Ingeniería para el Procesado Masivo de Datos

Dr. Pablo J. Villacorta

Tema 3. Apache Spark I

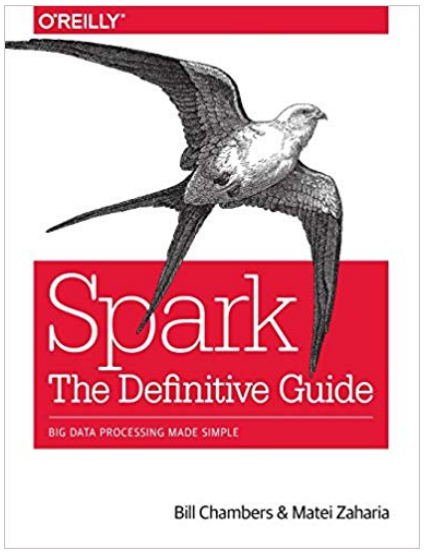
Noviembre de 2022

unir LA UNIVERSIDAD EN INTERNET

Universidad Internacional de La Rioja

Objetivos del tema

- Conocer el framework de ejecución Apache Spark y sus diferencias respecto a Hadoop MapReduce
- Entender el funcionamiento de una aplicación que utilice Spark
- Conocer los módulos principales que integran Spark, con énfasis en SparkSQL y SparkML
- Entender conceptos básicos como RDD, DataFrame, DAG, transformaciones, acciones, UDFs, jobs, stages, tasks
- Conocer y comprender la arquitectura de Spark cuando se despliega en un cluster


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia

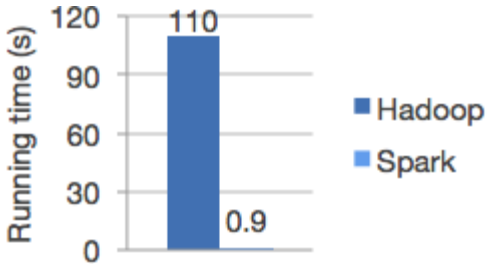
Apache Spark

Apache Spark es un motor unificado de cálculo y un conjunto de bibliotecas para procesamiento paralelo y distribuido de datos en clusters de ordenadores

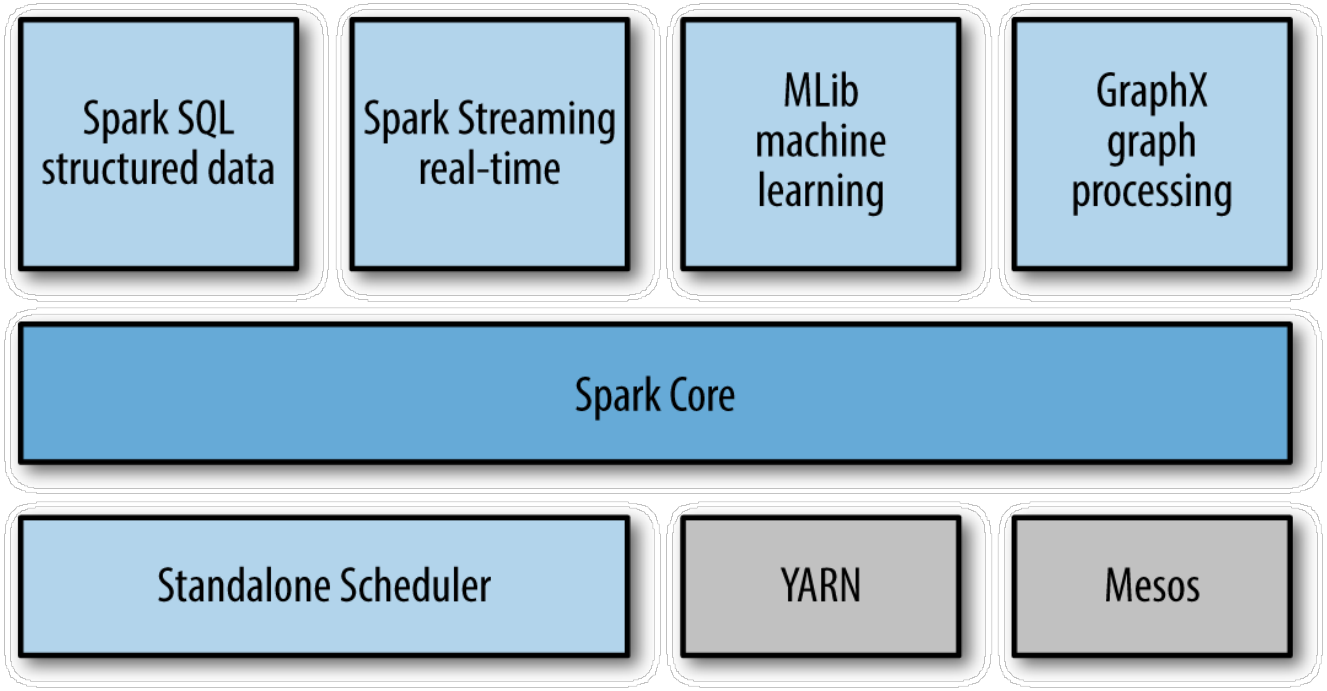
- Bibliografía recomendada: *Spark – The Definitive Guide*, O'Reilly, 2018. https://www.amazon.es/Spark-Definitive-Guide-Bill-Chambers/dp/1491912219
- Según esto, MapReduce también encajaría en la definición…
  - Spark trabaja *en memoria principal* (RAM de los nodos del clúster). Solo acude a disco si usamos explícitamente instrucciones de lectura o escritura.
- *Unificado*: el motor de cálculo es **único** e independiente de cómo utilicemos Spark:
  - Desde la API de DataFrames (R, python, Java, Scala)
  - Desde herramientas externas que lanzan consultas SQL contra Spark (ej: Hive, herramientas de BI conectadas a Spark, Tableau, PowerBI, etc)
  - Desde una instrucción de la API que recibe una consulta SQL como string
  - …todo se traduce a un grafo de tareas al que Spark aplica optimizaciones de código automáticamente → todos (API, BI, SQL, etc) se benefician de ellas
- Spark está escrito en Scala (la mayoría) y Java. Las interfaces para python y R son una capa intermedia que acaba llamando a la JVM para el procesamiento distribuido.

