Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

SPARK-SCALA

TRABAJO FIN DE MÁSTER

Jesús Salvador Renero Quintero

2016

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

**Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos**

TRABAJO FIN DE MÁSTER

SPARK-SCALA

Autor
Jesús Salvador Renero Quintero

Director
Joaquín Salvachúa Rodríguez

Departamento de Ingeniería de Sistemas Telemáticos

2016

Índice

1 INTRODUCCIÓN 5

1.1 ¿QUÉ ES SPARK? 6

1.2 STACK UNIFICADO 6

1.2.1 SPARK CORE 8

1.2.2 SPARK SQL 8

1.2.3 SPARK STREAMING 9

1.2.4 MLLIB 9

1.2.5 GRAPHX 10

1.2.6 GESTORES DE INFRAESTRUCTURA 10

2 RDD 11

2.1 FUNCIONES BÁSICAS 11

2.1.1 CREACIÓN DE RDD 13

2.1.2 OPERACIONES CON RDD 14

2.2 TRANSFORMACIONES Y ACCIONES MÁS COMUNES 18

2.3 PERSISTENCIA 23

3 PARES CLAVE-VALOR 24

3.1 CREACIÓN DE RDD DE PARES 24

3.1.1 AGREGACIONES 24

3.1.2 AJUSTE DEL NIVEL DE PARALELISMO 28

3.1.3 AGRUPACIONES 29

3.1.4 JOIN 30

3.1.5 ORDENACIÓN 30

3.1.6 ACCIONES 30

3.2 PARTICIONADO DE DATOS 31

3.2.1 PARTICIONADO DE UN RDD 35

3.2.2 OPERACIONES QUE AFECTAN O SE BENEFICIAN DEL PARTICIONADO 35

3.2.3 EJEMPLO: PAGERANK 37

3.2.4 PARTICIONADO PERSONALIZADO 39

4 EJEMPLO DE USO REAL DE SPARK 41

4.1 TRANSFORMACIONES 42

4.2 AGREGACIONES 43

4.3 ANÁLISIS 45

4.4 PREDICCIONES 46

4.5 SPARKSQL 47

5 CONCLUSIÓN 49

6 REFERENCIAS 52

Terminología

Algunos de los términos incluidos en el presente trabajo se han trasladado directamente del inglés sin traducir. La razón es la mayor familiaridad con el término en su lengua original que con la posible traducción que en cada caso, pueda tener. En este apartado se enumeran dichos términos, así como una posible traducción o explicación del mismo.

Término	Explicación
Cluster	*Grupo de máquinas interconectadas que trabajan de forma colaborativa.*
Shuffle	*Operación por la que las claves de un conjunto de datos son enviadas entre las máquinas de un cluster, según una función de asignación de claves a nodos.*
Batch	*Procesado por lotes*
Map	*Operación que transforma cada elemento de una colección de datos, según un función dada.*
Join	*Operación de unión entre conjuntos de datos, según una o varias claves.*
Hash	*Función de resumen que asigna un valor de salida finito a un conjunto de datos de entrada.*
Driver	*En Spark, el modulo de control del entorno de computación distribuido, residente en una sóla máquina también conocida como "master".*
Combiner	*Operación de MapReduce que combinaba y enviaba juntas todas las claves asignadas a un mismo nodo, en lugar de enviarlas de una en una.*

1 Introducción

Spark es un proyecto de software libre creado y mantenido por una floreciente y creciente comunidad de desarrolladores desde el año 2009, momento en el que nació como un proyecto de investigación en el UC Berkeley RAD Lab (más tarde conocido como el AMPLab –*Algorithms, Machines, People Lab*). Los investigadores de dicho laboratorio comenzaron trabajando con Hadoop, y se dieron cuenta de que MapReduce es tremendamente ineficiente [1] para tareas iterativas o interactivas. Así que, desde sus inicios, el objetivo primordial de Spark [2] fue que su diseño permitiera la rápida ejecución de *consultas* interactivas y algoritmos iterativos, por medio del almacenamiento en memoria y la recuperación eficiente de fallos.

