Ingeniería para el Procesado Masivo de Datos

# Tema 3. Spark I

# Índice

# Esquema

## 3.1. Introducción y objetivos

En la última sección del tema anterior, expusimos las deficiencias del paradigma de programación distribuida MapReduce. En la presente lección, conoceremos Apache Spark, un nuevo *framework* de programación distribuida más intuitivo y rápido que nació para tratar de resolverlas.

Los objetivos que persigue este tema son:

- Entender por qué Spark es superior a MapReduce atendiendo a su diseño.
- Identificar los módulos que componen Spark y el propósito de cada uno.
- Conocer la arquitectura y el funcionamiento interno de Spark.
- Practicar con algunas funciones típicas de procesamiento de datos con Spark.

## 3.2. Apache Spark

En el año 2009, surgió en Berkeley, EE. UU., una nueva propuesta para paliar estas deficiencias, que, poco a poco, ha ido imponiéndose hasta reemplazar completamente a Hadoop (más concretamente, a MapReduce).

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

Analicemos la definición:

- Es un *motor* de cálculo, esto es, un *framework* de propósito general orientado a resolver cualquier tipo de problema y con el que se puede implementar cualquier algoritmo.
- *Unificado*: el motor es único e independiente de cómo utilicemos Spark:
- Desde la API de DataFrames (para lenguaje R, Python, Java y Scala).
- Desde herramientas externas que lanzan consultas SQL contra Spark (Hive, herramientas de *business intelligence* conectadas a Spark como Tableau, PowerBI, Microstrategy, QlikSense, etc.).
- Desde una instrucción de la API de programación que recibe una consulta SQL como *string,* tan compleja como sea necesario.
- Cualquiera de las opciones anteriores se traduce a un grafo de tareas al que Spark aplica optimizaciones de código automáticamente; por tanto, todos (API, aplicaciones BI, SQL, etc.) se benefician de ellas.
- *En memoria*: todos los cálculos se llevan a cabo en memoria y solo se escriben

resultados a disco (parciales o finales) cuando el usuario lo indica explícitamente (operación de guardado) o cuando la operación indicada por el usuario requiere forzosamente realizar movimiento de datos entre nodos (*shuffle,* el cual se lleva a cabo desde el disco duro del emisor al disco duro del nodo receptor). Además, el movimiento de datos (y las operaciones de acceso a disco que conlleva) solo se produce cuando es irremediable, pero no de manera obligatoria, como ocurría en MapReduce. Esto consigue un rendimiento muy superior en tareas iterativas (varias pasadas sobre los mismos datos, p.ej., algoritmos de *machine learning)*.

## 3.3. Componentes de Spark

Apache Spark consiste en una API (bibliotecas de programación) distribuida, mucho más intuitiva que MapReduce. Al igual que este, abstrae todos los detalles de comunicación de red entre las máquinas del clúster, pero, además, opera de manera similar a las consultas SQL tradicionales, como si los datos fuesen tablas (distribuidas). Esto la hace fácil de usar y de aprender.

Spark ofrece distintas API para cuatro lenguajes: Java (muy tediosa), Scala (la más utilizada en la actualidad por los ingenieros de datos para aplicaciones en producción), Python (paquete de Python PySpark, muy usado por científicos de datos, con una API casi idéntica a la de Scala que, en realidad, no es más que una capa adicional que termina llamando al código de Scala) y R (paquete de R SparkR, con una API muy diferente al resto, más cercana a los comandos típicos de R). En este tema, usaremos PySpark para todos los ejemplos.

La figura 1 muestra los principales componentes de Spark.

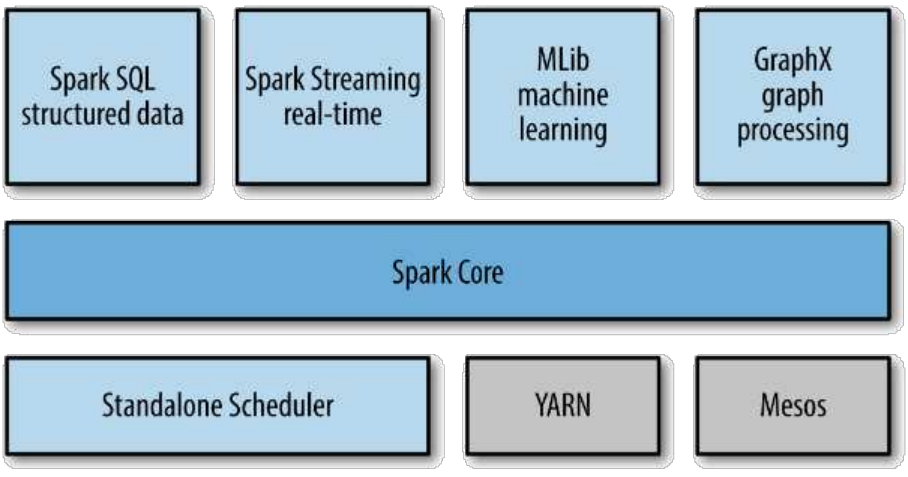



Figura 1. Módulos que componen Apache Spark. Fuente: Chambers y Zaharia 2018.

De estos, el principal es el módulo Spark Core. En él se encuentran las estructuras

de datos fundamentales de Spark, tales como los RDD (Resilient Distributed Dataset), de los que hablaremos más adelante, y las operaciones que se puede llevar a cabo con él. En última instancia, todas las operaciones que se pueden hacer con Spark, independientemente del módulo utilizado o la API donde se encuentren, se traducen automáticamente por parte de Spark a operaciones sobre RDD, que son las únicas que sabe ejecutar el motor.

Los tres módulos inferiores representan tres gestores de recursos de un clúster sobre los que puede ejecutarse Spark. Un gestor de recursos se encarga de asignar máquinas, CPU y memoria principal a Spark, de forma que disponga de un determinado número de nodos y de que, en cada uno, existan suficientes recursos para ejecutar la aplicación que hemos implementado con Spark.

El resto de módulos cumplen las siguientes funciones:

- Spark SQL y API estructurada es una API con una serie de funciones para manejar tablas de datos distribuidas, estructuradas en columnas con nombre y tipo, denominadas DataFrames. Un DataFrame proporcionan un nivel de abstracción adicional sobre un RDD, al cual recubre. Este módulo incluye también una función para ejecutar sentencias SQL sobre las mismas estructuras de datos distribuidas.

- Spark Streaming es el módulo para operar de manera distribuida sobre datos en tiempo real, según los vamos recibiendo (*stream* de datos). A partir de Spark 2.0, este módulo ha sido reemplazado por **Spark Structured Streaming,** que simplifica el procesamiento de flujos de datos en tiempo real.

- Spark MLlib contiene implementaciones distribuidas de algoritmos de Machine Learning.

