Procesamiento Distribuido para Big Data

Tema 3: Apache Spark

Autor: Dr. Pablo J.
Villacorta Iglesias

Actualizado Marzo 2020

INTRODUCCIÓN

En el tema anterior vimos cómo MapReduce permite procesar los datos almacenados de manera distribuida en el sistema de archivos (por ejemplo HDFS) de un cluster de ordenadores. MapReduce permite resolver todo tipo de problemas, pese a que no siempre es sencillo pensar la solución en términos de operaciones *map* y *reduce* encadenadas. Por ello se dice que es de propósito general, a diferencia de, por ejemplo, el lenguaje SQL que está orientado específicamente a realizar consultas sobre los datos. Implementar en SQL algoritmos tales como un método de ordenación, o el algoritmo de Dijkstra para encontrar caminos mínimos en un grafo no sería posible.

Recordemos que MapReduce tiene varias desventajas:

- El resultado de la fase *map* (tuplas) se escribe en el disco duro de cada nodo, como resultado intermedio. Los accesos a un disco duro son un orden de magnitud más lentos que los accesos a la memoria principal (memoria RAM) de cada nodo, por lo que estamos penalizando el rendimiento debido a cómo está estructurado el propio framework.

- Después de la fase *map*, hay tráfico de red obligatoriamente (movimiento de datos, conocido como *shuffle*). De hecho, el movimiento de datos forma parte como una etapa obligada en el framework de MapReduce. A pesar de que las redes que conectan los nodos de un cluster son habitualmente muy rápidas cuando se trata de clusters construidos *ad-hoc* para un propósito (por ejemplo, el cluster Mare Nostrum), no tiene porqué ser así cuando se trata de máquinas alquiladas a un proveedor de servicios cloud. En cualquier caso, el movimiento de datos de un nodo a otro constituye un segundo cuello de botella y es una operación que debe evitarse cuando utilizamos cualquier framework de programación distribuida.

- Por otro lado y relacionado con el punto anterior, para mover datos se necesita, en primer lugar, escribirlos temporalmente en el disco duro de la máquina origen, enviarlos por la red y escribirlos temporalmente en el disco duro de la máquina destino (el *shuffle* siempre va de disco duro a disco duro) para, finalmente, pasarlos a la memoria principal de dicho nodo.

- Estos dos inconvenientes se acentúan especialmente cuando el algoritmo que queremos implementar sobre un cluster es de tipo iterativo, es decir, requiere varias pasadas sobre los mismos datos para ir convergiendo. En el campo del

Machine Learning, que es uno de los que más se puede beneficiar del procesamiento de grandes conjuntos de datos para extraer conocimiento, este tipo de algoritmos son el estándar, y por ello se comprobó que sus implementaciones con MapReduce no eran eficientes debido a la propia concepción del framework.

- Finalmente, enfocar cualquier problema en términos de operaciones *map* y *reduce* encadenadas no siempre es fácil ni intuitivo para un desarrollador, y la solución resultante puede ser difícil de mantener si no está bien documentada.

En el año 2009 surgió en Berkeley, EEUU, 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 clusters de ordenadores

---

Analizando 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 de cálculo es único e independiente de cómo utilicemos Spark:
  - Desde la API de DataFrames (para los lenguajes 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, etc)
  - Desde una instrucción de la API de programación que recibe una consulta SQL como string, tan compleja como sea necesario
  - …todo se traduce a un grafo de tareas al que Spark aplica optimizaciones de código automáticamente, y 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 del disco duro del emisor al disco duro del nodo receptor). Además, el movimiento de datos solo se produce cuando es irremediable, y no de manera obligatoria. Esto consigue un rendimiento hasta 100 veces superior que MapReduce en tareas iterativas (varias pasadas sobre los mismos datos, como tareas de machine learning). La medición de tiempos en una regresión logística arrojó un factor de mejora de x100.

Breve historia de Spark

La Fig. 1 muestra un gráfico con la evolución temporal de Spark. Fue concebido como parte de la tesis doctoral de Matei Zaharia, un investigador en el grupo AMPLab de la Univ. de Berkeley, en California. Tras la publicación en 2010 del primer artículo sobre Spark, y la liberación desde el principio del código fuente, se vio su potencia y la ganancia que se podría llegar a conseguir sustituyendo el procesamiento anterior en MapReduce por el procesamiento con este nuevo framework.

