Apache Spark

KEEPCODING
Tech School

Antonio Gutiérrez López
gutierrez.lopez.ant@gmail.com

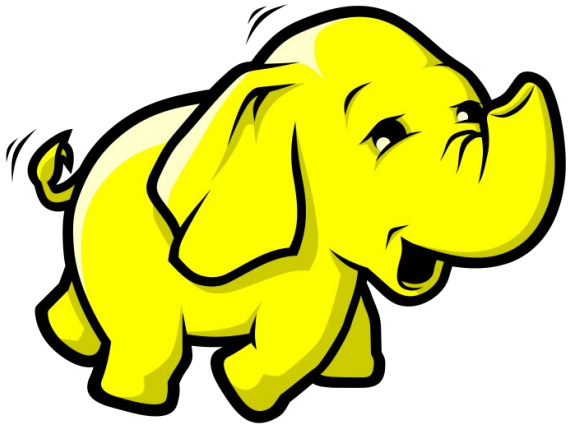
Big Data y Hadoop

Hadoop como respuesta al problema del Big Data inspirado en las ideas de GFS y de Google MapReduce

Proyecto ayudado por Yahoo! y liberado como proyecto open source en la Apache Software Foundation

Compuestos de tres módulos principales:

- HDFS (Hadoop Distributed File System)
- Hadoop YARN
- Hadoop MapReduce



Problemas de Hadoop

- Utiliza el disco como mecanismo de comunicación entre procesos.
- API aparatosa, rudimentaria y fea.
- El concepto MapReduce es funcional, Java No.
- Mucho código para poca funcionalidad.
- Hay mucho código repetido
- Hay que instalar un montón de historias para poder hacerlo funcionar.

Apache Spark al rescate

- Creado por la Universidad de Berkeley
- Apoyado por Databricks
- Es un framework de procesamiento distribuido
- Tiene un API muy potente y muy fácil de utilizar
- Un gran ecosistema propio

Apache Spark al rescate

El disco duro como un recurso más:

- Se realiza procesamiento en memoria. Se puede usar almacenamiento como entrada/salida y como temporal

Datos Inmutables:

- Implica que los procesos son reproducibles y tolerantes a fallos

Una API poderosa:

- Escrita en Scala, lenguaje funcional para el paradigma MapReduce
- Librerías para distintos casos de uso (MLlib, graphX, etc..)

Unión de procesamiento Batch y Procesamiento Streaming

¿Qué es Apache Spark?

- Apache Spark es un sistema de computación distribuida en memoria basado en **Hadoop Map Reduce**.
- Permite dividir y paralelizar ***jobs***, que trabajan con datos de manera distribuida.
- Proporciona distintas APIs para funcionar:
  - Core
  - SQL
  - Streaming
  - Graph
  - Machine Learning

Desarrollo en Spark

- 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.


Indice

- **Arquitectura Spark**
- **Spark Core**
  - Spark Shell
  - RDD
  - Core API

Arquitectura Spark

KEEPCODING
Tech School

Arquitectura

Spark proporciona diversas formas de despliegue:

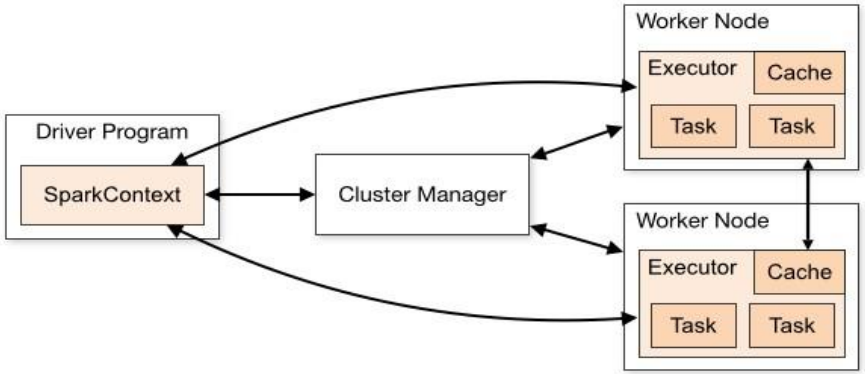
- Local
- Standalone
- Hadoop YARN
- Mesos
- Kubernetes