Apache Spark

- API (bibliotecas) orientada al programador, **mucho** más intuitiva que MapReduce
  - Abstrae todos los detalles de comunicación y hardware, pero además opera de manera similar a las consultas SQL tradicionales como si los datos fuesen tablas
- Spark ofrece APIs para cuatro lenguajes:
  - Java: muy tediosa de usar pero útil para aplicaciones legacy existentes
  - Scala: la más utilizada para aplicaciones en producción. Lenguaje funcional que se compila sobre la JVM (genera bytecode de Java). Muy compacto y cómodo de utilizar con Spark
  - Python (paquete de Python *pyspark*)*:* API casi idéntica a la de Scala, salvo peculiaridades del lenguaje. Wrapper (capa adicional) sobre la implementación en Scala que es la verdaderamente distribuida.
  - R: (paquete de R *SparkR*): API muy diferente al resto, más cercana a los comandos típicos de R. Últimamente, el paquete *sparklyr* (de RStudio) es más útil, ofrece mejor integración y es más intuitivo.
- *En memoria (in-memory)*: rendimiento hasta 100 veces superior que MapReduce en tareas iterativas (varias pasadas sobre los mismos datos, como tareas de machine learning). Medición de tiempos en una regresión logística:

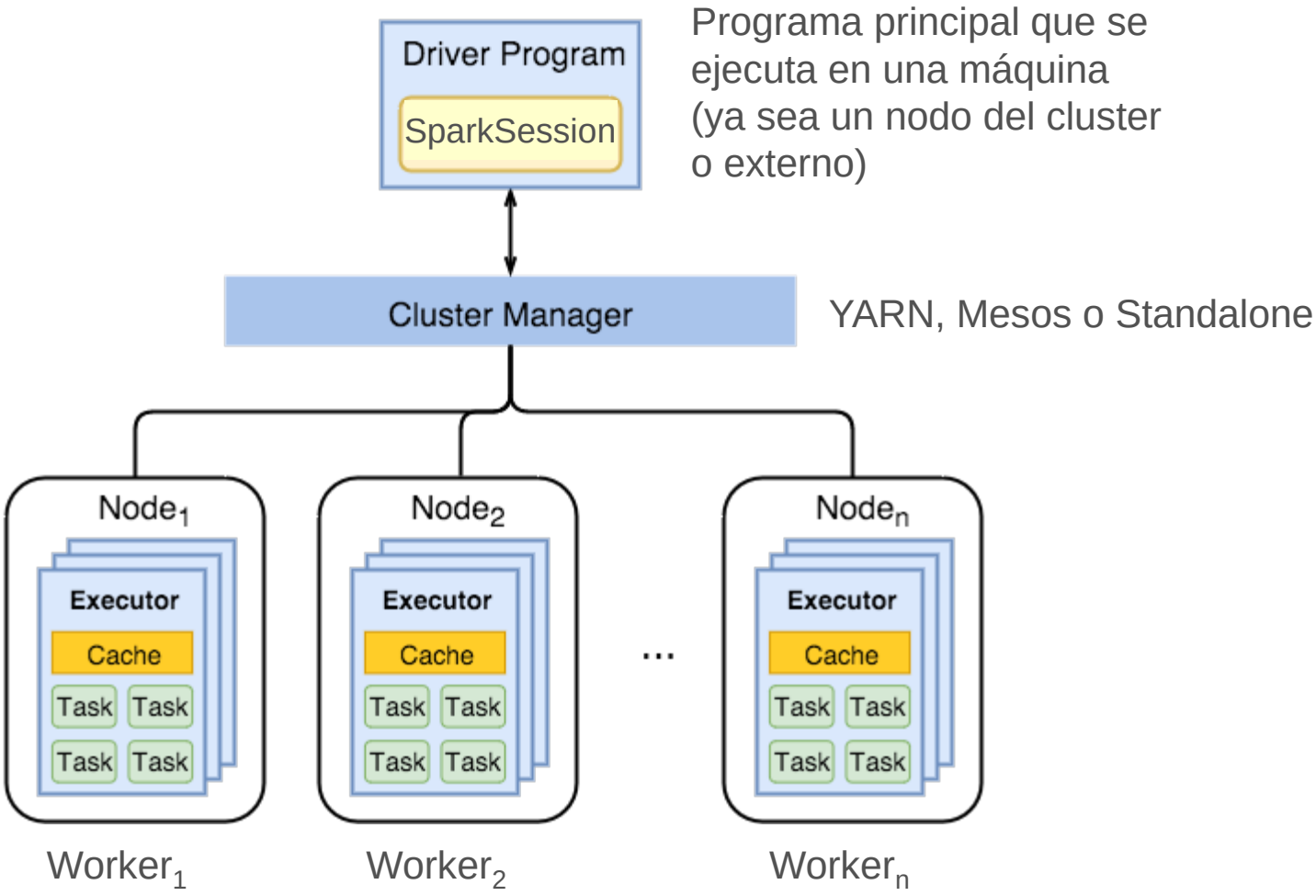


Componentes de Spark



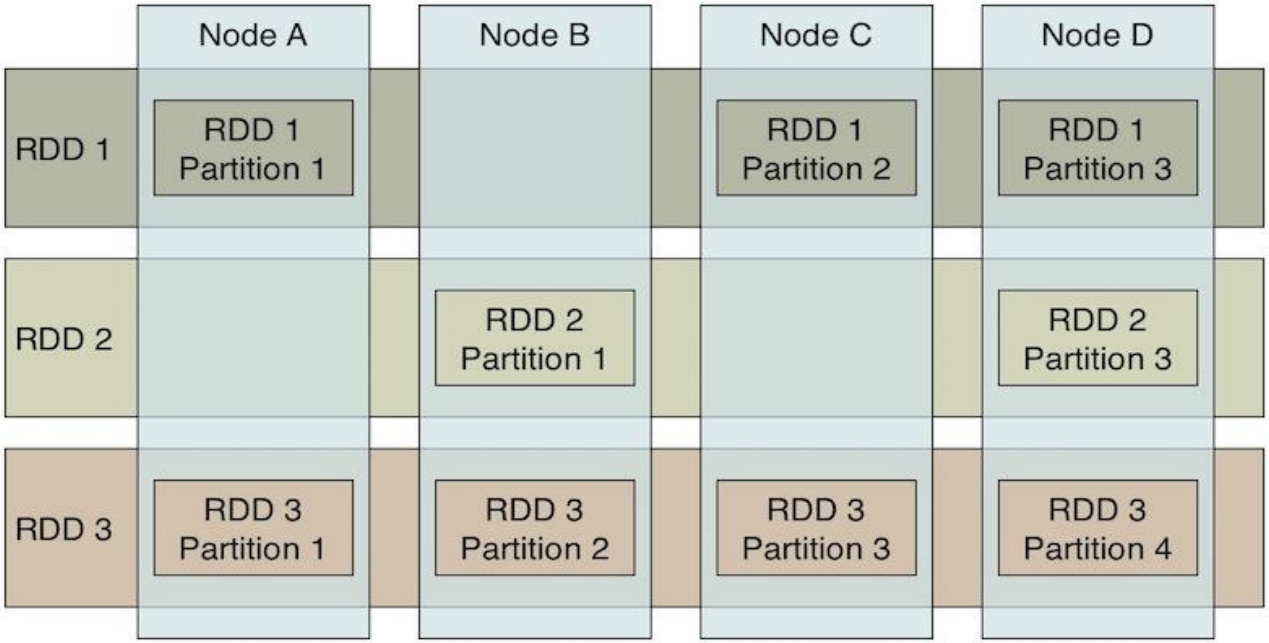
- Falta un nuevo módulo llamado *Structured Streaming* (introducido en Spark 2.0) que abstrae la mayoría de los detalles de Spark Streaming y simplifica su uso
- El *core* de Spark incluye los mecanismos de comunicación y gestión de la memoria, tareas, etc y la abstracción principal de Spark: los RDD
- Yarn y Mesos son dos gestores de un cluster, que permiten planificar tareas de Spark y de otras aplicaciones que comparten el cluster. Spark incluye su propio gestor de cluster, muy sencillo y que solo permite planificar tareas de Spark: *standalone*

Arquitectura de Spark



RDD: Resilient Distributed Datasets