Los primeros artículos aparecieron en conferencias, y poco después de su creación en 2009, Spark ya era 10-20 veces más rápido que MapReduce para ciertos tipos de tareas.

Los primeros usuarios de Spark fueron otros grupos de investigación dentro de la Universidad de California Berkeley. Entre ellos, investigadores en *machine learning* del proyecto *Mobile Millenium*, que usaron Spark para monitorizar y predecir problemas en el tránsito de vehículos en el área de la bahía de San Francisco. Sin embargo, en poco tiempo, otros grupos comenzaron también a usar Spark, y hoy son muchas las organizaciones de todo tipo que lo emplean y participan en los Spark *Meetups* o el Spark *Summit*. Además de la Universidad de Berkeley, las principales organizaciones que contribuyen al desarrollo de Spark son Databricks, Yahoo! e Intel.

Spark Fue liberado como proyecto de software libre en marzo de 2010 y fue transferido a la *Apache Software Foundation* en junio de 2013, donde figura como proyecto del más alto nivel dentro de esta organización.

1.1 ¿Qué es Spark?

Spark es una plataforma de computo distribuido de propósito general, diseñada para ser, sobre todo, rápida.

Spark extiende el modelo MapReduce e incluye capacidad de ofrecer consultas interactivas o computación en *streaming*. El factor determinante que hace que Spark sea más rápido es su capacidad para ejecutar la mayoría de las operaciones en memoria, pero el diseño de Spark hace que todo el sistema sea también más eficiente ejecutando tareas complejas que dependen mayormente del acceso a disco.

Sobre su diseño de propósito general, la clave está en que su diseño permite combinar diferentes cargas de trabajo que tradicionalmente implicaban la combinación de varios sistemas distribuidos distintos (aplicaciones *batch*, algoritmos iterativos, queries interactivas y *streaming*). Spark utiliza el mismo motor interno para dar soporte a todo esta variedad de tareas, permitiendo combinarlas de forma sencilla y barata, y reduciendo además el coste de gestión que implica mantener diferentes herramientas y sistemas.

1.2 Stack unificado

El núcleo de Spark es un "motor de cómputo" responsable de la planificación, distribución y monitorización de aplicaciones, que a su vez, consisten en multitud de tareas que se ejecutan en las máquinas que componen un *cluster*. La filosofía de

Spark se basa en la fuerte integración de todos sus componentes. El núcleo es, por tanto, el componente que habilita al resto de componentes de alto nivel que permiten las diferentes especializaciones, como el SQL o el aprendizaje máquina.

Las ventajas de esta integración fuerte presenta varias ventajas siendo la más evidente de todas ellas, que los componentes de alto nivel se benefician directamente de todas las mejoras que se presenten en el núcleo común que todas ellas comparten. Por ejemplo, cualquier optimización en el núcleo de Spark permite que las librerías SQL o de aprendizaje máquina mejoren automáticamente. En segundo lugar, los costes de ejecutar un sistema de este tipo se reducen, al pasar de desplegar, instalar, probar, administrar y mantener varios sistemas a tener sólo uno. Este hecho implica también que cada vez que se añade un nuevo componente a Spark, la organización que lo haya adoptado se beneficia automáticamente de él. El coste, por tanto, de introducir un nuevo componente en términos de despliegue, administración y aprendizaje se reduce al hecho de actualizar Spark.

Finalmente, quizá el mayor beneficio de esta arquitectura fuertemente integrada se encuentra en la posibilidad de combinar varios modelos procesamiento en una misma aplicación. El ejemplo más evidente es emplear Spark para clasificar mediante algún algoritmo de aprendizaje máquina, datos provenientes en tiempo real de una fuente en *streaming*. Simultáneamente, el resultado de esa clasificación podría ser consultada, también en tiempo real, vía SQL, además de poder ser objeto de un análisis más detallado, a través de la consola de Spark, o por medio de aplicaciones que efectúen un procesado *batch* más dirigido. Todo ello, con un único sistema.