También existes distribuciones/servicios que ayudan a su despliegue, gestión y uso:

- **Clouds**: Amazon EMR, Azure HDInsight, Google DataProc
- **Distribuciones**: Cloudera/Hortonworks, Databricks

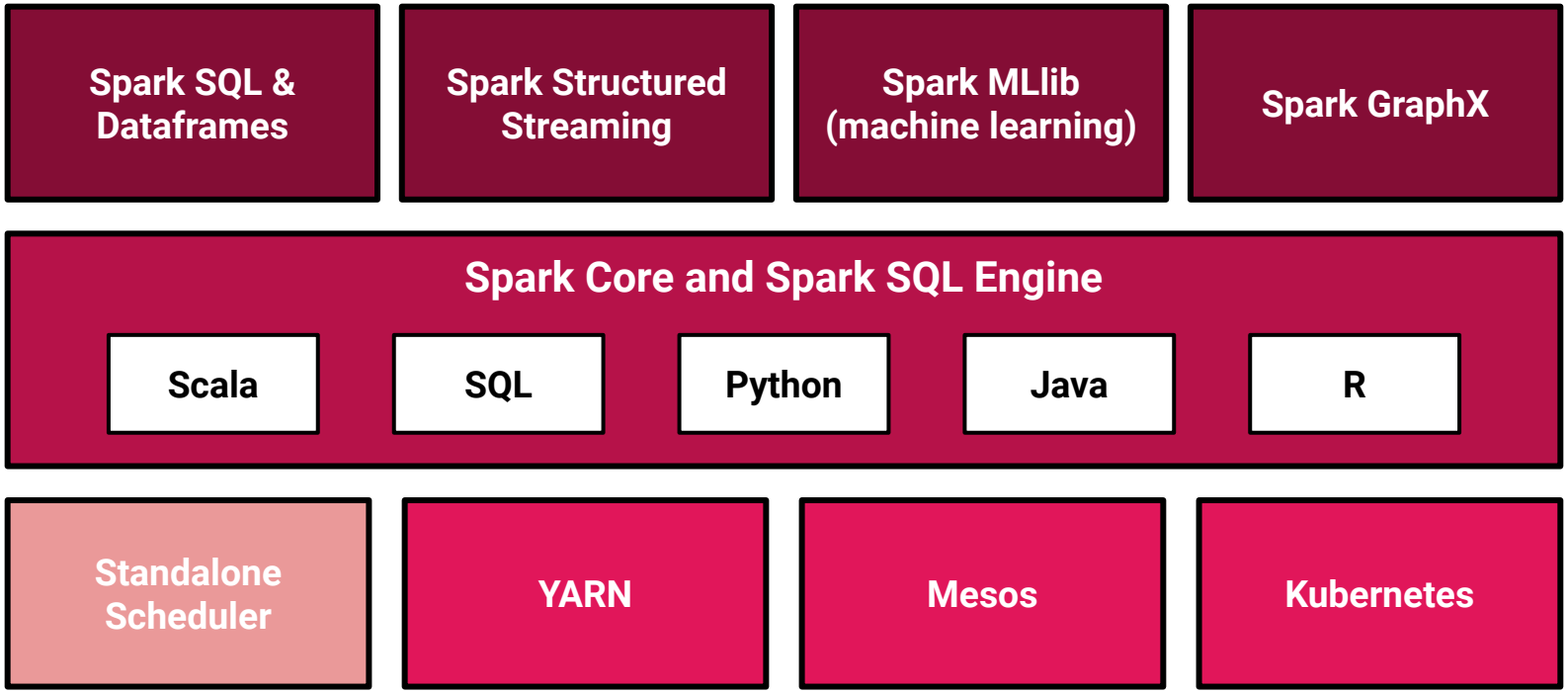
Arquitectura

- 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:** Máquinas 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 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 milisegundos.

Arquitectura: Spark Stack

- **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.

Deprecado:

- **Spark Streaming:** Procesamiento de datos en streaming, utilizando concepto de microbatches, con latencia de milisegundos.

