Apache Spark

Andrés Gómez Ferrer
andresgomezfrr@gmail.com

¿Qué es Apache Spark?

- Apache Spark es un sistema de computación basado en Hadoop Map Reduce.
- Permitir dividir y paralelizar ***jobs***, que trabajan con datos de manera distribuida.

- Proporciona distintas APIs para funcionar:
    - Core
    - SQL
    - Streaming
    - Graph
    - Machine Learning

Desarrollo

- Spark es multilenguaje y permite desarrollar en:
  - Scala, Java, Lenguaje JVM.
  - Python
  - R
- El framework de Spark desarrollado en **Scala**, **mejor opción**.
- Los analistas de datos trabajan mucho en Python usando **PySpark**.

Index

- **Arquitectura**: Conceptos, Spark Stack
- **Spark Core**: spark-shell, RDD, Core API

Arquitectura

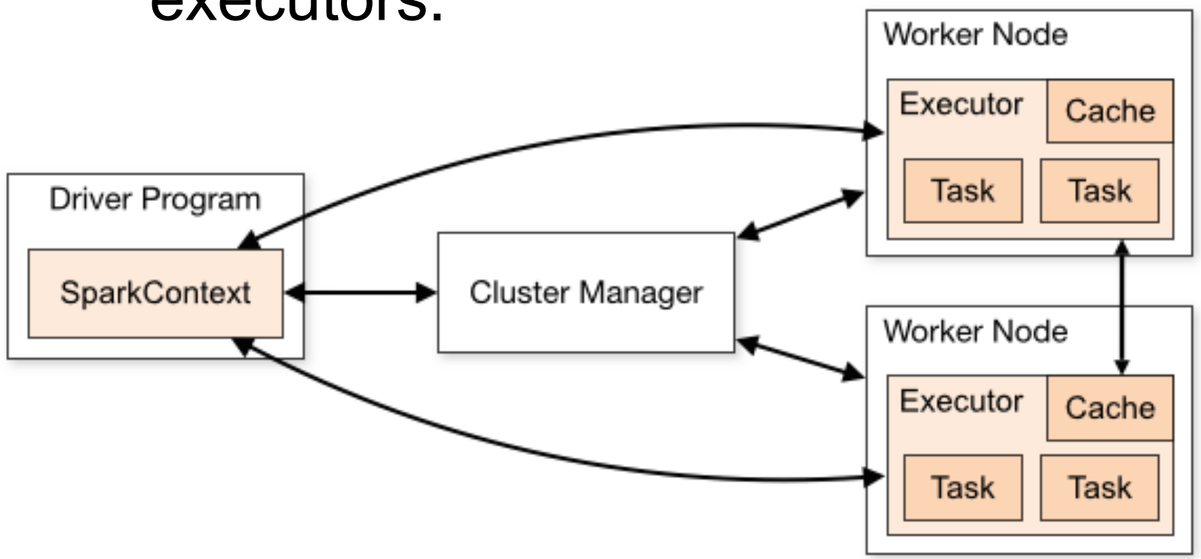
Arquitectura

- Spark proporciona diversas formas de despliegue:
  - local
  - standalone
  - Hadoop YARN
  - mesos
  - kubernetes

- También existes distribuciones/servicios que ayudan a su despliegue, gestión y uso:
  - **Clouds**:
    Amazon EMR, Azure HDInsight, Google DataProc
  - **Distribuciones**:
    Cloudera/Horonworks, Databricks