- Spark GraphX es el módulo de procesamiento de grafos, representados mediante RDD. Contiene algunos algoritmos de camino mínimo y similares. Actualmente, **este módulo ha quedado obsoleto** y ha sido reemplazado *de facto* por el paquete GraphFrames, que representa un grafo como una pareja de DataFrames, con los

nodos y los arcos respectivamente. Empezó como un proyecto separado de Spark y, finalmente, ha sido integrado en las versiones más recientes.

## 3.4. Arquitectura de Spark

La figura 2 muestra la arquitectura de una aplicación que utiliza Spark al ejecutarse en un clúster. No debemos olvidar que, al escribir una aplicación en Spark, en realidad, estamos escribiendo una aplicación **secuencial** (no paralela) utilizando la biblioteca de Spark para el lenguaje en el que estemos programando (Java, Scala, Python o R). Cuando ejecutamos el código de este programa (proceso que se denomina *driver*), lo hacemos sobre una máquina concreta, que puede ser interna o externa al clúster. Por ejemplo, podríamos escribir un programa en Spark y ejecutarlo en nuestro ordenador portátil, y, desde aquí, el programa driver podría conectarse a un clúster de Spark remoto.

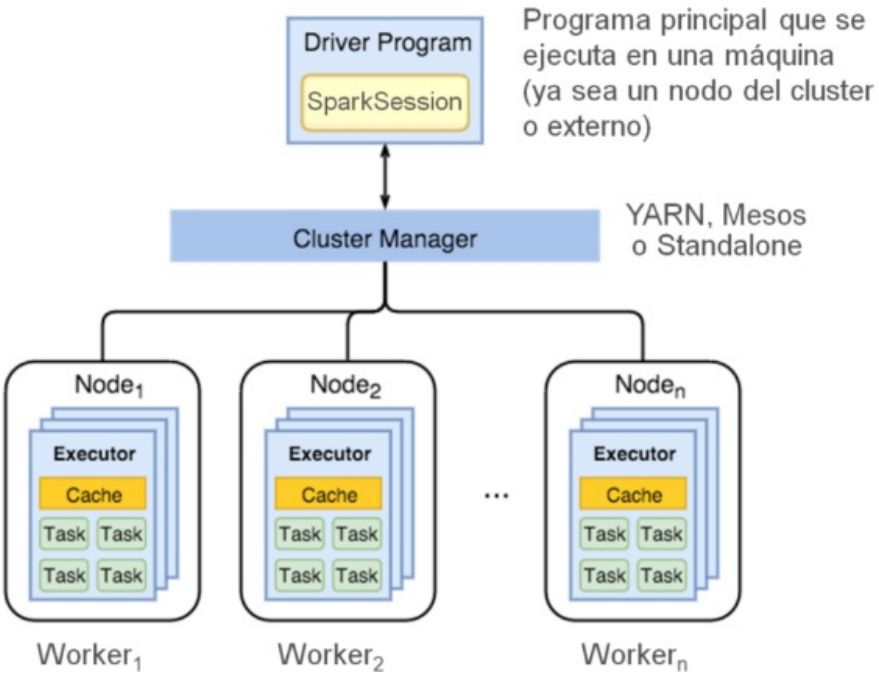



Figura 2. Arquitectura de una aplicación en Spark al ejecutarse en un clúster. Fuente: http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

En el transcurso de la ejecución, normalmente necesitaremos crear un objeto denominado `sparkSession`, de la clase `sparkSession` de Spark. En el momento de crear

este objeto, hay que indicar dónde (en qué dirección IP y en qué puerto) existe un clúster de Spark configurado. Ciertas aplicaciones, como, por ejemplo, Jupyter Notebook (cuando está configurado para Spark) o la línea de comandos (intérprete) de Spark, tanto en Scala (*spark-shell*) como en Python (*pyspark-shell*), ya nos dan un objeto `sparkSession` creado al arrancar la aplicación. No obstante, al iniciar el *shell*, debemos pasarle la configuración relativa al clúster para que la `sparkSession` se cree correctamente.

De lo contrario, se estará ejecutando en local, sin conexión con un clúster. De esta manera, la `sparkSession` establece comunicación con el gestor de clúster para poder enviar tareas a los *workers* (nodos del clúster disponibles para Spark). A partir de este momento, podemos utilizar dichos recursos para las sentencias que requieran ejecución distribuida.

Es importante recalcar esto último: **la ejecución de un programa en Spark es secuencial**, en una sola máquina, tal como sería cualquier otro programa en el lenguaje elegido, **excepto cuando el flujo de programa llega a ciertas funciones específicas de Spark que desencadenan ejecución distribuida,** que son la mayoría (pero no todas) de las que forman parte de la API de Spark. Muchas se aplican sobre el objeto `sparkSession` y otras sobre estructuras de datos distribuidas que hayamos ido creando en el programa, como, por ejemplo, RDD o DataFrames de Spark.

En el momento de crear el objeto `sparkSession`, hemos indicado una configuración con el número de nodos, la memoria RAM y el número de cores físicos que necesitamos reservar en cada nodo. En un nodo, estos recursos constituyen lo que se denomina un *executor*: un proceso de la JVM (máquina virtual de Java) que se ejecuta en el nodo y que ocupa los recursos indicados (cores, RAM, disco duro).

El proceso ejecutor es creado por el gestor de clúster cuando arranca nuestra aplicación de Spark y muere cuando la aplicación finaliza (ya sea con éxito o por

alguna condición imprevista que provoca que toda la aplicación termine abruptamente). Cada ejecutor queda preparado para ejecutar *tareas* de Spark, que es la unidad mínima de ejecución de trabajos. Cada tarea requiere un *core* libre para ejecutarse, por lo que, si un *executor* tiene reservados cuatro *cores*, podrá ejecutar cuatro tareas en paralelo. Detallaremos esto más adelante. **Cada uno de los nodos del clúster en los que se crean ejecutores se denomina *worker*.**

## 3.5. Resilient distributed datasets (RDD)

Los RDD constituyen la **abstracción fundamental de Spark**.

Un RDD es una colección no ordenada (*bag*) de objetos, distribuida en la memoria RAM de los nodos del clúster.

La colección está dividida en particiones, cada una de las cuales está en la memoria RAM de un nodo distinto del clúster. Al desgranar el nombre, tenemos que:

- ***Resilient*(‘resistente’, ‘adaptable’):** es posible reconstruir un RDD que estaba en memoria, a pesar de que una de las máquinas falle, gracias al DAG de ejecución (*directed acyclic graph*, el grafo de ejecución), que veremos más adelante.
- ***Distributed* (‘distribuido’):** los objetos de la colección están divididos en particiones que están distribuidas en la memoria principal de los nodos del clúster. La colección no está ordenada, por lo que no se puede acceder mediante una posición a objetos individuales.
- ***Dataset*:** la colección representa un conjunto de datos que estamos procesando de forma paralela y distribuida, para transformarlos, calcular agregaciones, etc.