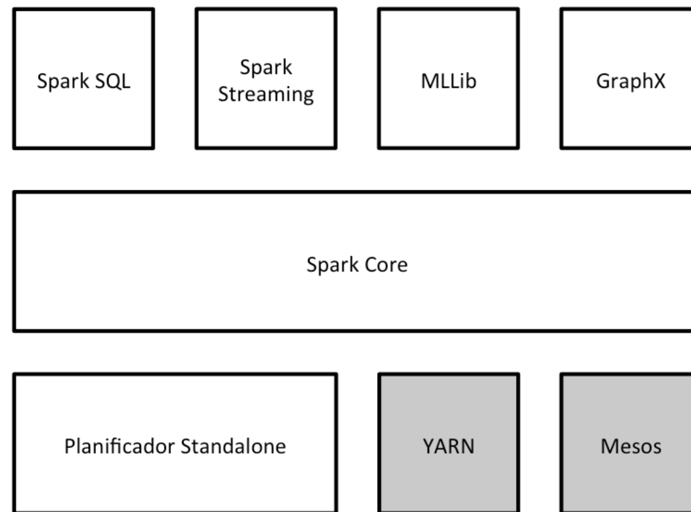

Figura 1

En la figura puede observarse la distribución en capas de los distintos componentes de Spark que se van a analizar con mas detalle en las secciones posteriores del documento.

1.2.1 Spark Core

El núcleo de Spark contiene la funcionalidad principal de Spark para la planificación de tareas, gestión de memoria, recuperación ante fallos, gestión de almacenamiento y algunos mas. Pero la característica más importante del núcleo Spark es que es donde reside el API de los RDD [3] (*Resilient Distributed Datasets,* que podríamos traducir como "conjuntos de datos distribuidos robustos"), y que son la abstracción de programación central en todo el modelo que propone Spark. Un RDD representa cualquier colección de elementos distribuidos entre varios nodos de computo y que pueden ser objeto de ser manipulados en paralelo.

1.2.2 Spark SQL

Spark SQL es el módulo de Spark encargado de trabajar con datos estructurados. Permite ejecutar consultas en SQL o en HQL de Hive, y es capaz de reconocer

varios tipos diferentes de fuentes de datos como Parquet, JSON o tablas de HIVE, por ejemplo. Mas allá de ofrecer un interfaz SQL a Spark, lo que este componente permite es la mezcla de consultas SQL con el resto de manipulaciones programáticas de los datos que es posible realizar con los RDD (en cualquier de los lenguajes soportados), todo ello en una misma aplicación, combinando así el SQL con la analítica más compleja. Esta integración tan estrecha con el entorno tan potente de computo que ofrece Spark, diferencia completamente a Spark SQL de cualquier otra herramienta de "*data warehousing*" existente.

1.2.3 Spark Streaming

Este componente permite el procesamiento de streams de datos como por ejemplo, los ficheros de log de servidores de producción, o las colas de mensajes con actualizaciones de estado de usuarios de un servicio web, por citar algunos. El API que ofrece Spark para manipular este tipo de fuentes de datos es prácticamente idéntica a la ofrecida para los RDD, simplificando el aprendizaje necesario por los programadores.

1.2.4 MLLib

Spark también ofrece una librería con funcionalidad básica de aprendizaje máquina (*machine learning* –ML) llamada MLLib, que implementa varios algoritmos de clasificación, regresión, *clustering* o filtrado colaborativo, además de contar con métodos de evaluación de modelos e importado de datos. Además de los modelos genéricos, MLLib también da acceso a funciones de bajo nivel en ML como el algoritmo de optimización por descenso del gradiente. Todos ellos han sido diseñados para que su funcionamiento escale en un *cluster*.