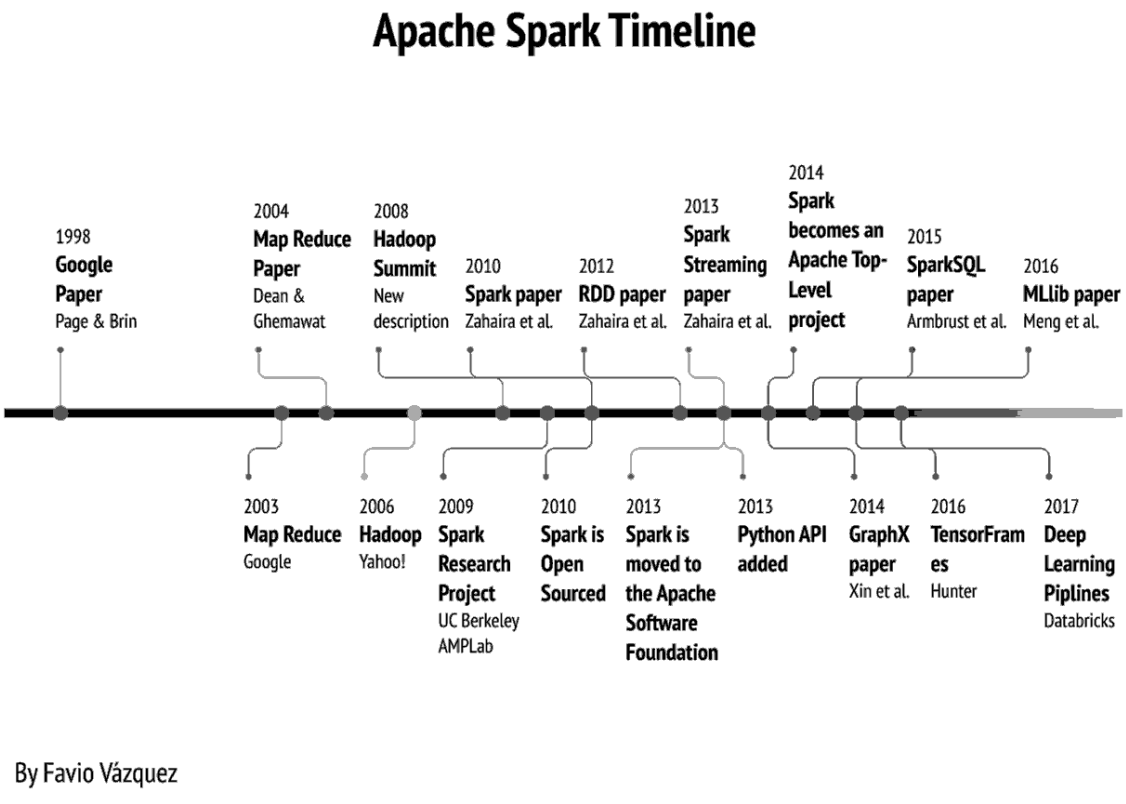


Fig. 1. Evolución de Apache Spark a lo largo del tiempo

Poco después del nacimiento de Spark, su creador Matehi Zaharia fundó la empresa Databricks, dedicada a dar servicios en torno a Spark, tales como como su propia plataforma Big Data y servicios cloud. Poco a poco se fue completando Spark con módulos específicos que aprovechaban esta tecnología para ofrecer API de mayor nivel de abstracción para tareas concretas, como procesamiento de datos en *streaming*, o la utilización de estructuras de datos distribuidas de más alto nivel y más similares a las tablas de una base de datos relacional, además de un módulo específico que implementa modelos de Machine Learning que pueden entrenarse con datos distribuidos.

COMPONENTES DE SPARK

Apache Spark consiste en una API (bibliotecas) orientada al programador, **mucho** más intuitiva que MapReduce. Al igual que éste, abstrae todos los detalles de comunicación de red y hardware, 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 APIs para cuatro lenguajes:

- Java: muy tediosa de usar (sintaxis muy verbosa) pero útil para aplicaciones legacy existentes.

- Scala: es la API más utilizada para aplicaciones en producción. Scala es un lenguaje funcional que se compila sobre la JVM (la compilación genera bytecode de Java), muy compacto y cómodo de utilizar con Spark. La mayor parte del código del propio Spark es lenguaje Scala.

- Python (paquete de Python *pyspark*)*:* API casi idéntica a la de Scala, salvo peculiaridades del lenguaje. En realidad es un wrapper (capa adicional) sobre la implementación en Scala, que es la verdaderamente distribuida. No obstante, Spark también permite ejecutar código python en los nodos del cluster.

- R (paquete de R *SparkR*): es una API muy diferente al resto, más cercana a los comandos típicos de R. Últimamente, el paquete *sparklyr* (creado por la empresa RStudio) es más útil, ofrece una mejor integración y más avanzada, y su sintaxis es más intuitiva.

La Fig. 2, tomada de [1], muestra los principales componentes de Spark.

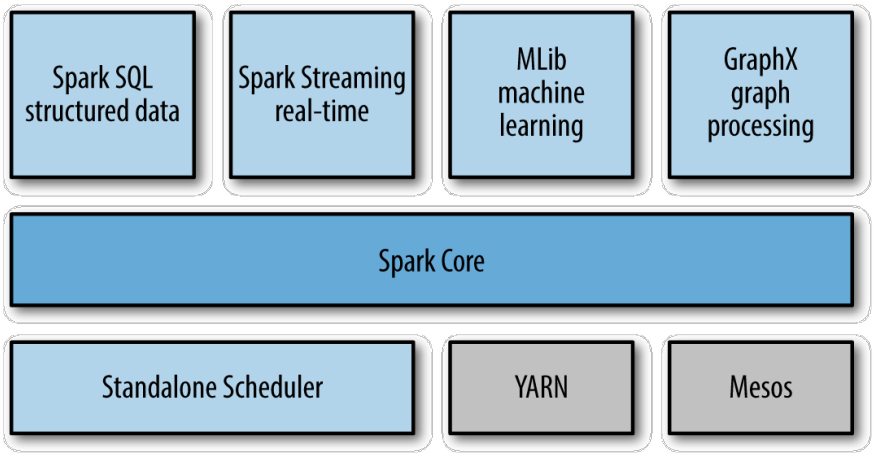


Fig 2. Módulos que componen Apache Spark

De estos, el principal es el módulo *Spark core*. En él se encuentran las abstracciones fundamentales de Spark, tales como el RDD (Resilient Distributed Dataset), del 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 a operaciones sobre RDDs, que son las únicas que Spark es capaz de ejecutar.

Los tres módulos inferiores representan tres gestores de recursos de un cluster sobre los que puede ejecutarse Spark. Un gestor de recursos se encarga de asignar máquinas, CPUs y memoria principal a Spark, de forma que disponga de un determinado número de nodos y en cada uno existan suficientes recursos para ejecutar la aplicación que hemos implementado con Spark. YARN y Mesos no forman parte de Spark pero son compatibles con Spark. El *Standalone Scheduler* es el gestor de recursos más sencillo, y viene incluido con Spark. No permite un control de grano muy fino de los recursos pero es suficiente para casos de uso simples.

El resto de módulos cumplen las siguientes funciones:

- Spark SQL 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. Estas sentencias son analizadas y convertidas en un plan lógico que en última instancia se traduce a operaciones sobre RDDs al igual que cualquier otra operación de Spark.

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

- Spark MLlib contiene implementaciones distribuidas de algoritmos de Machine Learning. Aunque el módulo conserva este nombre, el paquete actual de la API se llama Spark ML y simplifica el uso de estos algoritmos puesto que actúan sobre DataFrames.

- Spark GraphX es el módulo de procesamiento de grafos, representados mediante RDDs. 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. GraphFrames se desarrolló como una biblioteca externa a Spark pero la próxima versión de Spark ya la incluirá por defecto.