La figura 3 representa tres RDD diferentes, distribuidos en la memoria RAM de un clúster de cuatro nodos. No todos los RDD presentan el mismo número de particiones. En la figura, uno de los RDD tiene solo dos, otro tiene tres y otro, cuatro. La idea es similar al almacenamiento de los ficheros en HDFS, donde están distribuidos entre los discos duros de los nodos, con la diferencia de que, en este caso, los RDD están distribuidos en la memoria RAM de los nodos y, además, **no hay replicación de cada partición**.

Si un nodo falla, es posible reconstruir las particiones que estuvieran en ese

momento en su memoria principal gracias al DAG, que mantiene la traza de cómo se construyeron. El DAG es otro mecanismo que proporciona robustez sin necesidad de replicar las particiones de un RDD. Es preciso indicar también que, a pesar de que la figura 3 muestra una sola partición de cada RDD en cada nodo, lo habitual es que, en la memoria de un mismo nodo, haya numerosas particiones de un mismo RDD (de hecho, es normal que los RDD que se van calculando tengan decenas o cientos de particiones).

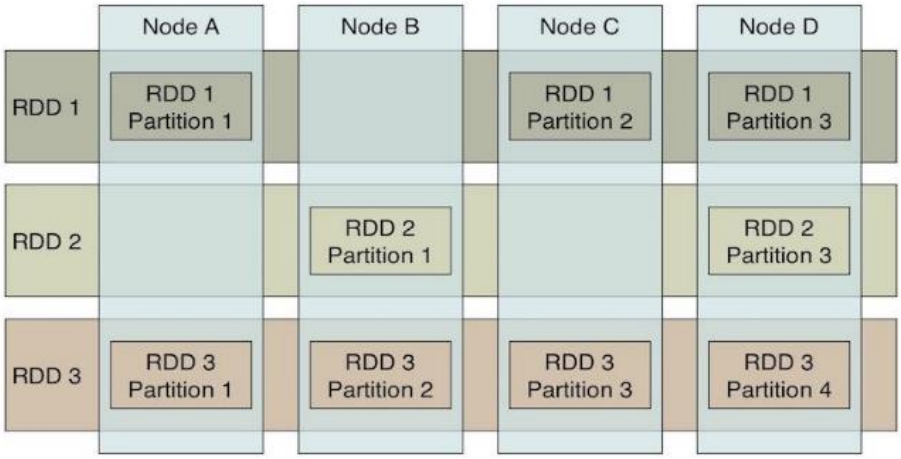



Figura 3. Representación de tres RDD en un clúster de Spark con cuatro *workers.*

En relación con los RDD, cabe destacar dos conceptos:

- **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, pero los datos del RDD existente no se alteran. La idea de la ejecución es que, cuando aplicamos una transformación, se ejecuta en paralelo sobre todas las particiones del RDD, de manera transparente al programador, para dar lugar a un nuevo RDD, cuyas particiones son el resultado de aplicar la transformación sobre cada una de las particiones del original.

> **Ejemplo:** dado un RDD de números reales, para multiplicar cada elemento por dos, aplicamos una transformación que actúa en cada elemento y lo

multiplica por dos. Spark lleva nuestro código de la transformación (lo serializa y lo envía por la red) a cada uno de los nodos del clúster donde haya particiones de ese RDD y lo ejecuta en ellos para que actúe en cada elemento de esa partición. Todo de manera transparente al programador.

Una vez más, **los datos son el centro, lo más importante; no se mueven salvo que sea imprescindible**. Ciertos tipos de transformaciones no requieren movimiento de datos, pero otros sí, como veremos después.

- **Partición:** subconjunto de los objetos de un RDD que están presentes en un mismo nodo. **Es la unidad de datos mínima sobre la que se ejecuta una tarea de transformación de manera independiente al resto de particiones.** Idealmente, tendría que haber, al menos, tantas particiones como *cores* físicos (procesadores) disponibles en nuestro clúster. De esta manera, nos aseguramos de que todos los *cores* estarán ocupados, incluso en el caso de que nada más que nuestra aplicación estuviese empleando ese clúster. Lo habitual es que haya muchas particiones de un mismo RDD en cada nodo, en un número muy superior al de procesadores existentes en el ellos. Se recomienda que cada RDD esté dividido en un número de particiones que sea entre el doble y el triple que el número de procesadores del clúster.

Originalmente, los programadores de Spark trabajaban con RDD. Sin embargo, en Spark 1.6, se introdujeron los DataFrames, que definiremos más adelante. Desde Spark 2.0, los propios creadores **recomiendan encarecidamente no utilizar los RDD, sino siempre DataFrames y su API correspondiente**. Aparte de la facilidad de uso, el motivo fundamental es que los DataFrames están sujetos a importantes optimizaciones automáticas de código por parte del analizador de Spark, llamado Catalyst. Los RDD no están sujetos a estas optimizaciones y la ejecución es

sensiblemente más lenta. Todos los ejemplos los presentaremos con PySpark.

## 3.6. Transformaciones y acciones

Podemos realizar dos tipos de operaciones cuando usamos la API de Spark:

- **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* (perezosas): no se ejecuta nada hasta que Spark encuentra una acción. Mientras tanto, Spark simplemente 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.

  - 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. Hay que recordar que una operación *shuffle* implica una escritura previa de los datos en el disco duro local del nodo emisor y después en el disco local del nodo receptor, de manera transparente al programador.

- **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 tipo de dato devuelto al *driver* no es un RDD, sino un tipo nativo del lenguaje que estemos utilizando (Java/Scala/Python/R).

  - IMPORTANTE: el resultado de la acción debe caber en la memoria de la máquina donde se está ejecutando el proceso *driver.*

  - Una acción desencadena instantáneamente el cálculo de toda la secuencia de transformaciones intermedias y la materialización de los RDD 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 en la RAM de los nodos y no es necesario recalcularlo después de que se haya materializado la primera vez.

Por defecto, el punto de partida del DAG son operaciones de lectura de datos, bien desde una fuente de datos como, por ejemplo, HDFS o el sistema Amazon S3, bien desde alguna base de datos (distribuida o no) o similar. Si ningún RDD intermedio ha sido cacheado, cualquier operación que haga referencia a un RDD exigirá reconstruir toda la secuencia de transformaciones previas a dicho RDD, empezando por la lectura de los datos, excepto que alguno de los RDD intermedios haya sido cacheado. Esto hace que la secuencia empiece en el RDD cacheado y no haya que remontarse al origen de datos.

La utilización del DAG sirve para poder reconstruir cualquier RDD de una secuencia de transformaciones, tanto si la necesidad de materializar el RDD se debe a que hacemos referencia a él varias veces a lo largo de nuestro código, como si obedece a que, durante el procesamiento, falla algún nodo y el contenido de su memoria RAM se pierde. Gracias al DAG, es posible reconstruir cualquier partición concreta de cualquier RDD y volver a lanzar la secuencia de transformaciones en otros nodos que sí estén activos.