Spark Core

KEEPCODING
Tech School

Spark Core: Spark Shell

- La distribución de spark (https://spark.apache.org/downloads.html), contiene un binario spark-shell, que proporciona un REPL interactivo para trabajar con spark (como la REPL de Scala)
- Al ejecutar la **spark-shell** se lanza un cluster de spark en modo local en el que podremos empezar a trabajar. Se nos creará un **sparkContext** y una **sparkSession** automáticamente.
- Se pueden pasar distintos parámetros a la spark-shell, por ejemplo:

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


Spark Core: Proyecto básico

Para empezar un proyecto de Spark, necesitamos añadir sus dependencias en nuestro proyecto de sbt en el IDE:

- Usaremos la última versión de Scala 2.12.x disponible. Actualmente tenemos la versión **2.12.17**
- Añadimos la dependencia de **spark-core** en nuestro **build.sbt:**

```
libraryDependencies += "org.apache.spark" %% "spark-core" %  "3.3.0"
```

- Una vez tenemos las dependencias, podemos crear un objeto principal con un método **`main`**, donde crearemos un nuevo **sparkContext** y una nueva **sparkSession**.

Spark Core: Proyecto básico

```
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()
    }
}
```


RDD

KEEPCODING
Tech School

RDD: Resilient Distributed Dataset

Spark trabaja con los datos bajo un concepto denominado **RDD, Resilient Distributed Dataset**

Los **RDDs** tienen las siguientes características:

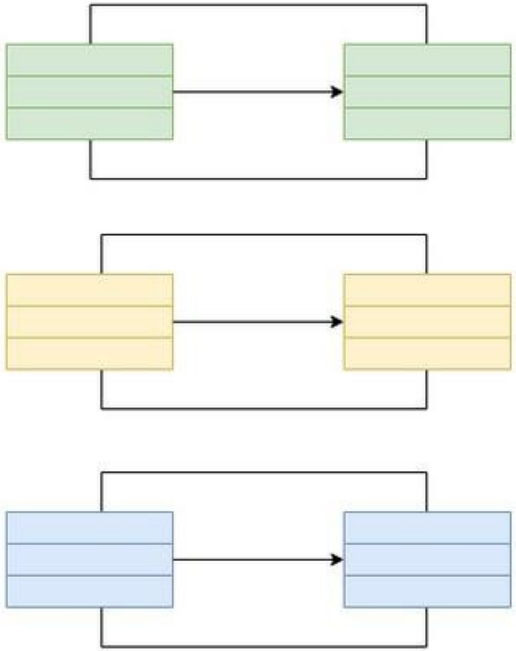
- **Inmutables:** No se pueden modificar una vez creados.
- **Distribuidos:** Divididos en particiones que están repartidas en el clúster.
- **Resilientes:** En caso de perder una partición se regenera automáticamente.

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.

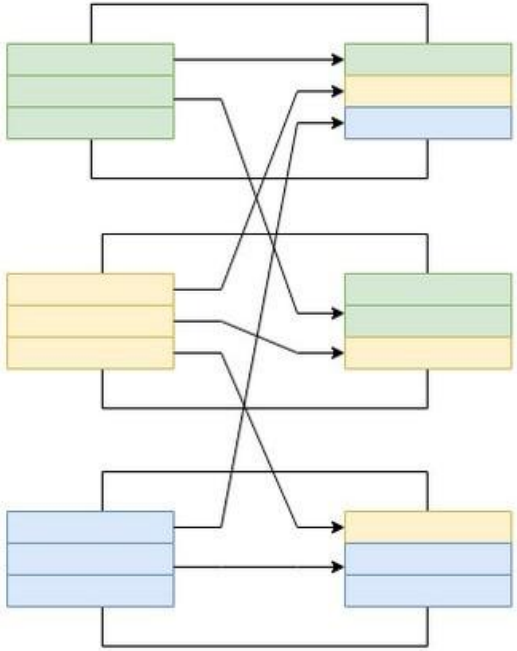
Las transformaciones son de dos tipos:

- **Narrow:** No necesitan intercambio de información entre los nodos del cluster.
- **Wide:** Necesitan intercambio de información entre los nodos del cluster.