ARQUITECTURA DE SPARK

La siguiente figura muestra la arquitectura de una aplicación que utiliza Spark al ejecutarse en un cluster.

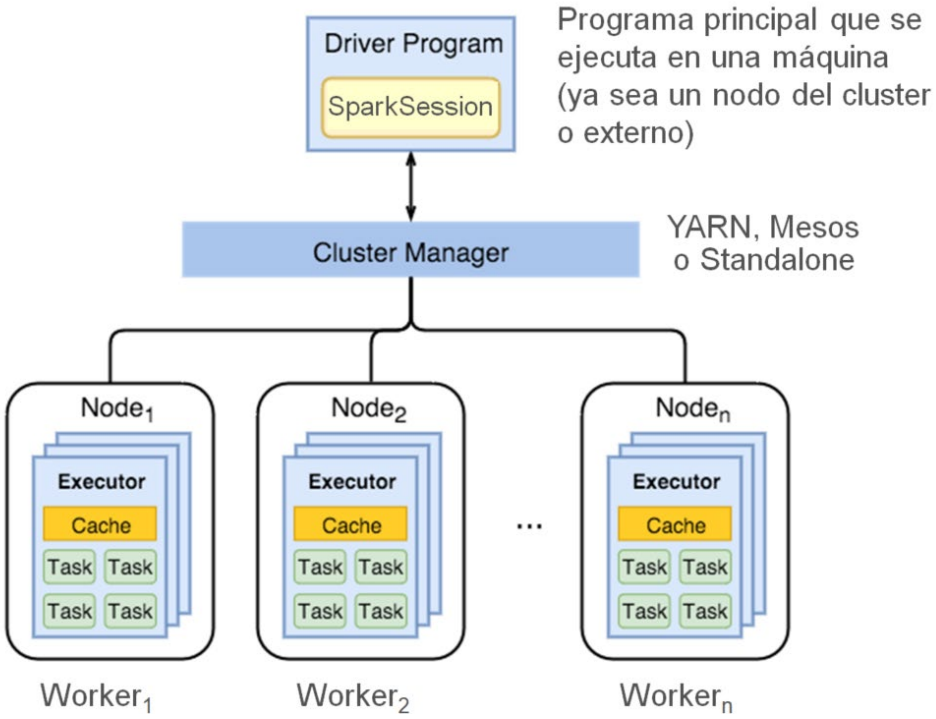


Fig. 3. Arquitectura de una aplicación en Spark

Cuando escribimos 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, el proceso en el cual se ejecuta se denomina *Driver*, y se ejecuta sobre una máquina concreta, que puede incluso ser externa al cluster. Por ejemplo, podríamos escribir un programa en Spark y ejecutarlo en nuestro ordenador portátil, y desde aquí, el programa Driver podría conectarse a un cluster de Spark remoto, lo cual se denomina ejecución en modo *client* (cliente). También existe la opción de que el programa Driver sea enviado al cluster y se ejecute dentro de uno de los nodos del cluster, lo cual se denomina ejecución en modo cluster.

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 cluster de Spark configurado o, más concretamente, dónde existe un gestor de cluster que nos pueda dar acceso a los recursos (nodos, memoria, CPUs) de un cluster. Ciertas aplicaciones, como por ejemplo Jupyter Notebook cuando está configurado para Spark, o la línea de comandos de Spark tanto en Scala (spark-shell) como en python (pyspark-shell) ya crean un objeto sparkSession por nosotros al arrancar la aplicación, aunque debemos pasarle como argumentos a la aplicación la configuración relativa al cluster para que la sparkSession se cree correctamente. De esta manera, la sparkSession establece comunicación con el gestor de cluster. 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 desencadenen ejecución distribuida, que son la mayoría (pero no todas) las que forman la API de Spark. Muchas de estas funciones se aplican sobre el objeto sparkSession, y otras se aplican sobre estructuras de datos distribuidas que hayamos ido creando en el programa, como por ejemplo RDDs 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 número de cores físicos que necesitamos reservar en cada nodo. Los recursos concedidos por el gestor de cluster específico no siempre coinciden exactamente con los solicitados (depende de las políticas y del número de recursos que ya estuvieran ocupados en ese momento). En cualquier caso, la solicitud los deja reservados para que, llegado el momento en el que se vaya a ejecutar una operación distribuida de nuestro programa, puedan ser utilizados.

Estos recursos en un nodo 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 cluster 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 require un core libre para ejecutarse, por lo que si un Executor tiene reservados 4 cores, podrá ejecutar 4 tareas en paralelo. Detallaremos esto más adelante. Cada uno de los nodos del cluster en los que se crean ejecutores se denomina *Worker*.

RDD: RESILIENT DISTRIBUTED DATASETS

Los RDD constituyen la **abstracción fundamental de Spark**[1].

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

La colección está dividida en *particiones*, y cada una está en la memoria RAM de un nodo distinto del clúster. Desgranando 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). Comentaremos esto en detalle más adelante.

- Distributed (distribuido): los objetos de la colección están divididos en *particiones* que están distribuidas en la memoria principal de los nodos del cluster. La colección no está ordenada, por lo que no se puede acceder mediante una posición a objetos individuales.

[1] http://spark.apache.org/docs/latest/rdd-programming-guide.html

- Dataset: la colección representa un conjunto de datos que estamos procesando de forma paralela y distribuida, transformándola, calculando agregaciones, etc.

La Fig. 4 representa tres RDDs diferentes distribuidos en la memoria RAM de un cluster de 4 nodos. No todos los RDD tienen el mismo número de particiones. En la figura, uno de los RDD tiene solo 2 particiones, otro tiene 3 y otro tiene 4. 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 la memoria principal de ese nodo 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. Nótese también que, a pesar de que la Fig. 4 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).