Existen ciertas operaciones de la API de Spark que **no son transformaciones ni acciones**, como, por ejemplo, `cache()`, sino que sirven para configurar, habilitar o deshabilitar ciertos comportamientos, o para obtener características relativas a la distribución física de un RDD [consultar el número de particiones que tiene, consultar si está cacheado, consultar el esquema (nombres y tipos de las columnas) de un DataFrame, etc.].

## Transformaciones más habituales con RDD

Recordemos que, para todas las operaciones que reciben una función, Spark serializa el código de la función y la envía por red a los nodos, donde es ejecutada.

# Ideas clave

- map : recibe como parámetro una función, que se ejecuta sobre cada uno de los elementos del RDD para transformarlos, y devuelve 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 una función que se aplicará sobre cada elemento del RDD y que deberá devolver un valor booleano (*true*, solo si ese elemento debe ser incluido en el nuevo RDD). Devuelve otro RDD con los elementos que han devuelto *true*.
- sample : devuelve una muestra aleatoria del RDD del tamaño especificado como parámetro.
- union : devuelve un RDD con la unión de dos RDD 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 (retiene cada elemento una sola vez).

Transformaciones específicas para un PairRDD, es decir, RDD de pares (clave, valor):

- groupByKey : cuando los elementos del RDD son tuplas (grupos de varios elementos ordenados), agrupa los elementos por la clave y considera esta como 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.

## Acciones más habituales en RDD

Por definición, todas las acciones llevan resultados al *driver*, por lo que estos tienen que caber en la memoria del proceso *driver*.

- 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). Puede causar una **excepción** por memoria si la lista no cabe en la memoria RAM de la máquina donde está corriendo el *driver.* Se debe usar solo 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.

**Ejemplo de código PySpark con RDD**

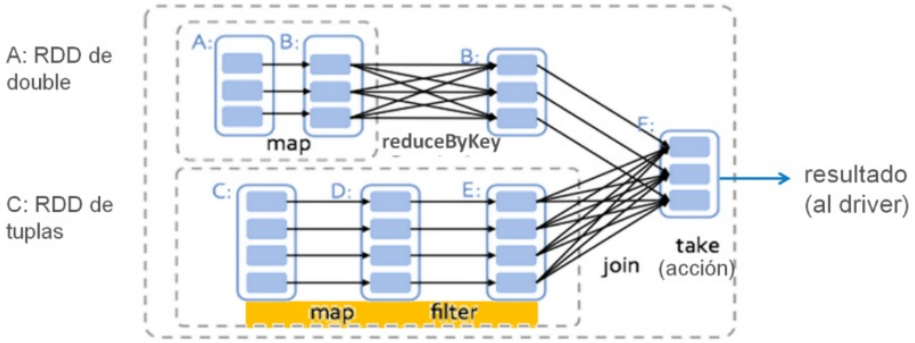



Figura 4. Ejemplo de transformaciones y de la acción take mediante el uso de RDD.

```
func_multiplicar = lambda x: (x, 3*x) # función que devuelve una tupla

A = sc.parallelize([5.0, 3.2, 1.1, -2.4, # distribuimos la lista como

8.9, 4.4, 3.7, 9.1], 3) # un RDD de 3 particiones

B = A.map(func_multiplicar)

B = B.reduceByKey(lambda v1, v2: v1+v2)

C = sc.parallelize([(5.0, 1.0), (1.1, -3)], 4) # lista a RDD de 4 part

D = C.map(lambda tuple: (tuple[0], 2*tuple[1]))

E = D.filter(lambda tuple: tuple[1] > 1)

F = E.join(B)

resultado = F.take(3)# ¡ahora es cuando se desencadena el cálculo!
```

> Como se observa, ciertas operaciones como `join` y `reduceByKey` asumen un RDD de (clave, valor), lo cual no es tan frecuente y todavía se asemeja a MapReduce en la manera de pensar (clave, valor). Para simplificar, existen los DataFrames, similares a tablas.

En el ejemplo anterior, hemos usado la variable `sc`, referida a `SparkContext`, el objeto que, en versiones anteriores, efectuaba la conexión con el gestor de clúster.

Actualmente, el objeto *sparkSession* envuelve (e incluye) a un `SparkContext`. Lo habitual es usar la `sparkSession` para leer datos de una fuente y crear desde ellos un DataFrame. No obstante, para poner crear un RDD (distribuido) a partir de una lista no distribuida del lenguaje, aún se necesita el objeto `sc`. En el código anterior, estamos creando, en primer lugar, un RDD llamado A, que la figura 5 representa con tres particiones, a partir de una lista de números reales. Por eso, el RDD resultante es un RDD de números reales.

Hemos definido una función `func_multiplicar`, que recibe un número y devuelve una tupla formada por el número y el resultado de multiplicarlo por 3. Dicha función la hemos aplicado a cada elemento de A mediante el método *map* de Spark. Este método es una **transformación** que aplica a cada elemento del RDD nuestra función y devuelve un nuevo RDD con el resultado. Para ello, **serializa el código de nuestra función** (en este caso, la función `func_multiplicar`) y lo envía por la red a los nodos, para que, en ellos (en aquellos nodos que contengan alguna partición del RDD A), se ejecute dicha función sobre los elementos de las particiones del RDD que estén presentes. Hemos llamado B al RDD resultante, que, en este caso, será un RDD de tuplas de dos elementos de tipo *double*, que es lo que devolvía `func_multiplicar`.

Nótese que, para llevar a cabo la transformación *map*, no es necesario movimiento de datos, ya que la función se aplica elemento a elemento sobre los que haya en el nodo y no necesita de otros elementos para calcular el resultado. Se dice que *map* es una transformación ***narrow*** (estrecha), a diferencia de otras transformaciones que forzosamente requieren movimiento de datos para poder calcular el resultado, llamadas transformaciones ***broad***.

Un RDD de tuplas de dos elementos se considera, a ojos de Spark, un PairRDD, es decir, un RDD de (clave, valor). Los RDD de (clave, valor) admiten operaciones adicionales, además de las estándar de cualquier RDD. Una de ellas es `reduceByKey`, que agrupa elementos del RDD que tengan el mismo valor de clave y agrega entre sí sus valores, empleando la función que le pasemos como argumento. Esta función

requiere movimiento de datos entre nodos, ya que existirán tuplas que compartan la misma clave, pero estén en particiones distintas, que, además, posiblemente, también estarán en nodos diferentes. El resultado de esta transformación lo hemos vuelto a asignar a la variable B.