1.2.5 GraphX

GraphX es la librería que ofrece Spark para la manipulación de grafos y realizar cálculos distribuidos sobre ellos. Al igual que el resto de componentes enumerados hasta ahora, GraphX también replica y extiende el API ofrecido por los RDD desde el núcleo de Spark. Entre las cosas que nos permite este componente, es posible crear un grafo dirigido con propiedades arbitrarias enlazadas a cada uno de sus vértices o sus arcos. GraphX permite realizar operaciones sobre los grafos como subgraph o mapVertices, además de contar con una librería adicional que implementa los algoritmos más típicos de los grafos (PageRank o cuenta de triángulos / cliques).

1.2.6 Gestores de Infraestructura

Spark también es flexible a la hora de decidir sobre que gestor de infraestructura debe ejecutarse. Spark puede correr sobre Hadoop YARN, Apache Mesos o un gestor muy simple de infraestructura incluido en su distribución llamado *Standalone Scheduler*, que debería usarse únicamente cuando se utiliza Spark sobre un conjunto de máquinas vacías, sin ningún otro componente, para poder comenzar a usarlo de forma rápida y sencilla.

2 RDD

En esta sección se describe el funcionamiento de los RDD, la extracción que utiliza Spark para trabajar con datos de manera distribuida. En Spark, todas las tareas se expresan en forma de creación de nuevos RDD, transformación de RDD existentes, u operaciones sobre RDD para proporcionar un resultado final. La gran venta Spark es que automáticamente distribuye los datos contenidos dentro de los RDD a lo largo del cluster y paraleliza, también automáticamente, todas las operaciones que se realizan sobre ellos.

2.1 Funciones básicas

Un RDD es una colección distribuida e inmutable de objetos. Cada RDD es dividido en múltiples particiones, pudiendo albergar tipos de objeto de cualquiera de los lenguajes ofrecidos por Spark. Un RDD puede quedarse dos maneras: mediante la carga de un conjunto de datos externo, o la distribución explícita de objetos (como por ejemplo, una lista o un conjunto). A continuación se muestra un ejemplo como cargar un conjunto de datos externo.

Ejemplo 1. Creación de un RDD de Strings a partir de un fichero de texto.

```
1. scala> val lines = sc.textFile("README.md")
```

En este ejemplo SC hace referencia al contexto que utiliza parte desde la consola de Spark. Una vez creado el RDB hay dos tipos de operaciones posibles: transformaciones y acciones. Las transformaciones construyen un nuevo RDB a partir de uno ya creado previamente. Por ejemplo, una transformación muy común es el filtrado de datos que se corresponden con un predicado dado. En nuestro ejemplo con el fichero de texto, podríamos crear, o transformar, nuestro RDD

original en otro nuevo, simplemente filtrando aquellas cadenas de texto que contengan la palabra `scala`, como se muestra en el siguiente ejemplo.

Ejemplo 2. Filtrado de líneas.

```
1.  scala> val scalaLines = lines.filter(line => line.contains("scala"))
```

Las acciones, por otro lado, calculan un resultado basándose en un RDD existente, y por tanto, lo que hacen es devolverlo a la consola o guardarlo en un sistema de almacenamiento externo (p.ej.: HDFS).

Acciones de transformaciones son diferentes simplemente por la manera en que Spark calcula los RDD. Aunque pueden ser definidos en cualquier momento, Spark calcula su valor en modo *'lazy'* o vago –sólo se calcula su valor cuando es necesario para una acción. Esta aproximación puede parecer inusual, pero tiene todo el sentido cuando se trabaja con grandes volúmenes de datos. En el ejemplo anterior, si Spark hubiera leído todas las líneas del fichero, suponiendo que éste hubiera sido muy grande, habría desperdiciado un muchos recursos, sobre todo teniendo en cuenta que la operación que se iba ejecutar inmediatamente después es un filtrado de todas ellas. En su lugar, Spark intenta comprender toda la cadena de transformaciones que se le solicita para así calcular tan sólo los datos que son necesarios para obtener el resultado final.