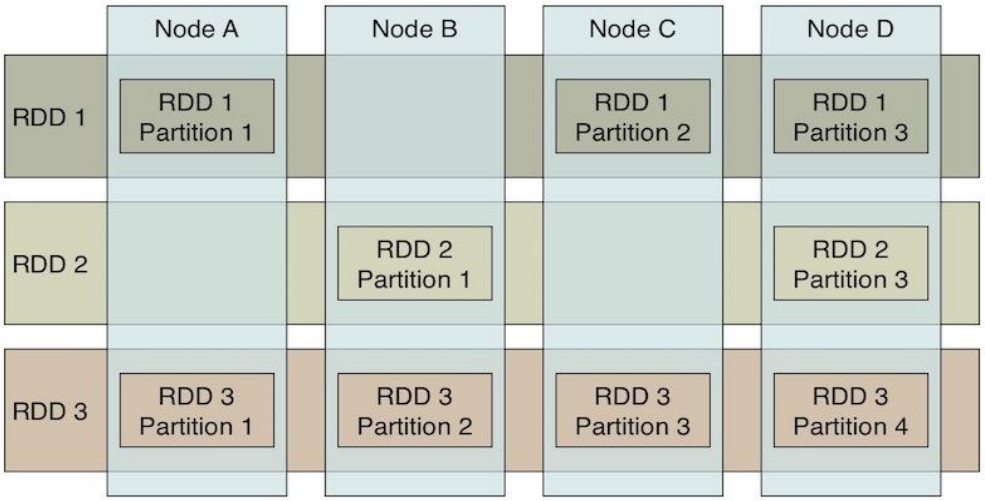


Fig. 4. Representación de tres RDDs en un cluster de Spark con cuatro workers.

Inmutabilidad de un RDD: 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 como resultado, 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 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.

  - 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 cluster. De esta manera nos aseguramos de que todos los cores estarán ocupados incluso en el caso de que todos estuviesen libres y nadie más que nuestra aplicación estuviese ocupando ese cluster.

Originalmente, los programadores de Spark trabajaban a nivel de RDD. En Spark 1.6 se introdujeron los DataFrames, que definiremos más adelante, y desde Spark 2.0 los propios creadores *recomiendan encarecidamente no utilizar los RDD sino siempre DataFrames con SparkSQL*. Todos los ejemplos los presentaremos con PySpark.

Transformaciones y acciones

Existen dos tipos de operaciones que podemos llevar a cabo 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.

- **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).

  - 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 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 en la RAM y no es necesario recalcularlo

después de que se haya materializado la primera vez cuando esté involucrado en una secuencia de transformaciones.

Por defecto, el punto de partida del DAG son operaciones de lectura de datos desde una fuente de datos como por ejemplo HDFS, o bien el sistema Amazon S3, o 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 en 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.

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).

Veamos un ejemplo de código Python que utiliza PySpark (API de Spark para Python).

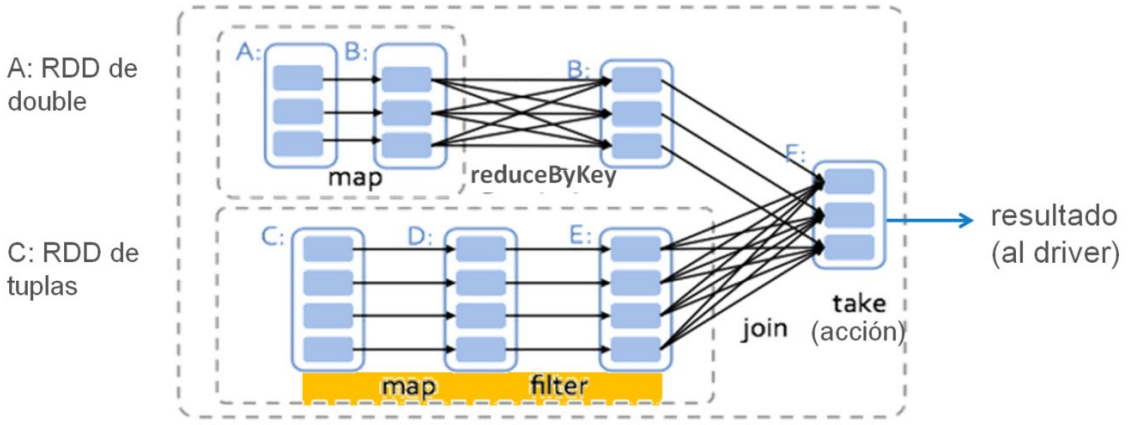


Fig. 5. Ejemplo de transformaciones y la acción take usando RDDs

```
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!

# 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 la manera de operar: DataFrames al rescate!
```

En el ejemplo anterior hemos usado la variable *sc*, referida a SparkContext, el objeto que en versiones anteriores efectuaba la conexión de nuestro driver (por ejemplo nuestro Notebook de Jupyter) con el gestor de cluster. Actualmente, lo que se utiliza es el objeto **spark** es una 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 Fig. 5 representa con 3 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 en ese nodo. 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 elementos 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, estén en nodos distintos. 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 Fig. 5 muestra con 4 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, solamente la usamos para pasarla como argumento a un método (que espera recibir una función como argumento, tal como le ocurre a *map* de Spark) y 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 debe devolver un booleano que indica si ese elemento debe 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 el driver. 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 3 elementos. Además, y lo que es más importante, al ser una acción, desencadena la ejecución 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 ejecutaba 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 3 elementos del RDD F, es decir, una lista de 3 tuplas que es lo que contiene F.

Transformaciones más habituales en RDDs

Nota general: 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.

Acciones más habituales en RDDs

Por definición, todas las *acciones* llevan resultados al *driver*, por lo que el resultado tiene 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).
    - IMPORTANTE: 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 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.

SPARK SQL

Manejar RDDs resulta tedioso cuando los tipos de datos que contienen empiezan a complicarse, ya que en todo momento necesitamos saber exactamente la estructura del

dato incluso cuando son tuplas jerárquicas (tuplas en las que algún campo es a su vez una tupla o lista). Sería más conveniente poder manejar los RDDs como si fuesen tablas de datos, estructuradas en filas y columnas, lo cual aportaría un mayor nivel de abstracción y más facilidad de uso. Esto es lo que nos proporciona Spark SQL[2] y la estructura de datos fundamental, los DataFrames de Spark.