Por otro lado, hemos creado otro RDD en la variable C, que la figura 5 muestra con cuatro particiones, como el resultado de paralelizar una lista de tuplas de números reales. Por tanto, C ya es, desde el comienzo, un PairRDD. Le hemos aplicado, a continuación, una transformación *map*. Dado que C contiene tuplas, la función que aplicamos tiene que saber que el argumento que recibe es una tupla. En nuestro caso, a partir de cada tupla, se devuelve otra cuyo primer elemento es igual y cuyo segundo elemento es el resultado de multiplicar por 2 el segundo elemento original. La sintaxis de *lambda* de Python simplemente sirve para indicar que estamos creando una función anónima, es decir, una función convencional, pero que no necesitamos invocarla desde fuera. Solo la usamos para pasarla como argumento a un método (que espera recibir una función como argumento, tal como le ocurre a *map* de Spark); por eso, no necesitamos darle nombre, aunque podríamos haberlo hecho creando una función convencional con nombre, como se suele hacer con *def* en Python.

El RDD resultante de esta transformación *map* lo almacenamos en la variable D, que también es un PairRDD. Sobre él aplicamos una nueva transformación *filter*, que, al igual que *map*, actúa sobre cada elemento del RDD sin necesitar ningún otro proveniente de otra partición ni de otro nodo para calcular el resultado. De hecho, lo que *filter* lleva a cabo es un filtrado, de manera que el RDD resultante solo contendrá aquellos elementos del RDD original que cumplan cierta condición. La función pasada como argumento a *filter* debe ser capaz de recibir un elemento del RDD (en este caso, una tupla de dos elementos) y siempre ha de devolver un booleano, que indica si ese elemento tiene que formar parte del resultado (*true*) o no (*false*).

A continuación, hemos invocado una transformación *join*, que se aplica a un PairRDD y recibe como argumento otro PairRDD. El resultado es un nuevo RDD que contiene tuplas jerárquicas tales que la clave es común entre una tupla de uno de los RDD y una tupla del otro, y el valor está formado por una tupla con el valor que tenía esa clave en un RDD y el valor que tenía en el otro. El RDD resultante de la operación *join* aplicada a B y E lo almacenamos en la variable F.

Por último, hemos llamado al método *take* sobre el RDD F. Este método es una acción que lleva el resultado al *driver*. Esto implica que el resultado debe caber en él. En este caso, no supone un problema, ya que *take(n)* coge n elementos del RDD y los envía al *driver*, y solo hemos solicitado tres elementos. Además, y lo que es más importante, al ser una acción, desencadena la realización de todas las transformaciones anteriores que estaban pendientes. De hecho, la ejecución de cada una de las líneas de código anteriores a *take* simplemente provocaba que se añadiesen fases al grafo de ejecución (DAG) de Spark, pero no se materializaba ninguna. Se puede comprobar porque la ejecución en Python devuelve inmediatamente el control al intérprete de Python, sin emplear tiempo en llevarla a cabo. El resultado de *take* ya no es un RDD, sino una estructura de datos del lenguaje que se esté manejando. En este caso, es una lista de Python formada por tres elementos del RDD F, es decir, una lista de tres tuplas, que es lo que contiene F.

## 3.7. Jobs, stages y tasks

Un *job* (trabajo) de Spark es todo el procesamiento necesario para llevar a cabo una acción del usuario.

Por ejemplo: *df.count(), df.take(4), df.show(), df.read(…), df.write(…)* , etc. Cada *job* se divide en una serie de *stages* (etapas).

Un *stage* es todo el procesamiento que puede llevarse a cabo sin mover datos entre nodos.

Cada nodo hace exactamente un procesamiento idéntico, aplicado a diferentes particiones del mismo DataFrame que están procesando todos. Cuando en nuestro código invocamos una operación que implica movimiento de datos (*shuffle*), finaliza un *stage* y se crea otro nuevo. Ej: *df.join(…), df.groupBy(…).agg(…)*.

Una *task* (tarea) es cada una de las transformaciones que forman una etapa. Es la unidad mínima de trabajo de Spark. Más formalmente:

Una tarea de Spark es el procesamiento aplicado por un *core* físico (CPU) a una partición de un RDD.

La siguiente figura representa la división en tareas, etapas y trabajos para el ejemplo de código contenido en el anexo.

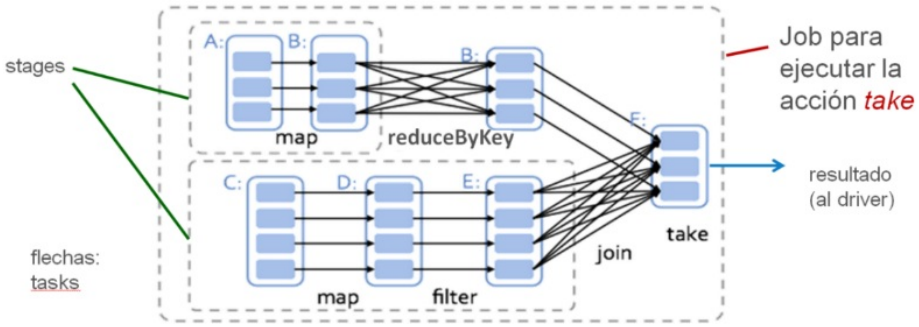



Figura 5. Representación de *jobs, stages* y *tasks* en Spark.

## 3.8. Ejemplo completo con RDD

Con el objetivo de comparar las características de la programación mediante RDD con respecto a la facilidad de uso que proporciona el componente que veremos en el siguiente tema (API estructurada y DataFrames), así como de destacar las diferencias con entornos como Apache Hive, que estudiaremos más adelante, vamos a ver, a continuación, un ejemplo completo resuelto en el que usaremos RDD. Posteriormente, resolveremos este mismo ejemplo en el siguiente tema, así como en el tema donde veamos Apache Hive, de forma que se puedan apreciar las similitudes, divergencias, ventajas e inconvenientes de cada uno de los entornos.

Para ello, vamos a usar los ficheros simplificados flights.csv y airport-codes.csv, almacenados ambos en HDFS bajo el directorio /user/data, y cuyas primeras líneas son las siguientes:

```
"year","month","day","origin","dest"
2014,1,1,"PDX","ANC"
2014,1,1,"SEA","CLT"
…

ident,name,iso_country,iata_code
LELT,Lillo,ES,
LEMD,Adolfo Suárez Madrid–Barajas Airport,ES,MAD
…
```

