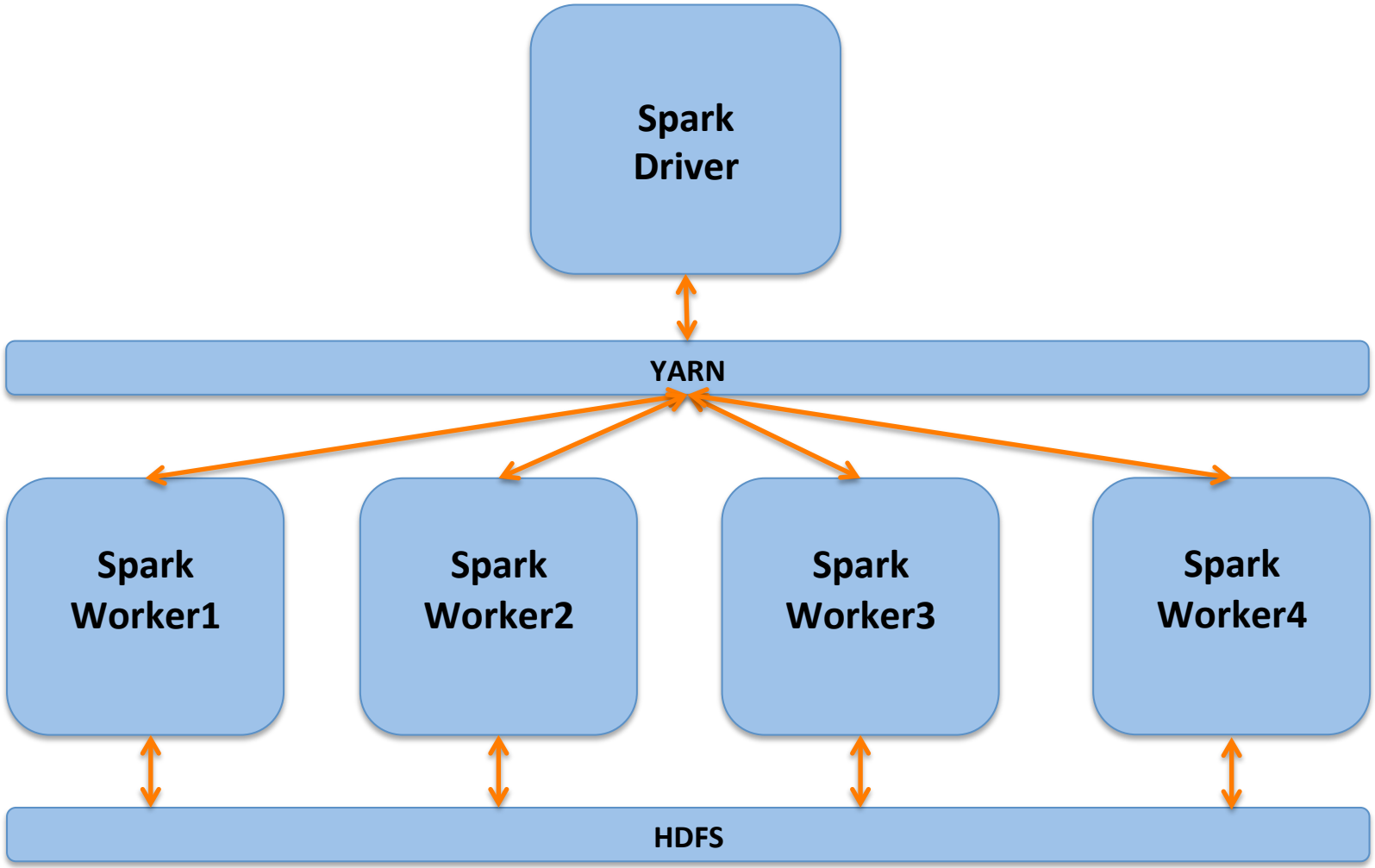
Framework para le ejecución de algoritmos de Machine Learning de forma distribuida por encima de Spark Core. Incluye algoritmos de clasificación, clusters, redución de dimensiones, entre otros.

Vamos a ver brevemente las principales componentes:

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib
- GraphX

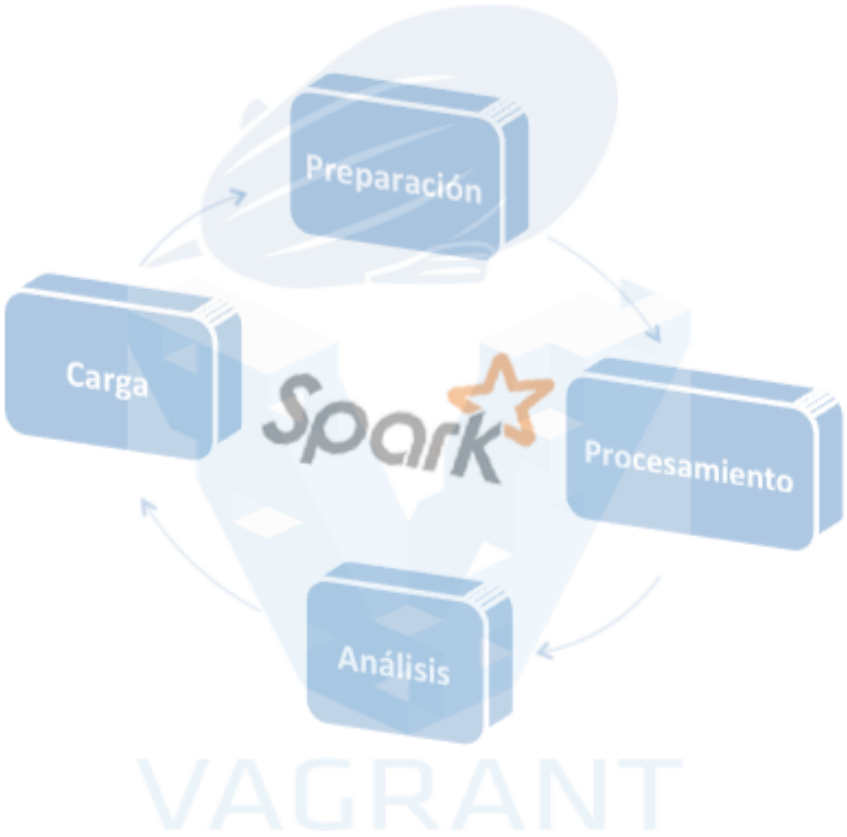
Framework para el prcesamiento distribuido de Grafos. Proprociona un API para expresar computación de grafos a través de un optimizado runtime.