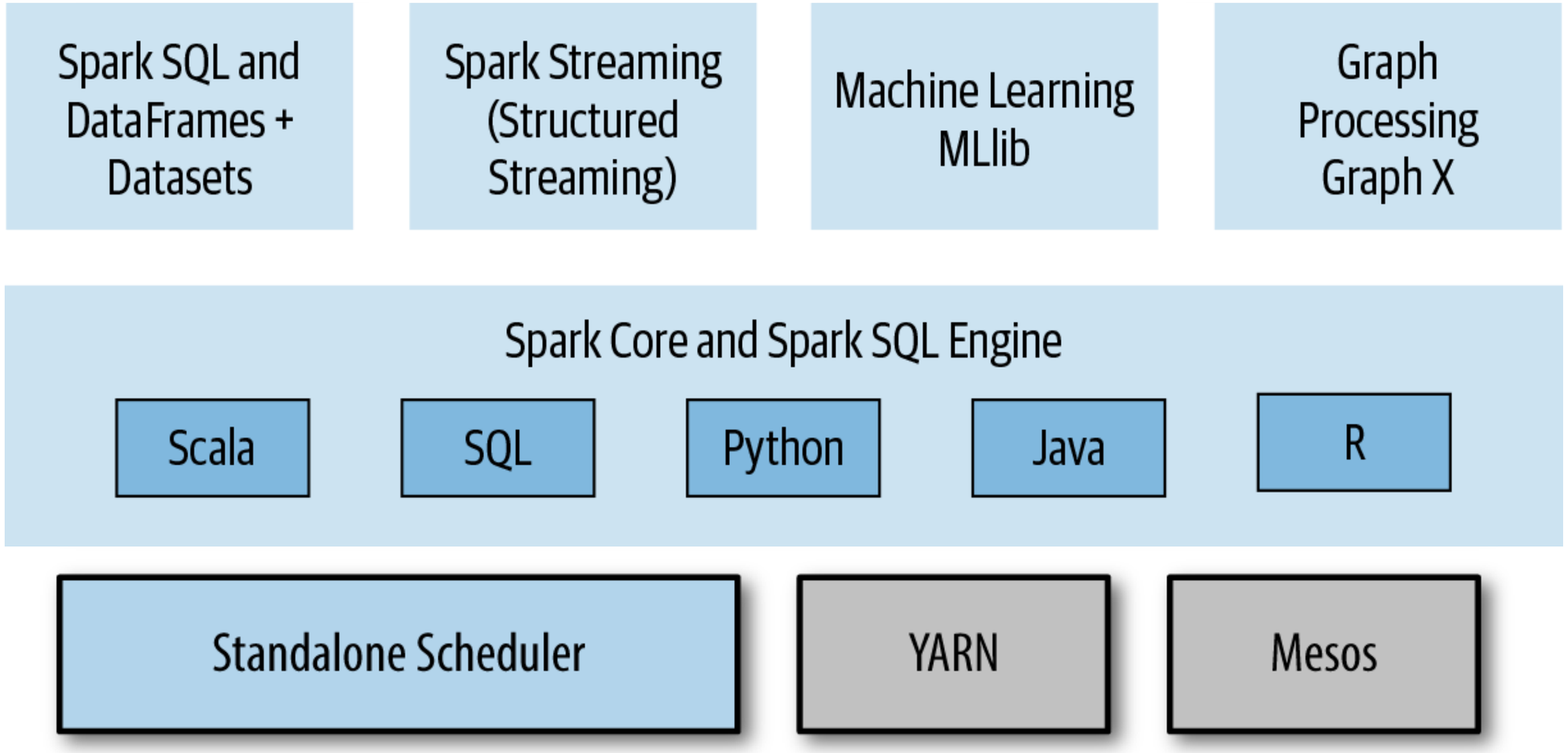
Arquitectura

- Independientemente del *backend* que Spark use, su coordinación se realiza con el **SparkContext**.
- **Cluster Manager:** Comunicación del driver con el backend para adquirir recursos físicos y poder ejecutar los executors.



- **Driver:** Proceso principal, controla toda la aplicación y ejecuta el SparkContext.
- **Worker Node:** Maquinas dependen del backend, y se encargan de ejecutar los procesos de los **executors**.
- **Executors**: Proceso que realizan la carga de trabajo, obtienen sus tareas desde el driver y cargan, transforman y almacenan datos.