DataFrame: tabla de datos distribuida en la RAM de los nodos, formada por filas y columnas con nombre y tipo (incluidos tipos complejos), similar a una tabla en una base de datos relacional

Internamente, un DataFrame no es más que un RDD de objetos de tipo *Row* de Spark, que representan a una fila como un vector cuyos campos tienen nombre y tipo predefinido. Cada DataFrame envuelve a un RDD.

A partir de la versión 2 de Spark, los creadores aconsejan encarecidamente utilizar exclusivamente DataFrames y no usar los RDD sin prácticamente ninguna excepción. 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, que genera código de manera automática cuando va a ejecutar el DAG con nuestras operaciones. Catalyst optimiza el plan lógico del DAG, es decir, el orden en que se efectúan las operaciones, para obtener el mejor rendimiento, como veremos más adelante. Los RDD no están sujetos a estas optimizaciones y por tanto, la ejecución es sensiblemente más lenta.

Es importante tener en cuenta que los DataFrame de Spark están distribuidos en la memoria RAM de los nodos, pese a que su manejo parezca ser una tabla de una base de datos. Por otro lado, el nombre DataFrame es el mismo que el definido en otras librerías de lenguajes como Python (Dataframes del paquete Pandas) o R (data.frame). Si bien el concepto es el mismo (tabla de datos cuyas columnas tienen nombre y tipo), la implementación y el manejo no tienen nada que ver, más allá de que los autores eligieron el mismo nombre. Los DataFrames de Spark son un tipo de dato definido por Spark en la JVM, están distribuidos en los nodos y se manejan mediante la API de Spark. En cambio, los Dataframes de Pandas existen solo en Python al importar específicamente esa biblioteca, no están distribuidos (toda la estructura está en la máquina única en la cual se esté ejecutando el intérprete de Python) y se utilizan mediante la API específica de Pandas. De la misma forma, la estructura data.frame de R está definida solo en el lenguaje R en el paquete base, no es distribuida y para utilizarla hay que acudir a la API de R. En el caso de Spark, existe en la API un método concreto de los DataFrames que permiten traerse todo el contenido a una sola máquina (el driver), devolviendo un Dataframe de Pandas. Es el método “toPandas()”, pero debe usarse con cuidado ya que, de nuevo, requiere que todo el contenido distribuido en los nodos quepa en la memoria RAM del proceso Driver. En caso contrario puede provocar una excepción *OutOfMemory*.

[2] http://spark.apache.org/docs/latest/sql-programming-guide.html

Lectura y escritura de DataFrames

Lectura de datos en pySpark de un datasource, que por defecto es HDFS:

```
myDF = spark.read.format(<formato>).load("/path/to/hdfs/file")
```

<formato> = "parquet" | "json" | "csv" | "orc" | "avro"
spark es el objeto SparkSession

Suele estar disponible un atajo como `spark.read.<format>("/path/to/file")` aunque no todos los formatos lo tienen.

En Parquet y ORC el tipo de dato de cada columna se almacena en el propio fichero. En otros casos, Spark puede intentar inferir el tipo de dato de cada columna, aunque se le puede indicar que no lo haga (y en ese caso leerá todo como string), o darle el esquema. En el caso de Parquet, el esquema se guarda junto a los datos así que los tipos leídos siempre son correctos (coinciden con los que había cuando se guardó).

Los CSV son más problemáticos. Spark no admite separador de más de un carácter

```
# En df1 se leen todas las columnas como string
df1 = spark.read.option("inferSchema", "false").csv("/path/hdfs/file")

myschema = StructType([
      StructField("columna1", DoubleType(), nullable = False),
      StructField("columna2", DateType(), nullable = False)
])
df2 = spark.read.option("header", "true")\ # pasamos el esquema para
                .option("delimiter","|")\   # que se lean correctamente
                .schema(myschema).csv("/path/hdfs/file")
```

Escritura en HDFS: `spark.write.<format>("path/to/hdfs/file")`

Donde <format> puede ser parquet, csv, etc. También puede escribirse el resultado en otros datastores muy diversos, siguiendo sintaxis específica. Para más detalles se puede consultar la web de Spark (nota al pie número 2).

Fuentes de datos en Spark

Para que Spark pueda conectarse a una fuente de datos debe existir un *conector* específico que indique cómo obtener datos de esa fuente y convertirlos en un DF

- HDFS: Spark puede leer varios formatos de archivo. *La terminación indicada en el nombre de archivo no informa a Spark de nada*, solo puede servir como pista al usuario que vaya a leer el fichero.

- Bases de datos:

    - Conectores para BBDD relacionales. Spark es capaz de leer en paralelo desde varios *workers* conectados simultáneamente a una BD relacional, cada uno leyendo porciones (trozos) diferentes de una tabla (cada

porción va a una partición). Spark puede enviar una consulta a esa BD y leer el resultado como DF

  - Conectores para BBDD no relacionales: específicos, desarrollados por la comunidad o por el fabricante. (Ejemplo: Cassandra, Mongo, Elastic…)

- Cola distribuida (Kafka): datos que se van leyendo de un buffer

- También datos que llegan en streaming a HDFS (ficheros que se van creando nuevos en tiempo real)

Manipulación de DataFrames: la API de Spark SQL[3]

La mayoría son operaciones sobre *columnas* (clase Column). Spark implementa muchas operaciones entre columnas de manera distribuida (operaciones aritméticas entre columnas, o con constantes, manipulación de columnas tipo string, comparaciones…) que debemos usar siempre que sea posible. Todas ellas son sometidas a optimizaciones por parte de Catalyst.