Introducción a Spark



Introducción a Spark

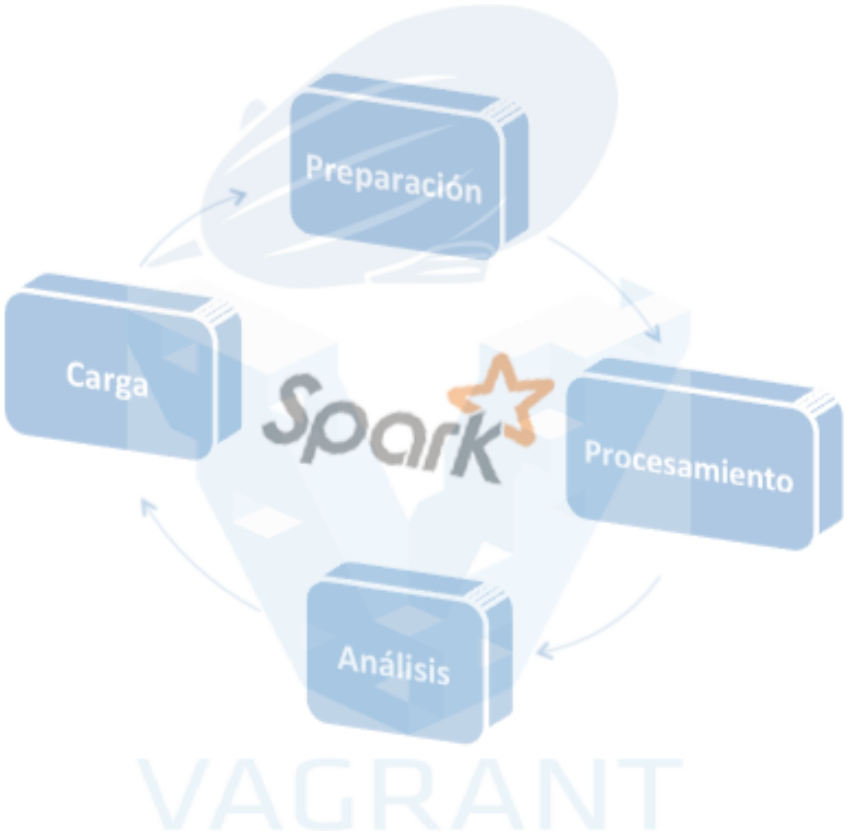
- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Spark Core

Todas las aplicaciones de Spark consiten en una aplicación que ejecuta una función principal y varias operaciones encadenadas.

La abstraccion principal de Spark proporciona un RDD (**Resilent Distributed Dataset**) que es una colección de elementos particionados a través de los nodos de nuestro cluster para operar en paralelo.

Los RDDs son creados en cada ejecución mediante la lectura de ficheros de HDFS o de otro sistema de archivos compatibles, son transformables en otros RDDs, pueden persistir en memoría bajo demanda para ejecutar operaciones de forma más eficiente y persite a errores.

Spark Core

Lo primero que una aplicación de Spark tiene que hacer es que crear el objeto **SparkContext** el cual indica a Spark como acceder al cluster.

En la creación del SparkContext se puede indicar la configuración de la aplicación mediante el objeto **SparkConf**:

- Nombre de la aplicación
- Maestro
- Librerias

Spark dispone de varias consolas para interpetrar las operaciones:

- **spark-shell**, consola para operar con scala
- **pyspark**, consola python
- **sparkR**, consola de R

Una vez desarrolladas las aplicaciones se ejecutarán a través de **spark-submit** con las siguientes opciones:

- --class <main class>
- --master:
  - local/[K]/[*]
  - yarn-client, modo cluster en cliente
  - yarn-cluster, modo cluster
- <application-jar>
- [application-arguments]
- --num-executors
- --driver-memory
- --executor-memory

Operaciones que se pueden realizar en los RDDs:

- Transformaciones:
  - **filter**(func), devuelve un dataset tras aplicar un filtro con la función, func.
  - **map**(func), devuelve un dataset parseando cada elemento mediante la función, func, sobre el dataset original.
  - **flatMap**(func), similar a map pero cada item puede mapear con 0 o más items de salida y devolverá un dataset en una secuencia de items únicos.
  - **groupByKey**([numTasks)], para una entrada (K,V) devuelve un dataset de pares (K,Iterable(V)).
  - **reduceByKey**(func,[numTasks]), para una entrada (K,V) devuelve un dataset (K,V) donde los valores para cada clave son agregados usando la reduce función, func.
  - **sortByKey**([ascending],[numTasks]), para una entrada (K,V) devuleve un dataset (K,V) ordenado por la clave K.

- Acciones:
  - **reduce**(func), agrega los elementos aplicando la función, func.
  - **collect**(), devuelve todos los elementos del dataset en un array.
  - **count**(), devuelve el número de elementos en el dataset.
  - **first**(), devuelve el primer elemento del dataset.
  - **take**(n), develve n elementos del dataset.
  - **saveAsTextFile**(path), escribe el dataset en el fichero de texto indicado en el path.
  - **foreach**(func), ejecuta la función, func, para cada elemento del dataset.

Se pueden compartir variables entre las máquinas del cluster:

- Broadcast. Estas variables permiten al programador mantener variables read-only cacheadas en cada máquina para que sean compartidas.
- Acumuladores. Variables para añadir operaciones asociativas que son soportadas de forma eficiente en paralelo. Por ejemplo, para implementar contadores o sumas.