- **Abstracción fundamental de Spark**. RDD: colección **no ordenada** (bag) de objetos distribuida en memoria entre los nodos del cluster. La colección está dividida en *particiones*, y cada una está en la memoria RAM de un nodo distinto del clúster
  - Resilient (resistente, adaptable): si un nodo falla, es posible **recalcular** las particiones de un RDD que estuviesen en ese nodo gracias al DAG de ejecución
  - Distributed (distribuido): los objetos de la colección están divididos en *particiones* que están distribuidas en la memoria principal de los nodos del cluster
  - Datasets: la colección representa un conjunto de datos que estamos procesando, transformando, agregando, etc.



RDD: Resilient Distributed Datasets

- **Inmutabilidad**. El contenido de un RDD no puede modificarse una vez creado. Lo que hacemos es aplicar transformaciones a los RDD para obtener otros nuevos
- Cuando aplicamos una transformación, se ejecuta en paralelo sobre todas las particiones del RDD, de manera transparente al programador.
  - Ejemplo: dado un RDD de números reales, para multiplicar cada elemento por 2, aplicamos una transformación que actúa en cada elemento y lo multiplica por 2.
  - Spark *lleva nuestro código de la transformación* (lo serializa y lo envía por la red) a cada uno de los nodos del cluster donde haya particiones de ese RDD, y lo ejecuta en ese nodo para que actúe en cada elemento de esa partición. Todo de manera transparente al programador.
  - *Los datos son el centro; no se mueven salvo que sea imprescindible*.
- *Partición*: unidad de datos mínima sobre la que se ejecuta una tarea de manera independiente al resto. Idealmente, tendría que haber al menos tantas como cores físicos (procesadores) disponibles.
- Originalmente, los programadores trabajaban a nivel de RDD. En Spark 1.6 se introdujeron los DataFrames y desde Spark 2.0, los propios creadores *recomiendan encarecidamente no utilizar los RDD sino siempre DataFrames con SparkSQL*