Utilización general : *objetoDataFrame.nombreDelMétodo(argumentos)*

Todas ellas devuelven como resultado un nuevo DataFrame. Por eso se suelen encadenar transformaciones: *df.método1(args1).método2(args2)*

Operaciones más frecuentes con DataFrames. Supongamos que hemos hecho
df = spark.read.parquet("/ruta/fichero/hdfs/datos.parquet"). Ahora podemos hacer:

- col("nombreCol") sirve para seleccionar una columna y devuelve un objeto Column. Es muy habitual llamarlo con F.col() tras ejecutar *import pyspark.sql.functions as F*.
- df.select("nombreColumna") o también select(F.col("nombreColumna"))
  - También: se pueden crear y seleccionar columnas al vuelo con el método select() (columnas existentes y operaciones con columnas que dan lugar a una columna nueva). Ej: creamos una columna diff y la seleccionamos (solo a ella) con df.select((F.col("edad") - F.lit(18)).alias("diff"))

  - IMPORTANTE: no se puede mezclar código SQL con objetos columna

- df.withColumn("nuevaCol", col("c1") + col("c2")): devuelve un nuevo DF con todas las columnas del original más una nueva columna añadida al final, como resultado de una operación entre columnas existentes que devuelva como resultado un objeto Column.

- df.drop("nombreColumna") ó drop(Column) para eliminar una columna

- df.withColumnRenamed("nombreExistente", "nuevoNombre") para renombrar columnas

[3] Documentación oficial: http://spark.apache.org/docs/latest/api/python/pyspark.sql.html

- when(condición, valorReemplazo1).otherwise(valorReemplazo2)
  para reemplazar valores de una columna según una condición que implica a esa o a otras columnas. Se utiliza generalmente en un withColumn:

  df.withColumn("esMayor", F.when("edad > 18", "mayor").otherwise("menor"))

- Funciones matemáticas y estadísticas que generan columnas y DataFrames nuevos de resumen
  - Columna de números aleatorios: rand (uniforme en [0,1]) o randn (normal estándar). Ejemplo: df.withColumn("aleat", F.rand())
  - DataFrame con estadísticos descriptivos (método describe())
  - Correlaciones entre columnas (devuelve un número), tablas de contigencia
- Unión de DataFrames: df1.unionAll(df2)
- Diferencia de DataFrames: df1.except(df2)
- Where (o filter) utilizando como argumento:
  - Una condición entre columnas: df.where(F.col("c1") > F.col("c2"))
  - Un string con una consulta en SQL: df.where("c1 > c2 and c3 != 'Francia'")
- Map y flatmap, sabiendo que siempre iteramos sobre objetos de tipo *Row*

```
def mifuncion(r): # r es un Row y se accede a sus campos mediante .
   return((r.DNI, "Bienvenido " + r.nombre + " " + r.apellido))

dfPares = df.map(mifuncion)
dfRenombrado = dfPares.toDF("DNI", "mensaje")
```

- Mismas operaciones de los RDD: transformaciones como sample, sort, distinct, groupBy … y acciones como *count*, *take*, *first*, etc

Ejemplo de código pySpark:

```
import pyspark.sql.functions as F

# el objeto spark (sparkSession) ya está creado en Jupyter. Sino,
# habría que crearlo indicando la dirección IP del master de un
# cluster de Spark existente

df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
resultDF = df.withColumn("distMetros", F.col("dist")*1000)\
  .withColumn("retrasoCat",
      F.when(F.col("retraso") < 15, "poco")\
       .when(F.col("retraso") < 30, "medio")
       .otherwise("mucho"))        # sin otherwise, pondría nulls!
  .select(F.col("aeronave"), F.col("origen"),
            F.col("disMetros"),
            F.col("retrasoCat"),
            (F.col("retraso") / F.col("dist")).alias("retrasoPorKm")
      )\
   .withColumnRenamed("dist", "distKm")
   .where(F.col("aeronave") == "Boing 747") # equivale a .filter()
   .where("distMetros > 20000 and origen != 'Madrid'")

# Línea anterior: where pasando una consulta SQL como string :-)
# En la línea siguiente hacemos de nuevo lo mismo

df2 = resultDF.select("retraso, compania")
  .where("aeropuerto like '%Barajas' and retrasoCat = 'poco'")

df2.show()
```

Ejemplos con funciones matemáticas y estadísticas[4]

```
# Generar con select dos columnas de distribuciones uniforme y normal
df.select("id", rand(seed=10).alias("uniform"),  # Uniforme en [0,1]
                randn(seed=27).alias("normal"))  # Normal estándar
  .show()
+--+------------------+--------------------+
|id| uniform          |              normal|
+--+------------------+--------------------+
| 0| 0.7224977951905031| -0.1875348803463305|
| 1| 0.2953174992603351|-0.26525647952450265|
| 2| 0.4536856090041318| -0.7195024130068081|
| 3| 0.9970412477032209|  0.5181478766595276|
| 4|0.19657711634539565|  0.7316273979766378|
| 5|0.48533720635534006| 0.07724879367590629|
| 6| 0.7369825278894753| -0.5462256961278941|
| 7| 0.5241113627472694| -0.2542275002421211|
| 8| 0.2977697066654349| -0.5752237580095868|
| 9| 0.5060159582230856|  1.0900096472044518|
+--+------------------+--------------------+
```

Podemos usar *describe* para calcular estadísticos descriptivos simples de cada columna que le pasemos como argumento, o de todas las columnas si no le pasamos ninguna:

[4] Fuente: Apache Spark Analytics Made Simple – Highlights from the Databricks Blog

```
In [4]: df.describe().show()
+-------+------------------+------------------+--------------------+
|summary|                id|           uniform|              normal|
+-------+------------------+------------------+--------------------+
|  count|                10|                10|                  10|
|   mean|               4.5| 0.5215336029384192|-0.01309370117407197|
| stddev|2.8722813232690143|  0.229328162820653|  0.5756058014772729|
|    min|                 0|0.19657711634539565| -0.7195024130068081|
|    max|                 9| 0.9970412477032209|  1.0900096472044518|
+-------+------------------+------------------+--------------------+

In [5]: from pyspark.sql.functions import mean, min, max
In [6]: df.select([mean('uniform'), min('uniform'), max('uniform')]).show()
+-----------------+------------------+-----------------+
|      AVG(uniform)|       MIN(uniform)|      MAX(uniform)|
+-----------------+------------------+-----------------+
|0.5215336029384192|0.19657711634539565|0.9970412477032209|
+-----------------+------------------+-----------------+
```

```
from pyspark.sql.functions import rand

df = sqlContext.range(0, 10).withColumn('rand1',
rand(seed=10)).withColumn('rand2', rand(seed=27))
df.stat.cov('rand1', 'rand2') # Resultado: 0.009908130446217347
df.stat.corr('rand1', 'rand2') # Resultado: 0.14938694513735398
df.stat.corr('id', 'id')      # Resultado: 1.0
```