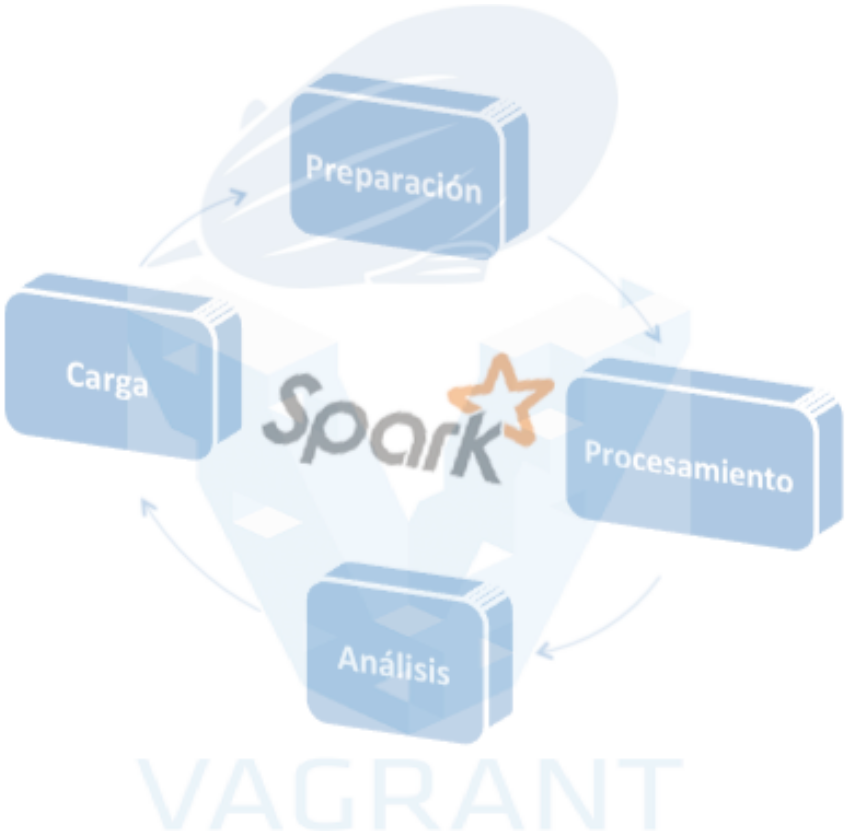
Ejemplo con scala:

```
>spark-shell
val textFile = sc.textFile("data/core/wordscount")
val wordCounts = textFile.flatMap(line => line.split(" "))
val filterWords = wordCounts.filter(word => word.length()>0)
val mapWords = filterWords.map(word => (word, 1))
val countWords = mapWords.reduceByKey((a, b) => a + b)
val countWordsSort = countWords.sortByKey()
countWordsSort.saveAsTextFile("output-wc")
```

```
val textFile = sc.textFile("data/core/wordscount").flatMap(line =>
line.split(" ")).filter(word => word.length()>0).map(word => (word,
1)).reduceByKey((a, b) => a + b).sortByKey().saveAsTextFile("output-
wc-1")
```

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Spark SQL

Spark SQL es un modulo de Spark para procesamiento de datos estructurados. Proporciona una abstracción de programación denominada DataFrames para distribución de sentencias SQL.

Un **DataFrame** es una colección distribuida de datos organizados en columnas. Conceptualmente equivale a una tabla en una base de datos relacional o a un data frame de R.

Operaciones con DataFrames:

- printSchema
- select
- filter
- groupBy
- join
- agg, max, min
- show
- describe
- collect
- count
- first
- take

Spark SQL

Ejemplo con python:

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

df = sqlContext.read.json("data/sql/tweets")
df.show()
df.printSchema()

df.select("text").show()
df.select("text").take(10)
df.select(df['user.screen_name'],
df['user.followers_count']).sort("user.followers_count").show()
df.select("retweeted_status.user.screen_name”)
df.groupBy("retweeted_status.user.screen_name").count().show()

df.registerTempTable("tweets")
sqlContext.sql("SELECT user.screen_name, user.followers_count FROM tweets ORDER
BY user.followers_count").show()
sqlContext.sql("SELECT retweeted_status.user.screen_name, COUNT(*) as total FROM
tweets  GROUP BY retweeted_status.user.screen_name").show()
```

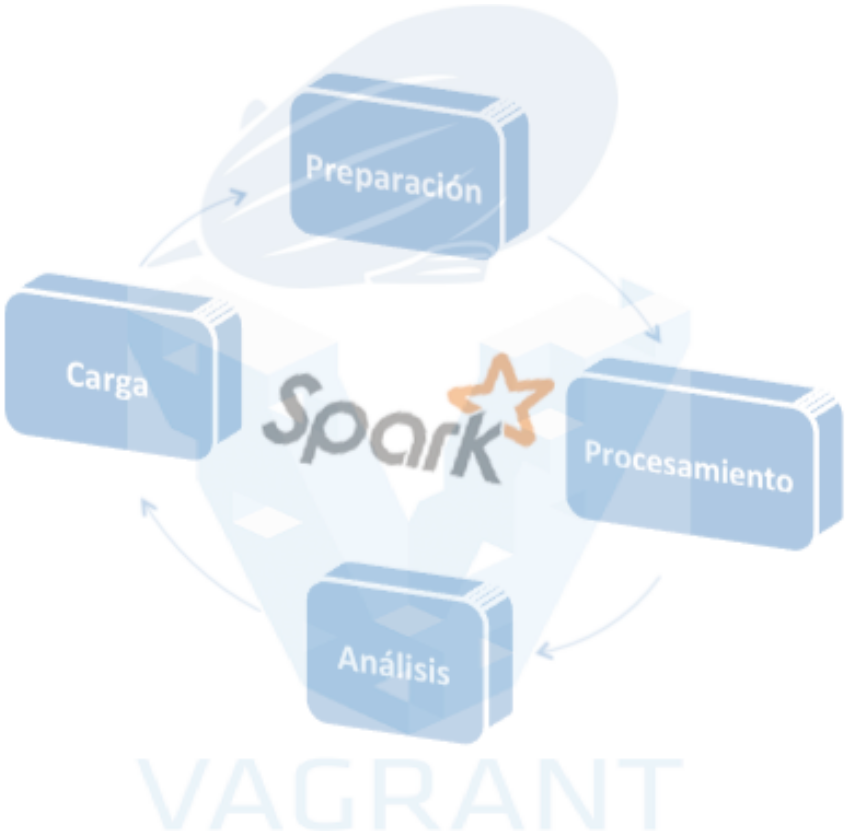
Spark SQL

Ejemplo con sparkR:

```
>sparkR
df <- jsonFile(sqlContext, "data/sql/tweets")
showDF(df)
printSchema(df)
showDF(select(df, "text"))

registerTempTable(df, "tweets")
dfFollowers <- sql(sqlContext, "SELECT user.screen_name,
user.followers_count FROM tweets ORDER BY
user.followers_count")
showDF(dfFollowers)
```

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Spark Streaming

Spark Streaming es una extensión del API Spark core que habilita el procesamiento de streaming de datos de una manera escalable y tolerante a fallos.

Los datos pueden ser ingestados de diversas fuentes:

- Kafka
- Flume
- Twitter
- Sockets TCP

Una vez que los datos son recibidos en ventanas temporales se pueden aplicar las mismas transformaciones y acciones que hemos visto anteriormente y, otros algoritmos de machine learning o de grafos para almacenarlos en diversos destinos como bases de datos, hdfs, …

Spark Streaming

Fco. Javier Lahoz Sevilla

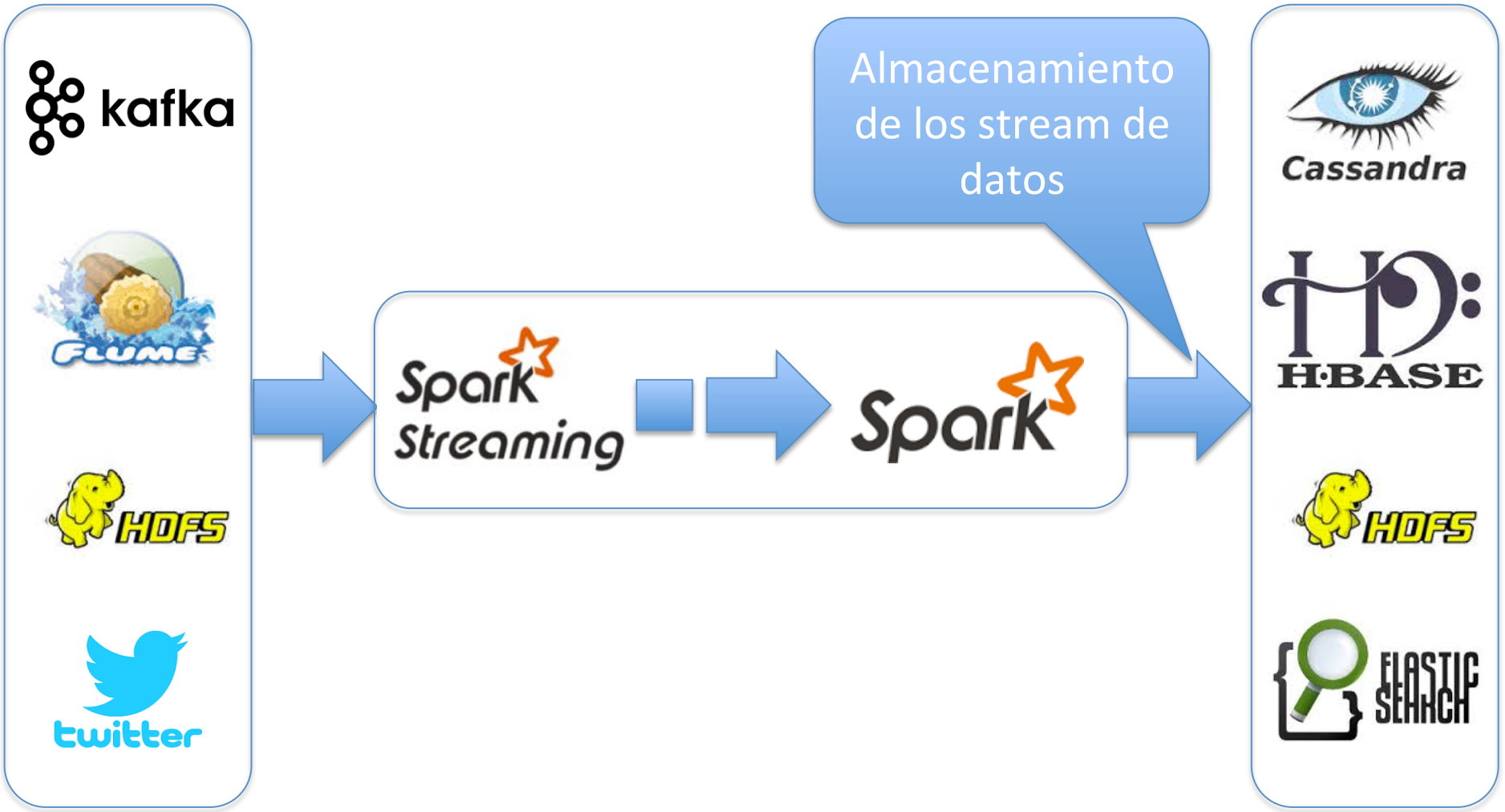
Spark Streaming

Fco. Javier Lahoz Sevilla

Spark Streaming

Fco. Javier Lahoz Sevilla

Spark Streaming



Fco. Javier Lahoz Sevilla

Spark Streaming

Similar a Spark SQL, Spark Streaming ofrece una abstracción de alto nivel llamada Dstream (Discretized Stream) el cual representa un stream continuo de datos.

DStreams puede ser creado para cada una de las fuentes vistas anteriormente para aplicarle las operaciones correspondientes.

En las aplicaciones de Spark Streaming se necesita crear el objeto **StreamingContext** indicando la ventana batch de procesamiento y el contexto de Spark o la configuración de nuestra aplicación.

Spark Streaming

Operaciones que se puede realizar sobre el RDD StreamingContext:

- fileStream