Los RDD de Spark son re calculados cada vez que se ejecuta o se solicita una acción sobre ellos. Por eso es importante que si se pretende reutilizar un RDD, se le pida a Spark que convierta ese RDD en un objeto persistente, De manera que no sea necesario recalcularlo cada vez. Para hacer esto tan sólo es necesario lo siguiente:

Ejemplo 3. Haciendo que un RDD sea persistente en memoria.

```
1.  scala> scalaLines.persist
```

En este ejemplo hemos solicitado a Spark que haga persistente el objeto en memoria, pero podemos solicitarle que lo haga también en disco o en varias combinaciones de ambos, como se verá más adelante. En este ejemplo Spark calcular el resultado una primera vez Y después almacenada el contenido del RDD en memoria en todas las máquinas que componen el cluster, para así poder reutilizarlo en futuras acciones. La no persistencia podría parecer que no tiene demasiado sentido, pero si lo tiene en el caso de utilizar grandes conjuntos de datos: si no se va a reutilizar el RDD, no hay razón para desperdiciar espacio de almacenamiento cuando es posible volver a calcular el resultado en cualquier momento, recorriendo todos los datos de nuevo. Y esta es, principalmente, la razón de la letra `R` en los RDD. Cuando una máquina que guarda un RDD falla, Spark recalcula las particiones que faltan de forma transparente al usuario.

2.1.1 Creación de RDD

La manera más sencilla de crear un RDD es coger una colección existente dentro del programa que se esté desarrollando y pasárselo al método `parallelize` del SparkContext, tal y como se muestra en el siguiente ejemplo.

Ejemplo 4. Método `parallelize`.

```
1.  scala> val linesRDD = sc.parallelize(List("pandas", "i like pandas"))
2.  res4: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[3] at
    parallelize at <console>:22
```

Como se muestra en el ejemplo, el resultado es un RDD que contiene objetos del tipo String. La ventaja que presenta esta aproximación es que resulta muy útil cuando se estaba aprendiendo Spark, ya que permite crear rápida y fácilmente nuestros propios RDD desde la consola y efectuar operaciones sobre ellos. Sin

embargo la manera más común de crear los RDD es la carga de datos a partir de almacenamiento externo, tal y como se ha visto en el ejemplo 1.

2.1.2 Operaciones con RDD

Como ya se ha tratado anteriormente, los RDD permiten dos tipos de operaciones: Transformaciones y acciones. Las transformaciones son operaciones sobre RDD que devuelvan un nuevo RDD, como por ejemplo `map` o `filter`. Las acciones son operaciones que devuelven un resultado o lo guardan en disco, pero para los que son necesarias hacer un cómputo, como por ejemplo `count` o `first`. Spark trata estos dos tipos de operaciones de manera muy diferente.

Driver

Antes de entrar el detalle de escribir qué y cómo se ejecutan las acciones en Spark es necesario describir un componente clave dentro de la arquitectura llamado driver. El driver es el proceso donde se ejecuta el método `main` del programa que queremos ejecutar. Es también el proceso que crea el SparkContext, los RDD y es responsable de llevar a cabo todas las transformaciones y acciones. Cuando se lanza una consola de Spark como la que viene recargada en la distribución de Spark, lo que se hace realmente es lanzar un programa driver.

Aún nivel de detalle funcional más bajo, el driver responsable de convertir un programa en unidades de ejecución más pequeñas llamadas tareas. Estas tareas surgen el análisis del grafo cíclico dirigido creado de manera implícita por cualquier programa que quiere ejecutarse en Spark (p.ej.: Figura 2). En este punto se realizan varias transformaciones con el objeto de optimizar la secuencia de ejecución resultante en el brazo, y convertirla en un conjunto de tapas. Cada etapa,

a su vez, consiste en múltiples tareas. Estas tareas son la unidad ejecución más pequeña que maneja Spark, empaquetándolas y preparándolas para ser enviadas a cualquiera de las máquinas que componen el cluster.