Tablas de contingencia

```
# Creamos un DataFrame con dos columnas (name, item)
names = ["Alice", "Bob", "Mike"]
items = ["milk", "bread", "butter", "apples", "oranges"]
df = spark.createDataFrame([(names[i % 3],
      items[i % 5]) for i in range(100)], ["name", "item"])

# Vamos a mostra 10 filas por pantalla
df.show(10)
+-----+-------+
| name| item  |
+-----+-------+
|Alice|   milk|
| Bob |  bread|
| Mike| butter|
|Alice| apples|
| Bob |oranges|
| Mike|   milk|
|Alice|  bread|
| Bob | butter|
| Mike| apples|
|Alice|oranges|
+-----+-------+

df.stat.crosstab("name", "item").show()
```

```
+---------+----+-----+------+------+-------+
|name_item|milk|bread|apples|butter|oranges|
```

```
+---------+----+-----+------+------+-------+
|      Bob|   6|    7|     7|     6|      7|
|     Mike|   7|    6|     7|     7|      6|
|    Alice|   7|    7|     6|     7|      7|
+---------+----+-----+------+------+-------+
```

Ejemplos de agregaciones:

El método *groupBy("nombreCol1", "nombreCol2", …)* sobre un DataFrame devuelve una estructura de datos llamada *RelationalGroupedDataset*, que **no** es un DataFrame y sobre la que apenas se pueden aplicar operaciones. Las únicas que se suelen aplicar son *count*(), que efectúa un conteo del número de elementos por cada grupo, o la función *agg*() que es la más habitual, y realiza para cada grupo las agregaciones que le indiquemos sobre las columnas que le indiquemos. El resultado solamente contendrá aquellas columnas que se incluyeron como argumentos en el *groupBy* más aquellas que sean mencionadas como argumento de alguna de las operaciones de agregación que incluimos en la función *agg*.

Cuando ejecutamos funciones de agregación sin haber llevado previamente una agrupación con *groupBy*, se devuelve un DataFrame con una sola fila que es el resultado de la agregación. El fragmento de código siguiente muestra cómo se utilizan *groupBy* y *agg*.

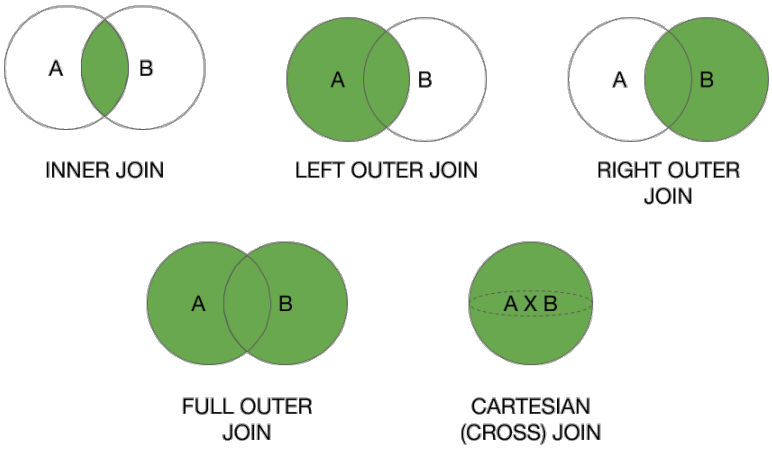
```
import pyspark.sql.functions as F

newDF = myDF.agg(max(F.col("mycol"))) # devuelve un DF de una sola fila

newDF = myDF.groupBy("mycol").agg(F.max(F.col("mycol"))) # Tantas filas
                                    # como  valores distintos en mycol
newDF = myDF.withColumn("complicated",
                  F.lit(2)*F.sin(F.col("colA"))*sqrt(F.col("colB")))

newDF = myDF.groupBy("id").agg(F.count("id").alias("count"),
                               F.max("date").alias("maxdate"),
                               F.countDistinct("prod").alias("nProd"))

# Sintaxis tipo diccionario para indicar varias agregaciones:
newDF = myDF.groupBy(F.col("someCol")).agg({"existingCol": "min",
                                            "otherCol": "avg"})

newDF = myDF.agg({"existingCol": "min",
                  "otherCol": "avg"}) # devuelve un DF de una sola fila
```

Operaciones Join:

Método *join*: misma operación que en SQL con misma semántica.

- Si el nombre de columna es el mismo, se especifica como string. Sino, se especifica una condición entre columnas que devuelva una columna de bool

- Provoca movimiento de datos entre nodos. Es la operación más pesada. Antes de usarla, debemos pensar si no hay más remedio.



Existen los mismos tipos de JOIN que en SQL (imagen izqda), incluso por condición arbitraria.

Si hay matching de una fila con varias filas del otro DF, se incrementa el número de filas del resultado respecto al número de filas original.

Fig. 6. Tipos de join.

Un tipo especial no contemplado en la figura es el *left_semi*: la operación **`A.join(B, ["col1", "col2"], "left_semi")`** es una especie de *filter* sobre el DataFrame A, de manera que el resultado contiene solo las columnas de A, y solo aquellas filas de A que tienen algún match con B en las columnas "col1" y "col2" (simultáneamente). No se añade ninguna columna de B al resultado.

A

Student			
stuId	**lastName**	**firstName**	**major**
S1001	Smith	Tom	History
S1002	Chin	Ann	Math
S1005	Lee	Perry	History
S1010	Burns	Edward	Art
S1013	McCarthy	Owen	Math
S1015	Jones	Mary	Math
S1020	Rivera	Jane	CSC

B

Enroll		
stuId	**classNumber**	**grade**
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

```
A.join(B, ["stuid"], "left_semi").show()
```

stuId	lastName	firstName	major
S1001	Smith	Tom	History
S1002	Chin	Ann	Math
S1010	Burns	Edward	Art
S1020	Rivera	Jane	CSC

El resultado tiene el mismo esquema (mismas columnas) de A, pero solo aquellas filas cuyo *stuid* coincide con algún *stuid* de la tabla B (no importa si coincide con uno o varios, es irrelevante).