Arquitectura: Spark Stack



Arquitectura: Spark Stack

- **Spark Core:** Corazón de Spark, API para procesamiento en batch. Esta API es la base para la construcción de todas las APIs y contiene la base de gestiones de recursos, clústeres e interacción con datos.
- **Spark SQL:** API de Spark que permite trabajar con datos estructurados y semi-estructurados. Proporciona realizar consultas SQL sobre los datos.
- **Spark Streaming:** Procesamiento de datos en streaming, utilizando concepto de microbatches, con latencia de milisegundos.
- **Spark Structured Streaming:** Evolución del motor de Spark SQL para funcionar con streaming, proporcionando trabajar en SQL sobre flujos de datos en tiempo real, consiguiendo ***exactly-one-semantics*** y latencia inferiores a milesegundos.
- **Spark Mlib:** Framework que facilita el uso de algoritmos de machine learning sobre Spark. Clasificaciones, regresiones, clustering, etc.
- **Spark GraphX:** Proporciona procesamiento de grafos distribuidos.

Spark Core

Spark Core: Base Project

- Para empezar un proyecto de Spark, necesitamos añadir sus dependencias en nuestro proyecto de sbt en el IDE.
- **IMPORTANTE: SCALA 2.12.12**

```
libraryDependencies += "org.apache.spark" %%
"spark-core" %  "3.0.0"
```

- Una vez tenemos las dependencias, podemos crear un objeto principal con un método *main*, donde crearemos un nuevo contexto de spark.
- Desde este punto podemos ejecutar en el play del IDE y ejecuta spark en modo **local[1], modo local, usando 1CPU de la maquina.**

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object SparkBase {
  def main(args: Array[String]): Unit = {

    val conf = new SparkConf()
      .setAppName("KeepcodingSparkBase")
      .setMaster("local[1]")
    val sc = new SparkContext(conf)
    // App Spark Code
    sc.stop()
  }
}
```

Spark Core: Spark-Shell

- La distribución de spark (https://spark.apache.org/downloads.html), contiene un binario ***spark-shell***, que proporciona una PERL interactiva para trabajar con spark.
- Ejecutar la shell es tan sencillo como, esto arranca un cluster de spark en modo local usando todos los cores disponibles de la maquina e iniciara un sparkContext, en la variable **sc**:

```
$ ./bin/spark-shell --master "local[*]"
```

- Spark-shell también permite añadir ficheros jars de librería o nuestro propio código fuente o descargar librerías externas usando maven.