RDD: Resilient Distributed Dataset



Transformación narrow

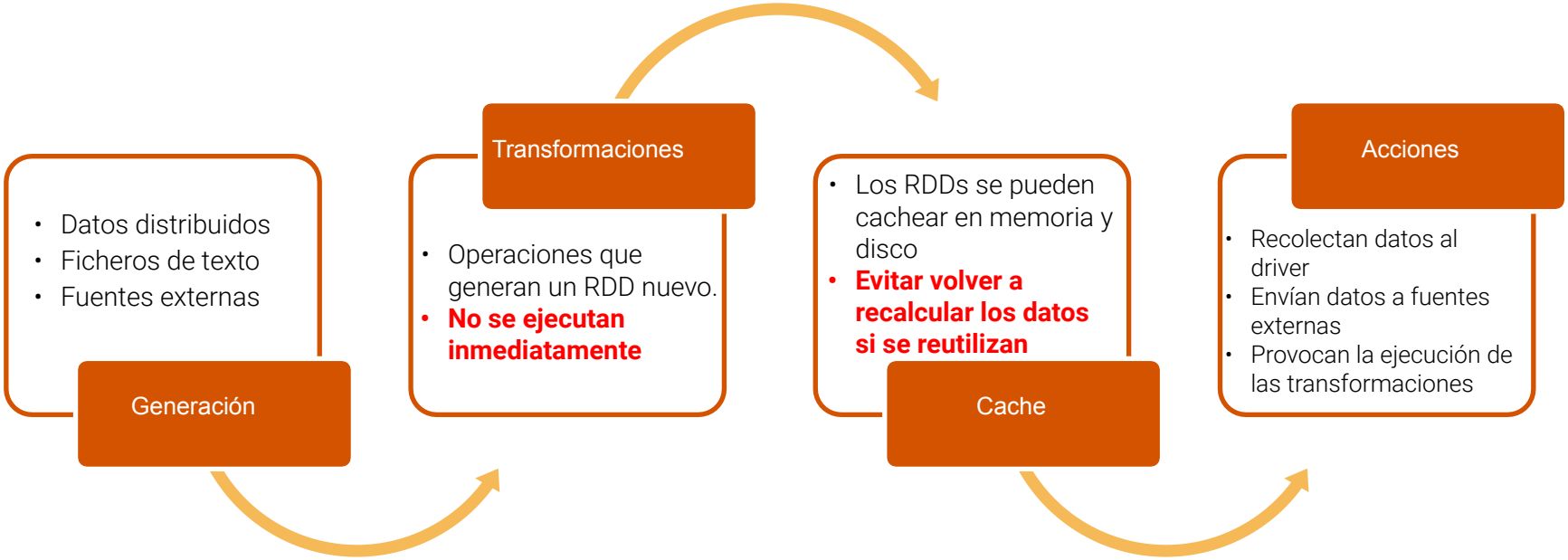


Transformación wide

RDD: Resilient Distributed Dataset

- 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...`***

RDD: Ciclo de vida



RDD: Parallelize

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

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

- Los datos son distribuidos en el cluster en particiones, spark decide cuántas 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)
distData20.partitions.size // 20
distData20.foreachPartition(x => println(x.mkString(",")))
```

RDD: 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 de nuestros datos agrupando las operaciones.

Exercise 1

Spark API: Transformaciones Simples

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

RDD: 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 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 transformaciones planificadas y finalmente la acción.
- Las acciones provocan que los datos se evalúen desde el origen aplicando todas las transformaciones.

```
múltiples acciones === múltiples 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

- foreach
- reduce
- saveAsTextFile

- collect
- count
- first
- take
- takeSample
- countByKey
- countByValue

- foreachPartition

- Llevan 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 registro 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 la función `sc.textFiles(...)`
2. Contar el número de líneas e imprimir por pantalla
3. Filtrar únicamente 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:

¿Cuántas acciones se hacen en este job?
¿Podríamos hacer algo para optimizar el job?

Exercise 5

Work with sample datasets

- WordCount
- TopN
- JSON Data & Unit Testing

Exercise 6

Spark UI

Exercise 7

Spark with GoogleCloud:
Dataproc & Google File System

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá

Datos de contacto