Existe un tipo análogo de *join* llamado *left_anti*, cuyo comportamiento es retener aquellas filas del DF izquierdo (sin añadirles columnas adicionales) que no tienen correspondencia con ninguna fila del DF derecho. *Left_anti* sería la operación complementaria de *left_semi* ya que el resultado contendría justamente las filas del DF de la izquierda que serían excluidas por *left_semi*.

En pySpark, cuando en la llamada especificamos el tipo de join ("left_outer", "left_semi", etc) es necesario especificar las columnas por las que hacemos join como una lista, incluso aunque la lista solo esté formada por una columna. Si no se especifica nada, por defecto Spark lleva a cabo un inner join.

Ejemplos:

```
result = employees.join(departments, "DepartmentID“)   # inner join
result = employees.join(departments, ["DepartmentID“], "left_outer")
result = orders.join(products,      # join con condición arbitraria
      (F.col("product“) == F.col("name" )) &
      F.col("date“) >= F.col("startDate“) &
      F.col("date“) <= F.col("endDate")
)
```

User-Defined Functions (UDF)

Una UDF (User-Defined Function) es una función que recibe datos de cada fila y devuelve un resultado. La función recibe directamente el dato de una o varias columnas, como tipo simple (dato convertido de la JVM a Python). Se aplica a cada fila

Es similar a un *map* pero aplicado a un DataFrame, donde intervienen solo ciertas columnas. Recibe el tipo simple solamente de las columnas implicadas (en vez de recibir toda la fila como un objeto *Row* como ocurriría con *map* al iterar por las filas).

Se utiliza habitualmente dentro de **withColumn()** para *añadir una nueva columna a un DF existente, con el resultado de aplicar a cada fila nuestra función, en la que intervienen solo ciertas columnas de la fila*. En cambio, *map* devolvería un DF completamente nuevo (no añade columnas sino que fabrica un nuevo DF en el que hemos creado de 0 cada fila). Ejemplo:

```
# Definimos una función de Python (nada de Spark)
def computeInitialGroup(year1, year2): # year1 y year2 son strings
    if (year1 != "None_or_-1" and year1 != "Other"):
        return year1
    elif (year2 != "None_or_-1" and year2 != "Other"):
        return year2
    else:
        return "Unknown"

# creamos la UDF pasando como argumento nuestra función y el tipo de
# dato devuelto como resultado
udfinitial_group = F.udf(computeInitialGroup, StringType())

# Ahora invocamos a nuestra UDF pasándole 2 columnas (de tipo string)
myDF = spark.read.parquet("/tmp/misdatos.parquet")

# Asumimos que myDF contiene, entre otras columnas, "year1Col" y
# "year2Col"
newDF = myDF.withColumn("initialGroup", udfinitial_group(
                              F.col("year1Col"),
                              F.col("year2Col"))
```

Al invocar a la UDF (recordemos que *withColumn* es una transformación, por lo que sólo se ejecutará cuando encontremos una acción más adelante), Spark se ocupará de:

- Serializar el código Python de la función *computeInitialGroup*.
- Enviarlo a los workers en los que existan particiones del DataFrame *myDF*.
- Para cada fila de la partición, transformar el tipo de dato de las columnas year1Col y year2Col, que serán String de Java (Scala), al tipo string de Python, y entonces invocar a *computeInitialGroup* pasándole los valores convertidos a

Python. Esta operación implica un coste computacional al convertir tipos de un lenguaje (Java) a otro (Python) pero es inevitable en las UDF de Python, salvo en la versión 2.4 en los casos que puedan solucionarlo las PandasUDF[5].

JOBS, STAGES Y TASKS (TRABAJOS, ETAPAS Y TAREAS)

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 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(…)

Una task (tarea) son cada una de las transformaciones que forman una etapa. Es la unidad mínima de trabajo de Spark.

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

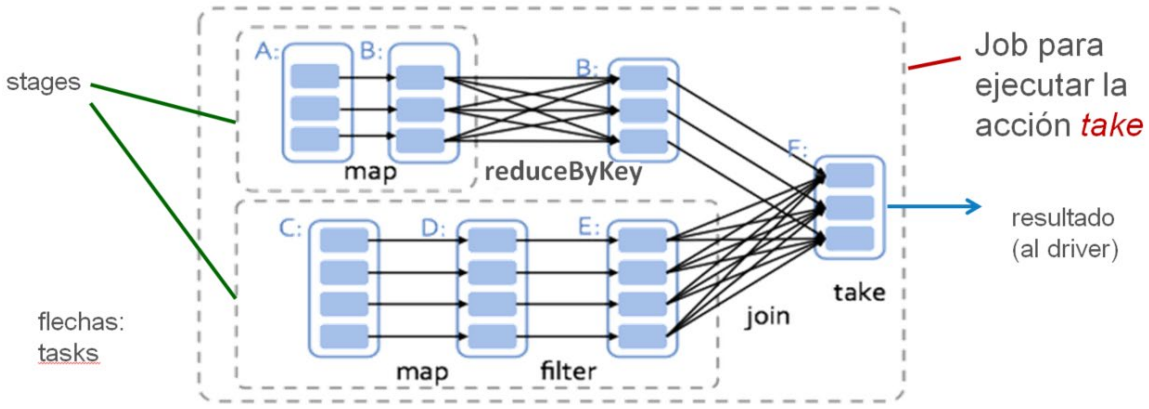


Fig. 7. Representación de Jobs, stages y tasks en Spark

BIBLIOGRAFÍA RECOMENDADA

- Bill Chambers y Matei Zaharia. *Spark: The Definitive Guide* O'Reilly, 2018.
- La web oficial de Spark, https://spark.apache.org/docs/latest/ contiene una extensa documentación.

[5] http://spark.apache.org/docs/latest/sql-pyspark-pandas-with-arrow.html