```
$ ./bin/spark-shell --master "local[*]" --jars code.jar
$ ./bin/spark-shell --master "local[*]" --packages "org.example:example:0.1"
```

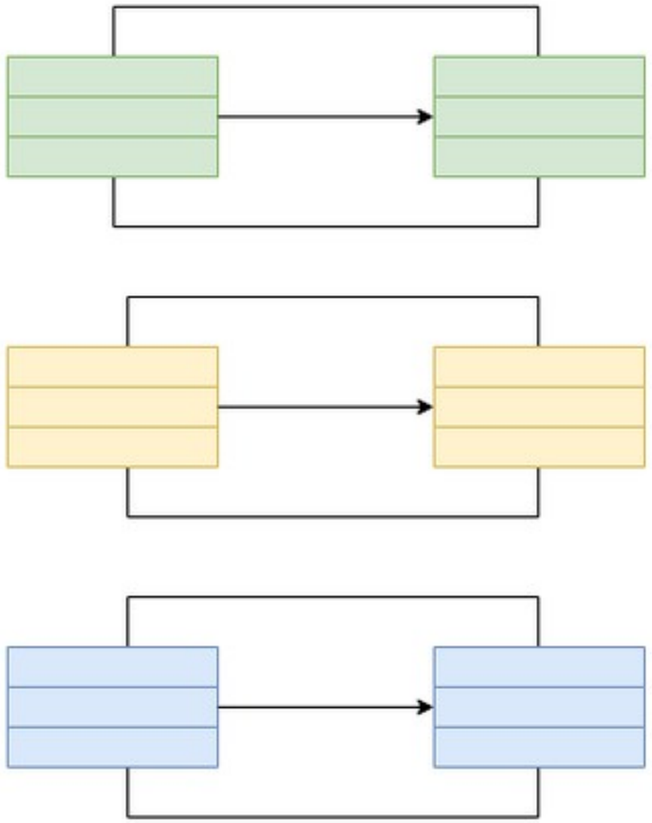
RRD: Resilient Distributed Datasets

- Spark trabaja con los datos bajo un concepto denominado RDD.

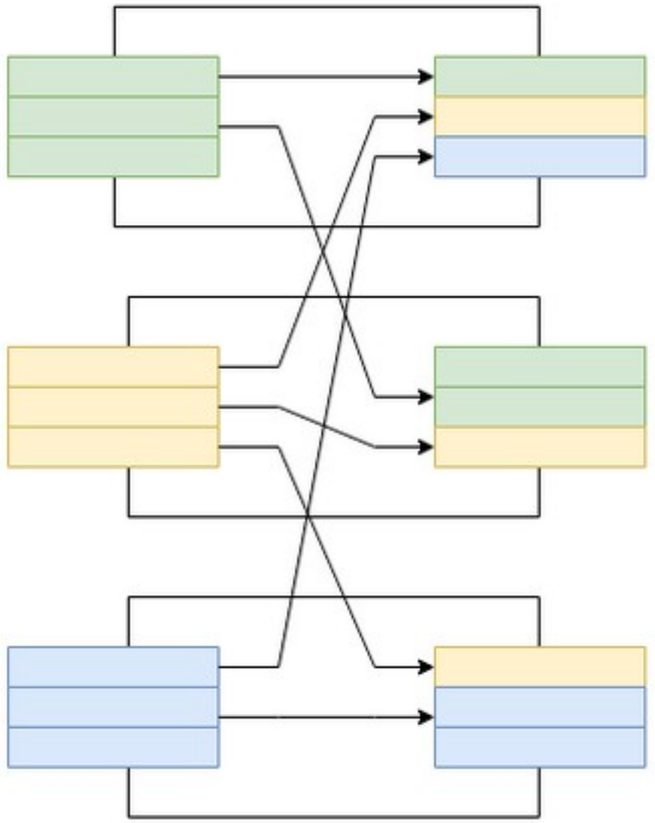
- Los RDDs tienen las siguientes características:
  - **Inmutables**: No se pueden modificar una vez creados.
  - **Distribuidos**: Divididos en particiones que estas repartidas en el cluster.
  - **Resilientes**: En caso de perder una partición se regenera automáticamente.

- Los RDDs se transforman, creando nuevos RDDs, estas transformaciones se aplican a los datos, por lo que trabajar con un RDD es trabajar con el conjunto de datos. Las transformaciones son de dos tipos:
  - **Narrow**: No necesitan intercambio de información entre los nodos del cluster.
  - **Wide:** Necesitan intercambio de información entre los nodos del cluster.

RRD: Resilient Distributed Datasets



Transformación narrow



Transformación wide

RRD: Resilient Distributed Datasets

- Existen distintas formas de generar RDDs:
  - Obtener datos de un fichero.
  - Distribución de datos desde el driver.
  - Transformar un RDD, para crear un nuevo RDD.

- Los RDD permiten dos tipos de operaciones: transformaciones y acciones.
  - **Transformaciones**: Consiste en generar un RDD a partir de otro RDD. Nos permite trabajar con datos y generar nuevos. *map, flatMap, filter…*
  - **Acciones**: Suelen ser puntos finales de procesamiento, devuelven un valor al driver o envían datos a una fuente externa. *count, collect, saveAsTextFile…*

RRD: Ciclo de Vida

Spark Core: parallelize

- Spark permite distribuir datos a través del cluster directamente desde el driver.