El objetivo del ejemplo es contar cuántos vuelos reciben los diferentes destinos (dest) que aparecen en el fichero flights.csv. Además, como no conocemos bien los códigos de los aeropuertos de llegada, nos gustaría ver esta agregación del número de vuelos que recibe cada aeropuerto como nombre completo del aeropuerto y número de vuelos que recibe. Para esto, nos ayudaremos de la información que contiene el fichero arirport-codes.csv, donde se relaciona el código del aeropuerto con su nombre.

```
from pyspark.sql import SparkSession

# En primer lugar, puesto que vamos a trabajar con RDD,
# necesitamos tener acceso al SparkContext.
# Para ello, primero obtenemos la SparkSession,
sparkSession = SparkSession.builder\
               .appName("demo_rdd")\
               .getOrCreate()
# y accedemos a uno de sus atributos, que es el SparkContext.
sparkContext = sparkSession.sparkContext

# Lo siguiente es leer el fichero flights.csv
# Cada línea se va a leer como un string completo:
flights_lines = sparkContext\
                .textFile("hdfs://<ip:port>/user/data/flights.csv")

# Dado que el fichero tiene cabecera (header), necesitamos
# ejecutar estas líneas para ignorarla (la librería de RDD
# no proporciona una opción directa para hacer esto):
header_line = flights_lines.first()
flights_lines = flights_lines\
                .filter(lambda lines: lines != header_line)

# Para comprobar qué contenido hemos leído, ejecutamos la acción take.
# ¡OJO! No hemos usado la acción collect, porque, en ese caso,
# se obtendrían TODOS los registros y podríamos provocar un error por
# quedarnos sin memoria (un fichero almacenado en HDFS probablemente
# esté almacenado ahí, de forma distribuida, porque no cabe
# en un sistema de ficheros local, mucho menos en la memoria de un
# único ordenador):
flights_lines.take(5)
```

Obtenemos los siguientes registros. Fijémonos en que cada uno de los cinco registros pedidos es una cadena de texto (*string*) completa, es decir, no están divididos por los diferentes campos que lo componen. La biblioteca de RDD no proporciona ningún método que permita interpretar la estructura del fichero, sino que sencillamente lee cada línea y la guarda como un registro del RDD. Además, cabe destacar que lo que observamos aquí no es el RDD. Cuando ejecutamos una acción como *collect* o *take* en PySpark, nos devuelve una lista de Python, tal y como se puede apreciar por la delimitación con corchetes de los cinco elementos (*strings*) de la lista.

```
['2014,1,1,PDX","ANC"',
 '2014,1,1,"SEA","CLT"',
 '2014,1,1,"PDX","IAH"',
 '2014,1,1,"PDX","CLT"',
 '2014,1,1,"SEA","ANC"']
```

```
# Por tanto, tenemos que dividir los diferentes
# campos a mano, aplicando la función split de string.
# Se separa cada línea utilizando el carácter '…',
# para cada registro del RDD. Usaremos la transformación map
# para aplicar una función a cada registro del RDD:
flights_tuples_rdd = flights_lines.map(lambda line: line.split(','))

# Puesto que es una transformación, todavía no existe el RDD
# flights_tuples_rdd. No se materializará hasta que se ejecute una
# acción sobre el RDD:
flights_tuples_rdd.take(5)
# En este punto, al ejecutar la acción take, es cuando se
# materializa tanto flights_tuples_rdd como todos los RDD
# previos necesarios para obtenerlo.
```

Obtenemos la siguiente salida. Ahora sí vemos cada línea separada en sus diferentes campos (año, mes, día, origen, destino) y delimitada en forma de lista de Python. Es decir, tenemos una lista con cinco registros de los RDD, cada uno de los cuales es una lista de *strings* correspondientes a los componentes de cada línea del fichero.

```
[['2014', '1', '1', '"PDX"', '"ANC"'],
 ['2014', '1', '1', '"SEA"', '"CLT"'],
 ['2014', '1', '1', '"PDX"', '"IAH"'],
 ['2014', '1', '1', '"PDX"', '"CLT"'],
 ['2014', '1', '1', '"SEA"', '"ANC"']]
```

Cabe recordar que, cada vez que ejecutamos una acción, como el último `take`, hay que volver a materializar todos los RDD necesarios desde la lectura de fichero, hasta obtener el RDD sobre el que se ejecuta la acción. Esto puede ser muy costoso en términos de procesamiento. Por tanto, si hay algún RDD sobre el que se necesite aplicar muchas acciones, puede ser interesante guardarlo en la memoria caché de

los *workers*. De esta forma, los RDD guardados en caché no hay que materializarlos en cada acción, sino que son accesibles directamente. Esto supone un compromiso entre la memoria y el procesamiento: los *workers* tienen una cantidad de memoria limitada, por lo que no se pueden cachear todos los RDD que queramos para evitar quedarnos sin memoria en los *workers*. Solo es conveniente cachear aquellos que necesiten materializarse repetidamente.

```
flights_tuples_rdd.cache()
```

```
# Como se puede observar en los resultados previos,
# todos los componentes que representan
# las diferentes líneas de cada sublista
# son de tipo string. Sin embargo, el año,
# el mes y el día estarían mejor representados por un tipo entero.
# Aunque esto no tiene impacto en este ejemplo, vamos a hacer
# un cabio de tipo para mostrar cómo se haría:
flights_tuples_format_rdd = flights_tuples_rdd\
                            .map(lambda fields_list :
                                    (int(fields_list[0]),
                                     int(fields_list[1]),
                                     int(fields_list[2]),
                                     fields_list[3],
                                     fields_list[4]))

print(flights_tuples_format_rdd.take(5))
```

En los siguientes resultados, podemos ver que ahora año, mes y día son enteros, mientras que origen y destino siguen siendo *strings*. Esta forma de proceder es poco intuitiva, ya que, si tuviéramos una lista con más campos, sería necesario saber qué posición ocupa cada uno para poderle aplicar correctamente el tipo deseado (no se puede hacer por nombre).

```
[(2014, 1, 1, '"PDX"', '"ANC"'),
 (2014, 1, 1, '"SEA"', '"CLT"'),
 (2014, 1, 1, '"PDX"', '"IAH"'),
 (2014, 1, 1, '"PDX"', '"CLT"'),
 (2014, 1, 1, '"SEA"', '"ANC"')]
```

```
# Ahora queremos agrupar los destinos y contar cuántos
# vuelos recibe cada uno de ellos. Es decir, cuántas veces
# aparece ese destino en el fichero (y ahora RDD) de vuelos.

# Para hacer agrupaciones y agregaciones, necesitamos
# trabajar con PairRDD, es decir, RDD que sean tuplas de
# dos elementos: el primero de ellos será la clave por la que se
# quiera agrupar y el segundo, un valor asociado a la clave.
# En nuestro caso, la clave para cada registro va a ser el destino
# y el valor, 1, para contabilizar como 1 cada vez que
# aparezca dicho destino (y que, al sumarlos, tengamos
# el número de veces que aparece):
flights_dest_counter_rdd = flights_tuples_format_rdd\
          .map(lambda flight_tuple : (flight_tuple[4], 1))
flights_dest_counter_rdd.take(5)
```

Obtenemos un RDD con registros como los que se muestran a continuación: son registros de un *pairRDD*, es decir, un RDD que contiene tuplas de dos elementos, donde el primero será interpretado por Spark como la clave por la que hacer agrupaciones.

```
[('"ANC"', 1),
 ('"CLT"', 1),
 ('"IAH"', 1),
 ('"CLT"', 1),
 ('"ANC"', 1)]
```

```
# Con objeto de tener una idea de cómo funciona la agrupación
# de RDD, vamos a asociar todos los valores por clave (destino, en
# nuestro caso) y los vamos a mostrar en una lista. Para ello,
# agrupamos por clave y pedimos que se cree una lista con los valores # de
cada clave:
flights_dest_grouped_rdd_toshow =
flights_dest_counter_rdd.groupByKey().mapValues(list)

# Mostramos solo un registro del resultado:
print(flights_dest_grouped_rdd_toshow.take(3))
```