RDD: transformaciones y acciones

- *Transformación*: operación que se ejecuta sobre un RDD y devuelve un nuevo RDD, en el que sus elementos se han modificado de algún modo. Son **lazy**: no se ejecuta nada hasta que encontrar una *acción*. Simplemente se añade la transformación al grafo de ejecución (el DAG) que mantiene la trazabilidad y permite la *resiliency*
  - El DAG guarda toda la secuencia de transformaciones que se realizaron para obtener cada RDD concreto que se vaya creando en nuestro código.
- *Acción*: recibe un RDD y calcula un resultado (generalmente un tipo simple, enteros, doubles, etc) y lo devuelve al *driver* (programa principal, que corre en una máquina).
  - El resultado de la acción debe caber en la memoria de *un* solo nodo (el driver)
  - Desencadena instantáneamente el cálculo de toda la secuencia de transformaciones intermedias, y la materialización de los RDDs involucrados
  - Una vez materializado un RDD, se aplica la transformación que toque según indica el DAG para generar el siguiente RDD, y el anterior se libera (no permanece en la memoria RAM, salvo que se indique expresamente mediante el método *cache()*).
    - Un RDD *cacheado* permanece materializado y no es necesario recalcularlo
  - Si la transformación no implica *shuffle* (movimiento de datos entre nodos) se denomina *narrow*, y cada partición da lugar a otra en el mismo nodo. Lo contrario se denomina transformaciones *wide*.

RDD: transformaciones más frecuentes

- La transformación más típica y utilizada en RDDs es ***map***
- **map**(miFuncion): aplica la función que le pasamos por argumento a cada uno de los elementos del RDD. Devuelve otro RDD transformado, donde cada elemento es el resultado de aplicar *miFuncion* a un elemento del RDD original. La función *miFuncion* es una función programada por el usuario
- La función es ejecutada *en los* ***executors*** (cada uno sobre sus particiones)
- Es habitual que la función se pase como lambda, es decir, anónima.

Supongamos un fichero lineas.txt con esta estructura:

```
Jose Antonio;42;Española;Madrid
Giuliano;37;Italiana;Milán
François;16;Francesa;Niza
```

```
rddTuplas = spark.sparkContext\
            .textFile("/ruta/hdfs/lineas.txt")\  # RDD de strings
            .map(lambda linea: linea.split(";"))\ # RDD de listas
            .filter(lambda v: int(v[1]) >= 18)\ # RDD de listas
            .map(lambda v: (v[0], v[2]))\ # RDD de tuplas
            .filter(lambda tupla: tupla[1] == "Española")
print(rddTuplas.take(1))
```

RDD: transformaciones más frecuentes

- *En todas las operaciones que reciben una función, Spark la serializa y la envía por red a los nodos*
- map: recibe como parámetro una función que se ejecuta sobre cada uno de los elementos del RDD para transformarlo, devolviendo un nuevo RDD con los elementos transformados.
- flatMap: similar a la anterior, pero en este caso la función devuelve un vector de valores para cada elemento. En lugar de generar un RDD de vectores, los aplana para tener un RDD del tipo interior
- filter: recibe como parámetro una función que se aplicará sobre cada elemento del RDD y deberá devolver un valor booleano (*true* solo si ese elemento debe ser incluido en el nuevo RDD)
- sample: devuelve una muestra aleatoria del RDD del tamaño especificado como parámetro.
- union: devuelve un RDD que es la unión de dos RDD distintos pasados como parámetros.
- intersection: devuelve la intersección de los dos RDD, es decir, los elementos que están presentes en ambos.
- distinct: quita los elementos repetidos del RDD (retiene cada elemento una sola vez).
- Transformaciones específicas para un *PairRDD* :
  - groupByKey: cuando los elementos del RDD son tuplas (grupos de varios elementos ordenados), agrupa los elementos por la clave, considerando esta el primer elemento de la tupla.
  - reduceByKey: similar al anterior, pero se agregan los elementos para cada clave empleando la función especificada como parámetro. Esta debe recibir dos valores y devolver uno, y cumplir las propiedades conmutativa y asociativa.
  - sortByKey: ordena los elementos del RDD por clave.
  - join: combina dos RDD de tal modo que se junten los elementos que tienen la misma clave.