```
$ ./bin/spark-shell --master "local[*]"

val localData = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val distData = sc.parallelize(localData)
```

- Los datos son distribuido en el cluster en particiones, spark decide cuantas particiones crea para cada distribución de datos, basándose en el número de cores del cluster. Aunque podemos indicar específicamente el número de particiones que queremos:

```
val distData20 = sc.parallelize(localData, 20)
```

- Podemos comprobar el número de particiones de nuestros datos de la siguiente forma:

```
distData.partitions.size
distData20.partitions.size
```

- Mostar distribución de datos por partición:

```
distData.foreachPartition(x => println(x.mkString(",")))
```

RRD: Transformaciones / Lazy evaluation

- Las transformaciones en RDD se evalúan perezosamente, lo que significa que Spark no comenzará a ejecutarse hasta que se muestre o se lance una acción.

- En lugar de pensar en un RDD que contiene datos, es mejor pensar como en un conjunto de instrucciones sobre cómo calcular los datos que construimos a través de transformaciones.

- Spark utiliza la evaluación diferida para reducir el número de pasadas para hacerse cargo en nuestros datos agrupando las operaciones.

Exercise 1

Spark API: Transformaciones Simples

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- sample
- union
- intersection
- distinct
- pipe
- coalesce
- repartition
- cartesian
- groupBy

RRD: Key/Value Pairs

- RDDs donde cada elemento de la colección es una tupla de dos elementos: CLAVE -> VALOR
- Pueden ser generados:

```
val keyValuePairRDD = sc.parallelize(Seq((1,2), (2,3)))
```

- Construirse a mediante transformaciones:

```
val words = sc.parallelize(List("avion", "tren", "barco", "coche", "moto", "bici"), 2)
val rddWithKey = words.keyBy(_.length) // se usa la longitud de la palabra como clave
rddWithKey.groupByKey.collect()
```

Exercise 2

Spark API: Transformaciones K/V Pairs

- groupByKey
- reduceByKey
- aggregateByKey
- sortByKey
- join

RDD: Acciones

- Las acciones en Spark, provocan procesamiento de datos.
- Cuando se ejecuta una acciones se aplican todas las traformaciones planificadas y finalmente la acción.
- Las acciones provocan que los datos se evalúen desde el origen aplicando todas las transformaciones.

múltiples acciones === multiples evaluaciones de los datos desde el origen

- Existen acciones que mueven datos al proceso del Driver y otras que se ejecutan directamente en los executors.

IMPORTANTE: No llevar demasiados datos al driver, poco recursos

Exercise 3

Spark API: Acciones

- collect
- count
- first
- take
- takeSample
- countByKey / countByValue
- foreach
- reduce

- foreachPartition
- saveAsTextFile

- Provocan llevar datos al driver
- Usado para enviar datos a sistemas externos: BBDD, Sistemas de colas, servicios REST

Spark Core: textFile / wholeTextFiles

- Spark crea dataset desde cualquier almacenamiento soportado por Hadoop: Sistema local de ficheros, HDFS, Cassandra, Hbase, Amazon S3, Google Storage, Azure File System, etc.
- Spark soporta ficheros de texto, secuencias, y cualquier formato soportado por Hadoop InputFormat.
- Los ficheros pueden ser leídos usando el sparkContext, devolviendo un record por línea:

```
val distFile = sc.textFile("data.txt")
```

- Funciona con directorios, ficheros comprimidos y comodines *

- **Nota**: Usando el sistema local, los ficheros tienen que estar disponibles en todos los executors.

Exercise 4

Spark API: read-operate-write

1. Leer los datos del fichero sample.txt usando al función *`sc.textFiles(...)`*
2. Contar el numero de lineas e imprimir por pantalla
3. Filtrar unicamente las palabras que comiencen por la letra *`T`* o *`t`*
4. Imprimir el path completo donde se van a escribir los datos y escribir los resultados en la carpeta de resources dentro de una carpeta de nombre aleatorio

***Preguntas**:*

1. ¿Cuántas acciones se hace en este job?
2. ¿Podríamos hacer algo para optimizar el job?

Exercise 5

Works with sample datasets

- WordCount
- TopN
- JSON Data & Unit Testing

Exercise 6

Spark UI

Exercise 7

Spark with GoogleCloud:
Dataproc & Google File System