Dado un plan ejecución de tareas, la coordinación y planificación de las mismas también es responsabilidad del driver. Pero la característica más interesantes que nos interesa resaltar en este punto, es la de contar con toda la información de coordinación asociada a las estructuras de datos distribuidas, pero no de capacidad distribuida de almacenamiento. Esto significa, que el resultado de cualquiera de las acciones que llevamos acabo sobre RDD resultan en una estructura de datos que debe caber en la memoria del driver. Por tanto este es un aspecto que debe tenerse en cuenta a la hora de diseñar las operaciones, Como se verá más adelante en este documento.

Transformaciones

El aspecto más importante de las transformaciones, dejando a un lado que deben devolver otro RDD, es que son calculadas en modo perezoso, o *lazy*, lo que quiero decir que sólo se ejecutará el cómputo necesario cuando sea necesario para una acción. La mayoría de las transformaciones actual al nivel de elemento, lo que quiere decir que sólo tratan uno de los elementos de la estructura RDD sobre la que operan a la vez. El ejemplo más común es el tratamiento de un gran conjunto de datos de un fichero de *log* del que se quieren extraer únicamente las líneas que contengan mensajes de error.

Ejemplo 5. Transformación de filtrado sobre un RDD.

```
1.  val inputRDD = sc.textFile("log.txt")
2.  val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

El resultado de estas dos operaciones no es más que un puntero a un nuevo RDD. De hecho podríamos ejecutar un nuevo filtrado sobre el conjunto de datos de entrada buscando una cadena de texto diferente.

Ejemplo 6. Transformación de union sobre un RDD.

```
1.  val inputRDD = sc.textFile("log.txt")
2.  val errorsRDD = inputRDD.filter(line => line.contains("error"))
3.  val warningsRDD = inputRDD.filter(line => line.contains("error"))
4.  val badLines = errorsRDD.union(warningsRDD)
```

Cada una de estas transformaciones provoca que Spark vaya manteniendo las dependencias entre ellos en un grafo. Esta información se utiliza para calcular cada RDD a medida que sea necesario y así poder recuperarse de posibles pérdidas de datos, en el caso de que un RDD persistente se pierda. En la siguiente figura se muestra uno de estos grafos de dependencias para el ejemplo anterior.

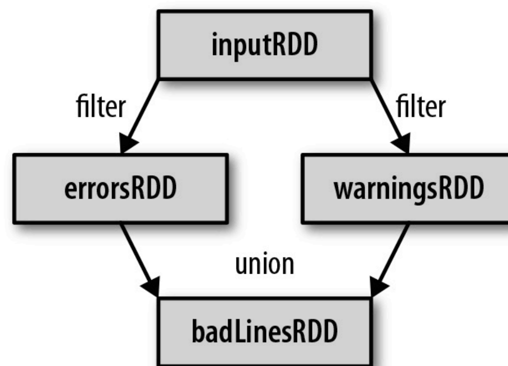

Figura 2

Acciones

Hasta ahora hemos visto cómo crear RDD Y cómo realizar transformaciones sobre ellos, pero en algún momento queremos hacer algo con nuestro conjunto de datos. Como se ha dicho antes, las acciones son el segundo tipo de operaciones sobre RDD. Son las operaciones las que devuelven valor final al driver o son las encargadas de escribir datos a un sistema de almacenamiento externo. Por tanto

las acciones fuerzan la evaluación de la transformación requerida por el RDD desde el que son invocadas, ya que necesitan producir una salida.

Siguiendo con el ejemplo anterior de nuestro fichero de *logs*, en un momento dado querremos bien contar el número de líneas (`count`), o examinar algunas de ellas (`take`).

Ejemplo 7. Cuenta el número de líneas y muestra en pantalla las 10 primeras.