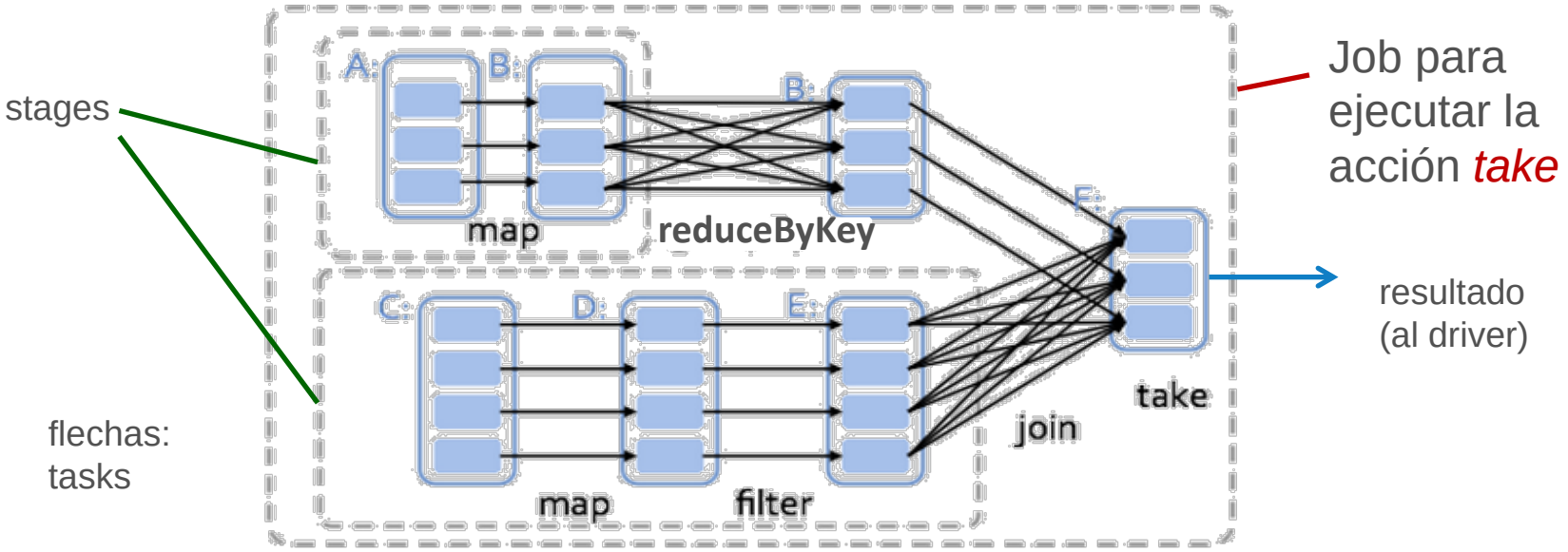
RDD: acciones más frecuentes

- Por definición, todas las *acciones* llevan resultados al *driver*, por lo que el resultado tiene que caber en la memoria!

- reduce: ejecuta una agregación de los datos empleando la función especificada como parámetro. Esta agregación se calcula sobre todos los datos, independientemente de que haya o no claves.
- collect: devuelve todos los elementos contenidos en el RDD como una colección del lenguaje (listas en python y R, arrays en Java y Scala).
  - IMPORTANTE: puede causar una excepción por memoria si la lista no cabe en la memoria RAM de la máquina donde está corriendo el driver. Usar sólo en casos muy controlados
- count: devuelve el número de elementos contenidos en el RDD.
- take: devuelve los *n* primeros elementos contenidos en el RDD. En general, no hay garantías de ordenación en un RDD salvo que se hayan empleado transformaciones como *sortByKey*.
- first: devuelve el primer elemento del RDD. Es equivalente a *take* cuando *n*=1.
- takeSample: devuelve *n* elementos aleatorios del RDD.
- takeOrdered: devuelve los *n* primeros elementos del RDD tras haber realizado una ordenación de todos los elementos contenidos en el mismo.
- countByKey: cuenta el número de elementos en el RDD para cada clave diferente.
- saveAsTextFile: guarda los contenidos del RDD en un fichero de texto.

Jobs, stages, tasks y el DAG

- Un *job* de Spark es todo el procesamiento necesario para llevar a cabo una *acción* del usuario (ej: df.count(), df.take(4), df.show(), df.read(…), df.write(…), etc).
- Cada *job* se divide en una serie de *stages* (etapas)
  - Stage: todo el procesamiento que puede llevarse a cabo sin mover datos entre nodos. Cada nodo hace exactamente el mismo procesamiento aplicado a diferentes particiones del mismo DataFrame que están procesando todos.
  - Cuando hay una operación que implica movimiento de datos (*shuffle*), finaliza un stage y se crea otro nuevo. Ej: df.join(…), df.groupBy(…).agg(…)
- Task (tarea): cada una de las transformaciones que forman una etapa
  - Es el procesamiento aplicado por un core físico (CPU) a una partición de datos.



…eso es todo por hoy…

:-)

Cualquier duda, consulta o comentario:

mensaje a través de la plataforma!

www.unir.net