Cada registro resultante de la agrupación será una tupla que contiene la clave por la

que se agrupó, así como una lista con todos los valores asociados a esa clave (en nuestro caso, tantos 1 como veces aparece el destino en el fichero):

```
[('"ANC"', [1, 1, 1, 1, 1, ...]),
 ('"CLT"', [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...],
 ('"ORD"', [1, 1, ...])]
```

```
# Ahora sí: nuestro objetivo era contar cuántos vuelos
# tenían como destino cada uno de los destinos
# en el fichero flights.csv
# Para ello, podemos utilizar el pairRDD previo, agrupar por destino y
# sumar los valores de cada clave:
flights_dest_total_rdd = flights_dest_counter_rdd\
        .groupByKey().\
        .map(lambda tuple_dest : (tuple_dest[0], sum(tuple_dest[1])))

# O también podemos contar cuántos valores tiene asociados cada clave #
(longitud, len) de la lista:
flights_dest_total_rdd = flights_dest_counter_rdd\
                         .groupByKey().\
                         .mapValues(len)
print(flights_dest_total_rdd.take(5))
```

Como podemos ver, el resultado en ambos casos es el mismo:

```
[('"ANC"', 7149),
 ('"CLT"', 1224),
 ('"ORD"', 7021),
 ('"PHX"', 8660),
 ('"SLC"', 5976)]

[('"ANC"', 7149),
 ('"CLT"', 1224),
 ('"ORD"', 7021),
 ('"PHX"', 8660),
 ('"SLC"', 5976)]
```

```
# Para saber a qué aeropuerto corresponde cada destino,
# vamos a usar el fichero airport-codes.csv.
# Procedemos igual que con el fichero flights.csv:
airports_lines = sparkContext\
                   .textFile("hdfs://<ip:port> /user/data/airport-
```

```
codes.csv")

header_line = airports_lines.first()
airports_lines = airports_lines\
                   .filter(lambda lines: lines != header_line)
airports_tuples_rdd = airports_lines\
                         .map(lambda line: line.split(','))
airports_tuples_format_rdd = airports_tuples_rdd\
          .map(lambda fields_list : (fields_list[3], fields_list[1]))
print(airports_tuples_format_rdd.take(2))
```

Vemos un par de aeropuertos entre los leídos. Cabe destacar que hemos creado otro pairRDD (obligatorio para el join que queremos hacer), para el que nos hemos quedado solo con dos de los campos de cada línea, y que les hemos dado la vuelta para que el código del aeropuerto sea el primero de la tupla y, por tanto, la clave. Fijémonos en que el formato del código no es igual que en el fichero flights.csv, ya que, en este último, el código está entrecomillado (la lectura de RDD no sabe interpretar que eso son *strings*, por lo que no es necesario el entrecomillado).

```
[('ORD', ' Chicago O'Hare International Airport'),
 ('SFO', ' San Francisco International Airport')]
```

```
# Vamos a quitar las comillas de los destinos en el RDD de vuelos
# para poder hacer el join (si no, los códigos de destino de
# ambos RDD no serán iguales)
# con tuple_flight[0][1:4]. Nos quedamos con
# los caracteres 1-3, ambos inclusive.
# En definitiva, quitamos las comillas (posiciones 0 y 4):
flights_dest_total_rdd = flights_dest_total_rdd\
                     .map(lambda tuple_flight : (tuple_flight[0][1:4],
                                                 tuple_flight[1]))

# Y, ahora sí, podemos hacer un join usando la clave de ambos
# pairRDD:
dest_count_airports_rdd = flights_dest_total_rdd\
                            .join(airports_tuples_format_rdd)
print(dest_count_airports_rdd.take(5))
```

El resultado del join con RDD son pairRDD, donde la clave es el código del

aeropuerto y el valor, otra tupla que contiene el valor del primer pairRDD (el número de vuelos) y el del segundo (el nombre del aeropuerto).

```
[('ORD', (7021, "Chicago O'Hare International Airport")),
 ('SFO', (12809, 'San Francisco International Airport')),
 ('IAD', (1344, 'Washington Dulles International Airport')),
 ('KOA', (734, 'Ellison Onizuka Kona International At Keahole Airport')),
 ('JNU', (1282, 'Juneau International Airport'))]
```

```
# Si lo queremos mostrar como nombre y número de vuelos,
# solo resta elegir esos campos, pero hay que tener en cuenta el
# anidamiento de tuplas (de nuevo, poco intuitivo):
dest_count_airports_rdd = dest_count_airports_rdd\
                      .map(lambda tuple_result : (tuple_result[1][1],
                                                  tuple_result[1][0]))
print(dest_count_airports_rdd.take(5))
```

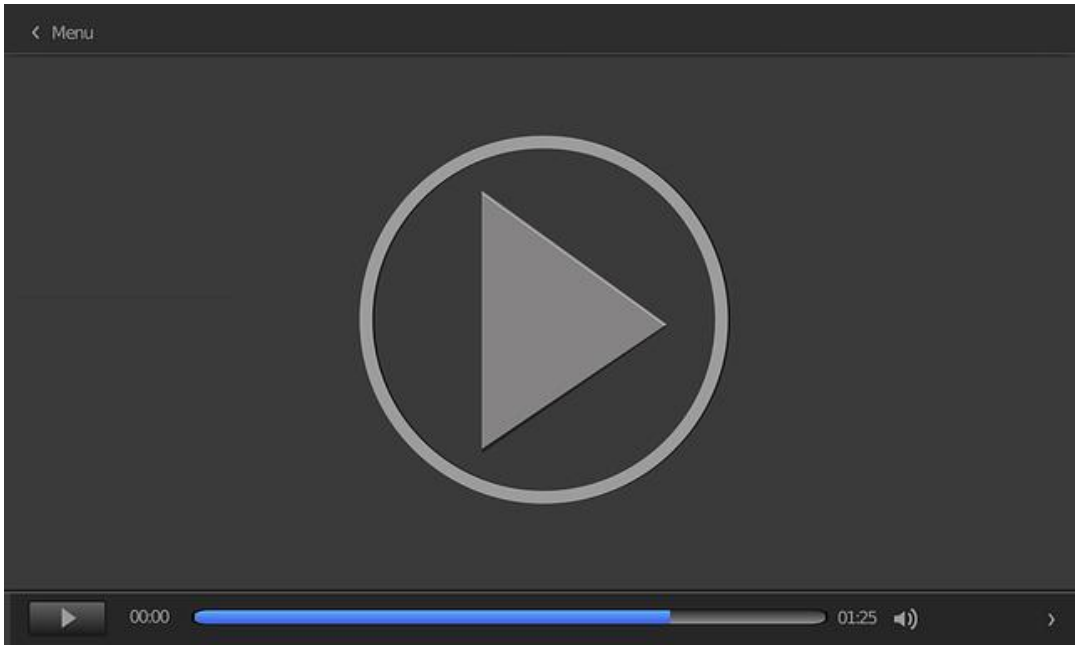
Por fin, obtenemos el resultado deseado:

```
[("Chicago O'Hare International Airport", 7021),
 ('San Francisco International Airport', 12809),
 ('Washington Dulles International Airport', 1344),
 ('Ellison Onizuka Kona International At Keahole Airport', 734),
 ('Juneau International Airport', 1282)]
```