- textFileStream
- socketStream
- socketTextStream
- queueStream
- start()
- stop()
- awaitTermination()

Ejemplo con python.

En este caso vamos ejecutar una aplicación de python a través de spark-submit con el programa NetworkWordCount.py

En un terminal arrancaremos el programa:

>spark-submit --master yarn-client $HOME/src/ NetworkWordCount.py

En otro terminal enviaremos palabras mediante netcat:

>nc -lk 9001

Ejemplo con scala.

Ahora vamos ejecutar una aplicación de scala a través de spark-submit con el programa CursoSparkTwitter.scala.

En un terminal ejecutaremos la aplicación previamente ensamblada (con nuestras claves de Twitter):

>cd $HOME/src/twitter

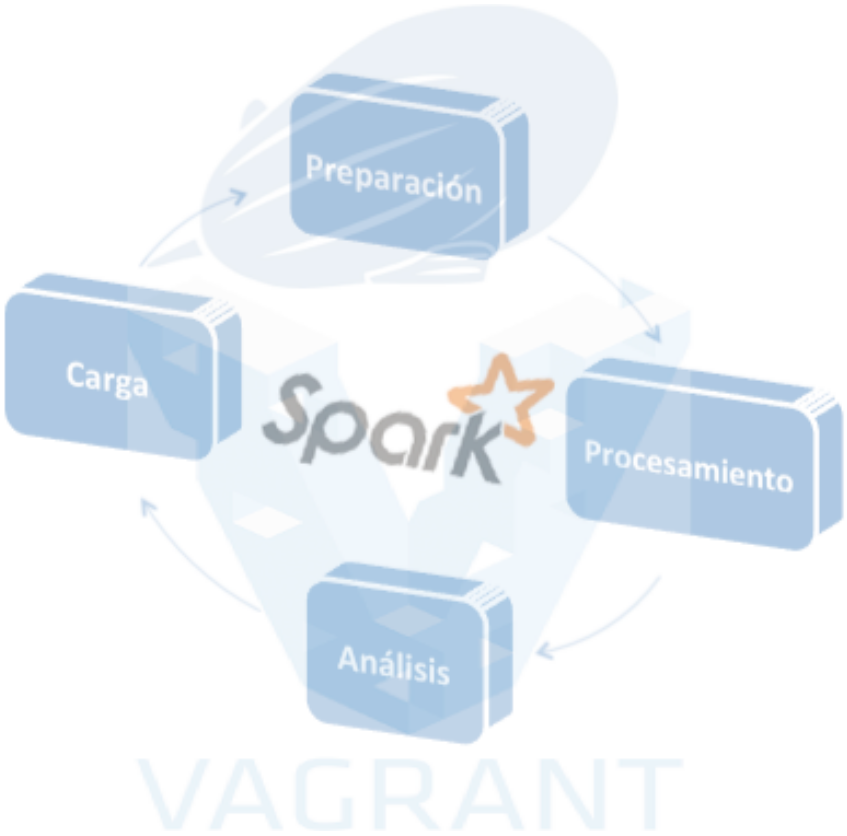
>sbt/sbt assembly

>spark-submit --master yarn-client --num-executors 2 --executor-cores 2 --executor-memory 1g --class CursoSparkTwitter $HOME/src/twitter/target/scala-2.10/Twitter-assembly-0.1-SNAPSHOT.jar

http://twitter4j.org/javadoc/twitter4j/Status.html

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Spark MLLib

Spark MLLib proporciona un conjunto de algoritmos de machine learning escalables y fáciles de implementar.

El API se divide en dos paquetes:

- spark.mllib, contiene el api original construido sobre RDDs.
- spark.ml, proporciona una api de alto nivel construido sobre DataFrames.

Spark MLLib

Algoritmos implementados:

- Estadística básica
  - Correlaciones, testeo de hipótesis
- Clasificación y Regresión
  - Modelos lineares: SVM, regresión lineal y logística
  - Naïve Bayes
  - Arboles de decisión
- Filtros colaborativos
- Cluster
- Reducción de dimensiones
  - SVD
  - PCA

Spark MLLib

Ejemplo de árbol de decisión con scala:

```
>spark-shell
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32

val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
  impurity, maxDepth, maxBins)

// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.count()
println("Test Error = " + testErr)
println("Learned classification tree model:\n" + model.toDebugString)

// Save and load model
model.save(sc, "arbolDecisionSpark")
val myModel = DecisionTreeModel.load(sc, "arbolDecisionSpark")
```

Ejemplo de cluster con scala:

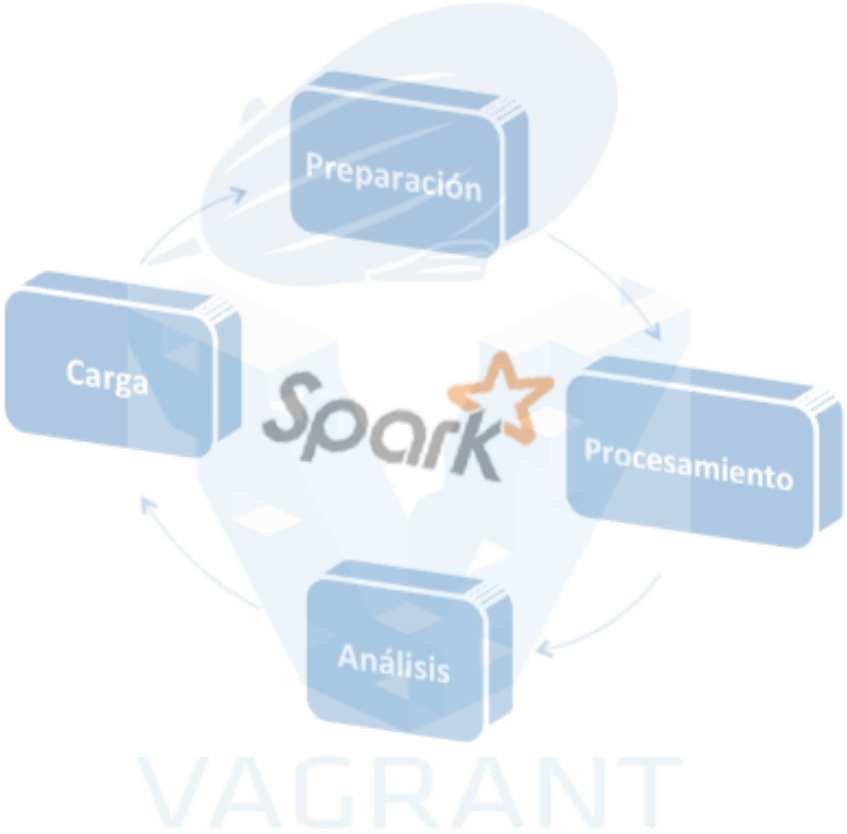
```
>spark-shell
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
import org.apache.spark.mllib.linalg.Vectors

// Load and parse the data
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()

// Cluster the data into two classes using KMeans
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)

// Evaluate clustering by computing Within Set Sum of Squared Errors
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
```

Ejemplo de Regresión Logística con sparkR:

```
>sparkR
# Create the DataFrame
df <- createDataFrame(sqlContext, iris)

# Fit a linear model over the dataset.
model <- glm(Sepal_Length ~ Sepal_Width, data = df, family = "gaussian")

# Model coefficients are returned in a similar format to R's native glm().
summary(model)

# Make predictions based on the model.
predictions <- predict(model, newData = df)
head(select(predictions, "Sepal_Length", "prediction"))
```

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



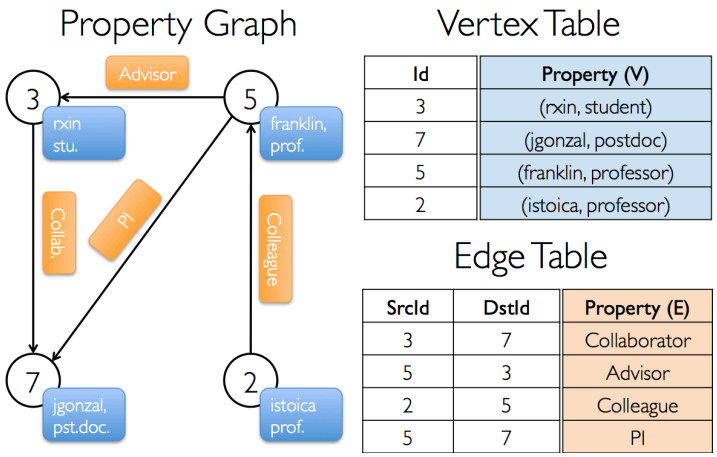
Fco. Javier Lahoz Sevilla

Spark GraphX

GraphX es un nuevo componente de Spark para computación paralela de grafos.

A alto nivel GraphX extiende el Spark RDD introduciendo un grafo abstracto compuesto por:

- Vértices → vertexRDD
- Nodos → edgeRDD



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Operaciones con los grafos:

- filter
- degrees, el grado de cada vértice en el grafo
- numEdges
- numVertices
- pageRank, ejecuta una dinámica versión de PageRank devolviendo un grafo con vértices conteniendo el PageRank y nodos conteniendo los normalizados pesos de los vértices
- connetedComponents, ejecuta los componentes conectados de cada vértice y devuleve un grafo con el valor de vértices que contiene el id vértice más bajo en el componente conectado que contiene ese vértice

Spark GraphX

Ejemplo con scala:

```
>spark-shell
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val vertexArray = Array(
(1L, ("Jose", 28)),(2L, ("Juan", 27)),(3L, ("Maria", 65)),(4L, ("Lurdes", 42)),(5L, ("Javier", 55)),(6L, ("Eva", 50))
)
val edgeArray = Array(
Edge(2L, 1L, 7),Edge(2L, 4L, 2),Edge(3L, 2L, 4),Edge(3L, 6L, 3),Edge(4L, 1L, 1),Edge(5L, 2L, 2),Edge(5L, 3L, 8),Edge(5L, 6L, 3)
)

val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)

graph.vertices.count
graph.edges.count

// Filtramos
graph.vertices.filter(v => v._2._2 > 40).collect.foreach(v => println(s"${v._2._1} is ${v._2._2}"))

for (triplet <- graph.triplets.collect) {
println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")
}
```

Ejemplo de **PageRank** con scala:

```
>spark-shell
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

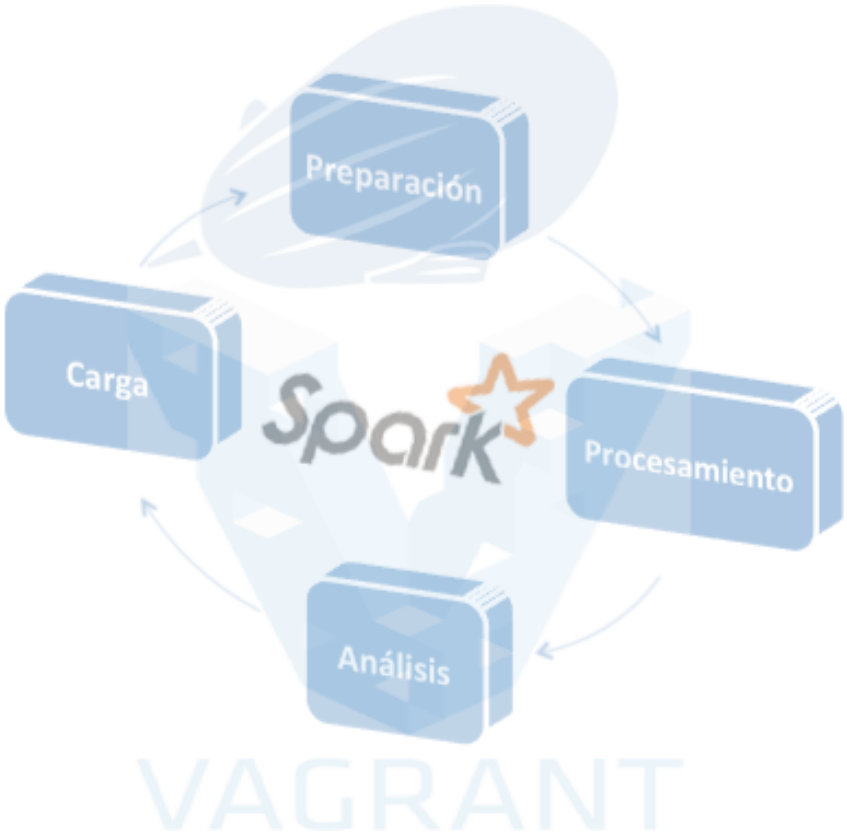
Spark GraphX

Ejemplo de **connectedComponents** con scala:

```
>spark-shell
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

Introducción a Spark

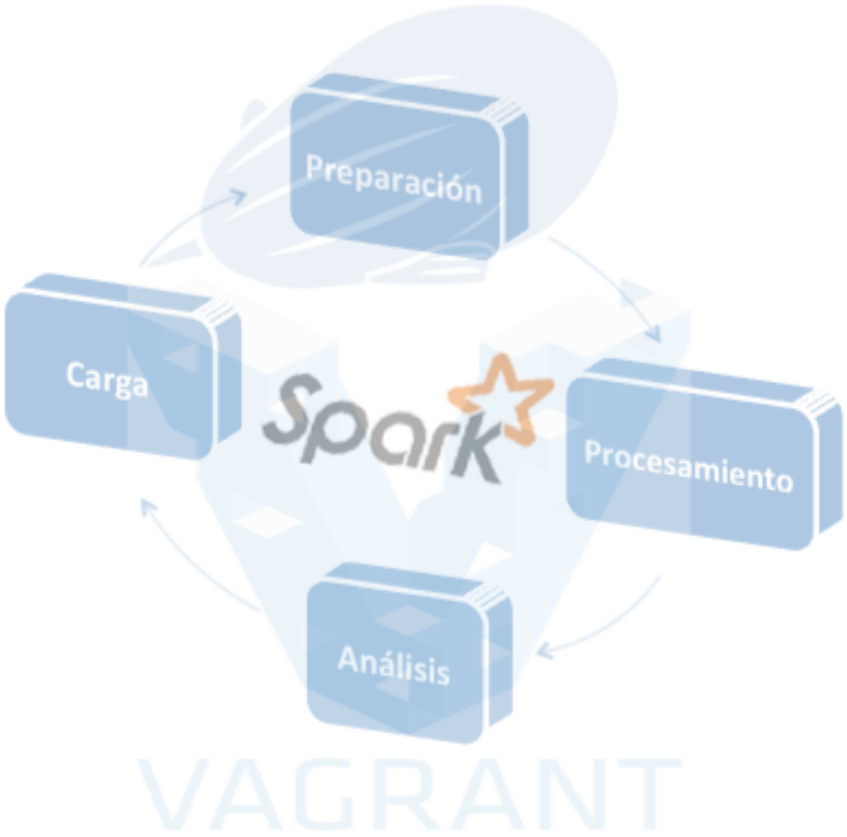
- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Introducción a Spark

- Parte 1. Introducción
  - Vagrant
  - Spark
- Parte 2. Componentes
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX
- Parte 3. Análisis de datos
  - Zeppelin



Fco. Javier Lahoz Sevilla

Zeppelin

Ahora que ya sabemos procesar datos con Spark el siguiente paso sería analizarlos de una manera amigable y visual.

Apache Zeppelin nos proporciona un notebook basado en web para poder cargar y procesar datos, mediante Spark, que nos permite realizar:

- Data Ingestion
- Data Discovery
- Data Analytics
- Data Visualization & Collaboration

Zeppelin

Zeppelin soporta diversos lenguajes:

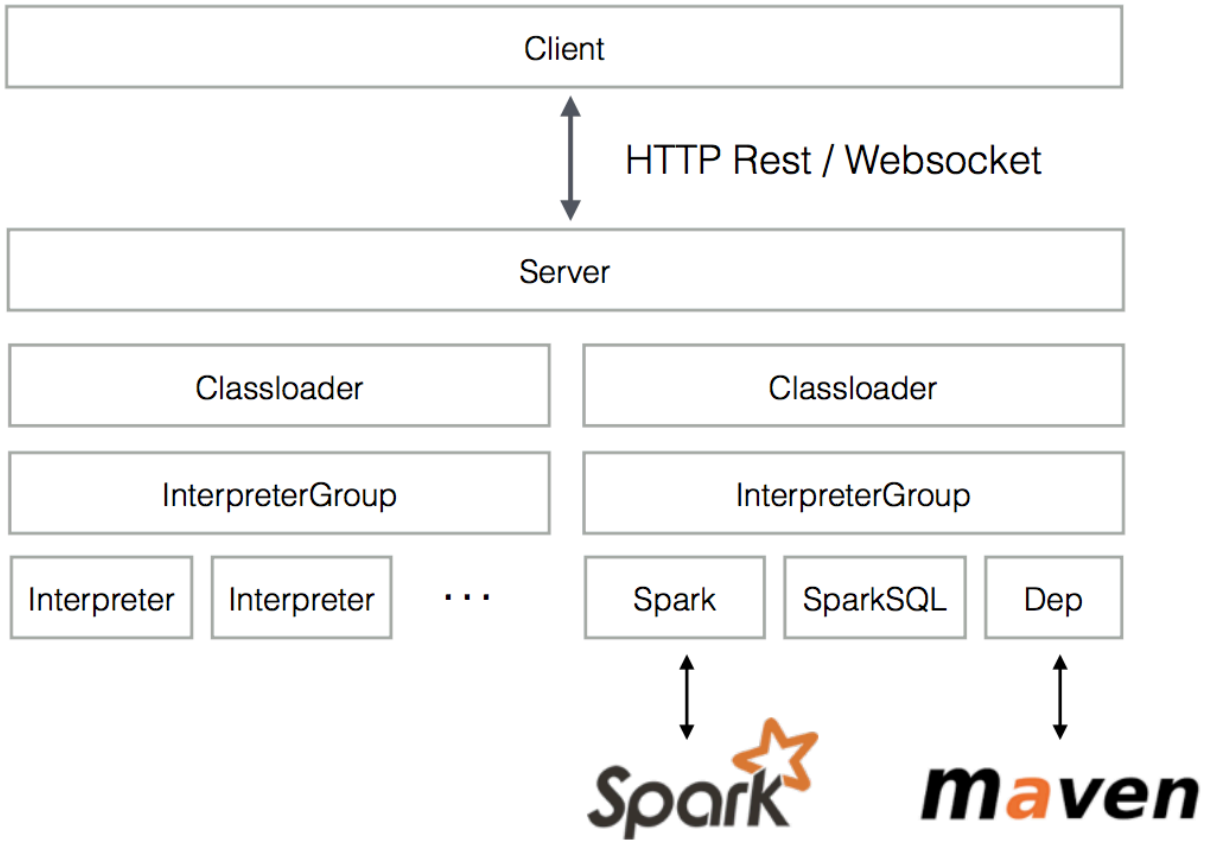
- Scala
- Python
- SQL

Y diversos interpretes asociados a estos lenguajes:

- Spark
- Flink
- Hive
- Cassandra

Zeppelin

Arquitectura:



Zeppelin

Vamos a ver algunos ejemplos.

Primero arrancamos Zeppelin:

>sudo /opt/zeppelin/zeppelin-0.5.0-incubating/bin/zeppelin-daemon.sh start

Después nos podemos conectar a través del navegador:

http://192.168.0.200:8080

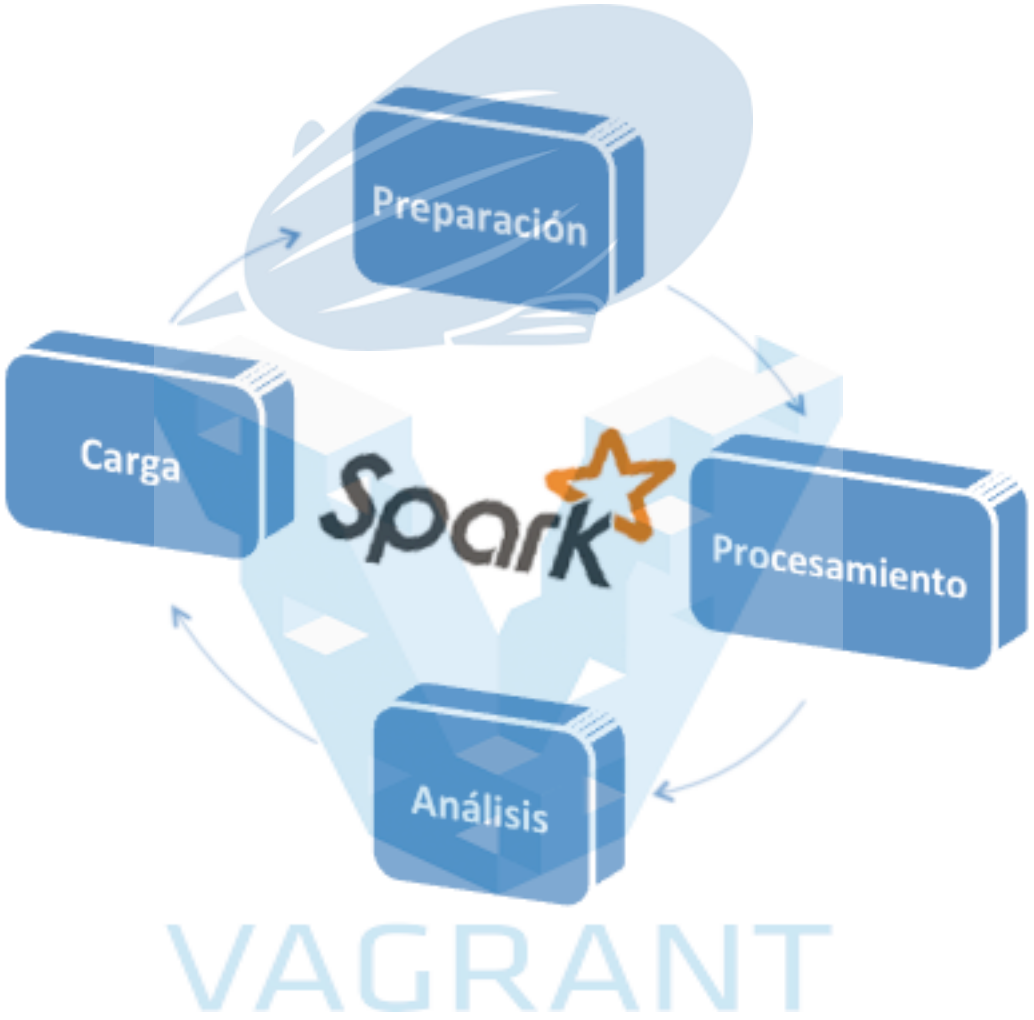
Thank you for your attention!!

Fco. Javier Lahoz Sevila

https://es.linkedin.com/in/fcojavierlahoz

https://twitter.com/FcoJavierLahoz (@FcoJavierLahoz)

Introducción a Spark



Fco. Javier Lahoz Sevilla