Este ejemplo bien sirve para demostrar que el manejo de RDD, aunque proporciona gran potencial en el manejo de grandes cantidades de datos, no es demasiado intuitivo, ya que es necesario tener siempre muy presente la estructura interna del RDD y el tipo de datos que contiene, además de la siguiente restricción: necesita pairRDD para poder hacer operaciones de agregación y operaciones entre RDD. En el siguiente capítulo, veremos qué soluciones proporciona Spark para facilitar este tipo de tareas y hacerlas mucho más intuitivas con una API estructurada y Spark SQL.

Para finalizar, puedes ver este vídeo, en el profundizaremos en la utilización de los RDD de Spark desde la herramienta JupyterLab de Dataproc.

# Ideas clave




Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=bf539479-7a1e-4b21-9fea-ac70011af767

Vídeo. Spark RDD.

## 3.9. Referencias bibliográficas

Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.

## Documentación oficial de Apache Spark

Apache Spark. Página web oficial: https://spark.apache.org/docs/latest/

Documentación *online*, detallada y de muy buena calidad sobre el sistema de computación Apache Spark.

## Hadoop: the end of an era

Grishchenko, A. (2019, 23 de marzo). Hadoop: the end of an era [entrada de blog]. *Distributed Systems Architecture*. https://0x0fff.com/hadoop-the-end-of-an-era/

Hadoop es uno de los *frameworks* más importantes para el *big data*, cuyo propósito es almacenar grandes cantidades de datos y permitir consultas sobre estos, que se ofrecerán con un bajo tiempo de respuesta. Nació como iniciativa de Apache para dar soporte al paradigma de programación MapReduce, que fue inicialmente publicado por Google. En esta entrada de blog, se propone una interesante reflexión sobre su uso en la actualidad.

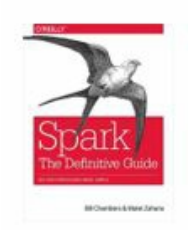
## DATA + AI Summit

DATA + AI Summit. Página web oficial: https://databricks.com/sparkaisummit

Sitio web del congreso más famoso de Spark a nivel mundial, del que tienen lugar también versiones más reducidas en cada continente. Está organizado por Databricks, empresa que soporta el desarrollo de Spark.

## Spark: the definitive guide

Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.




Guía detallada de Spark, en su versión más actualizada. Contiene numerosos ejemplos y muestra exhaustivamente todas las capacidades de Spark.

## High performance Spark

Karau, H. y Warren, R. (2017). *High performance Spark*. O'Reilly.


Manual avanzado sobre cómo escribir código optimizado en Spark.

# Test

1. ¿Cuál es la principal fortaleza de Spark?

   A. Opera en memoria principal, lo que hace que los cálculos sean mucho más rápidos.

   B. Nunca da lugar a movimiento de datos entre máquinas (*shuffle*).

   C. Las respuestas A y B son correctas.

   D. Las respuestas A y B son incorrectas.

2. ¿Qué tipo de procesos se benefician especialmente de Spark?

   A. Los procesos en modo *batch*, como, por ejemplo, una consulta SQL.

   B. Los proceso aplicados a datos no demasiado grandes.

   C. Los algoritmos de aprendizaje automático que dan varias pasadas sobre los mismos datos.

   D. Las respuestas A, B y C son correctas.

3. ¿Cuál es la estructura de datos fundamental en Spark?

   A. RDD.

   B. DataFrame.

   C. SparkSession.

   D. SparkContext

4. En una operación de Spark en la que sea necesario movimiento de datos…

   A. Siempre debemos escribirlos primero en el disco local del nodo emisor.

   B. No hay acceso al disco local, puesto que Spark opera siempre en memoria.

   C. Spark nunca provoca movimiento de datos, a diferencia de MapReduce.

   D. Las respuestas A, B y C son incorrectas.

**5.** Elige la respuesta correcta: Cuando se ejecuta una transformación en Spark sobre un RDD…

A. Se crea inmediatamente un RDD con el resultado de la transformación.

B. Se modifica inmediatamente el RDD con el resultado de la transformación.

C. Se añade la transformación al DAG, que creará un RDD con el resultado de la transformación cuando se materialice el RDD resultante.

D. Se añade la transformación al DAG, que modificará el RDD original con el resultado de la transformación cuando se materialice el RDD resultante.

**6.** Elige la respuesta correcta: La acción *collect* de Spark…

A. No existe como acción; es una transformación.

B. Aplica una función a cada fila del RDD de entrada y devuelve otro RDD.

C. Lleva todo el contenido del RDD al *driver* y podría provocar una excepción.

D. Lleva algunos registros del RDD al *driver*.

**7.** Elige la respuesta incorrecta: Un PairRDD…

A. Es un tipo de RDD que permite realizar tareas de agregación y *joins*.

B. Es un tipo de RDD que contiene una tupla con un número variable de componentes.

C. Es un tipo de RDD cuyo primer componente se considera la clave y el segundo, el valor.

D. Se define como cualquier otro RDD, pero con un formato concreto.

**8.** ¿Qué es un *executor* de Spark?

A. Cada uno de los nodos del clúster de Spark.

B. Un proceso creado en los nodos del clúster, preparado para recibir trabajos de Spark.

C. Un nodo concreto del clúster que orquesta los trabajos ejecutados en él.

D. Ninguna de las definiciones anteriores es correcta.

**9.** La acción *map* de Spark…

A. No existe como acción; es una transformación.

B. Aplica una función a cada fila del RDD de entrada y devuelve otro RDD.

C. Lleva todo el contenido del RDD al *driver* y podría provocar una excepción.

D. Lleva ciertos registros del RDD al *driver*.

**10.** Cuando Spark ejecuta una acción…

A. Se materializan en la memoria RAM de los *workers* todos los RDD intermedios necesarios para calcular el resultado de la acción y después se liberan todos.

B. Se añade la acción al DAG y no hace nada en ese momento.

C. Se materializan los RDD intermedios necesarios que no estuviesen ya materializados, se calcula el resultado de la acción y se liberan los no cacheados.

D. Ninguna de las respuestas anteriores es correcta.