```
1.  scala> badlines.count
2.  badlines.take(10).foreach(println)
```

En el ejemplo anterior y quedamos sobre las 10 primeras líneas del RDD, para lo cual es necesario traerlas desde su localización distribuida hacer el driver. Existe otro método sobre RDD muy útil, llamado `collect`, que trae el RDD entero a la memoria del driver. Este método solo utilizarse cuando el resultado tiene un tamaño pequeño Y lo que quiere es procesarse manera local. Debe tenerse cuidado con esta operación ya que el RDD debe acabar habiendo entero en la memoria de una sola máquina, en la que se ejecuta el driver.

La mayoría de los casos el RDD resultante no cabrá en la memoria después de realizar una operación `collect`, por lo que lo interesante será escribirlo en un sistema de almacenamiento externo como HDFS o S3.

Evaluación Perezosa

Como sea expuesto anteriormente, las transformaciones sobre RDD son evaluadas de forma perezosa, lo que significa que Spark no las ejecutará hasta que no Sean necesarias en una acción. Debemos, por tanto, pensar en los RDD no como en una estructura que contiene datos, si no en un conjunto de instrucciones capaces de calcular esos datos a través de transformaciones.

La razón por la que Spark emplear la evaluación perezosa es la reducción del número de pasadas que es necesario hacer sobre los datos, Entiendo agrupar el mayor número de operaciones juntas. En sistemas como Hadoop MapReduce, los desarrolladores pasan mucho tiempo pensando en cómo agrupar operaciones de manera que es el número de pasadas se minimice. En Spark, diseñar una operación *map* de forma compleja no aporta un beneficio sustancial si lo comparamos con el encadenamiento de un número mayor de operaciones más sencillas. Así, los desarrolladores pueden organizar sus programas en operaciones más pequeñas inmanejables.

2.2 Transformaciones y Acciones más comunes

Transformaciones elemento a elemento

Las dos transformaciones más comunes que se utilizan en Spark son `map` y `filter`. `map` aplica una función, que toma como argumento, a cada uno de los elementos del RDD siendo la salida de la función sobre cada uno de ellos, el resultado. `filter` también aplica una función booleana que toma como argumento sobre cada uno de los elementos del RDD pero la salida son sólo aquellos para los que se cumple la condición expresada en la función.

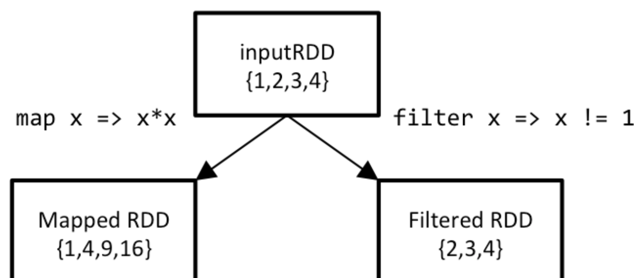

Figura 3

La función `map` es extremadamente versátil y puede emplearse para conseguir un gran número de transformaciones sobre los datos. Nótese que la salida de esta función no tiene porque ser del mismo tipo que los datos de entrada.

Habrá ocasiones en las que se querrá producir múltiples elementos de salida y no sólo uno. La operación que hace esto posible es `flatMap`. Al igual que con la función `map`, esta transformación se aplica a cada uno de los elementos de manera individual. Pero en lugar de devolver el único elemento, lo que se devuelve es un iterador con varios valores de retorno. La particularidad esta gente en lugar de producir un único RDD de iteradores, lo que se devuelve es un RDD qué consisten los elementos de todos los iteradores.

Ejemplo 8. flatMap aplica una función que devuelve una lista, y aplana el resultado sobre la lista original.

```
1.  scala> def g(v:Int) = List(v-1, v, v+1)
2.  g: (v: Int)List[Int]
3.
4.  scala> l.map(x => g(x))
5.  res64: List[List[Int]] = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4), Li
    st(3, 4, 5), List(4, 5, 6))
6.
7.  scala> l.flatMap(x => g(x))
8.  res65: List[Int] = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
```

Operaciones con Pseudo-conjuntos

Los RDD permiten utilizar muchas de las operaciones que se aplican sobre los conjuntos matemáticos, como por ejemplo la unión hola intersección, incluso cuando los RDD no sean conjuntos propiamente dichos. La propiedad de los conjuntos que normalmente no se cumple en los RDD es la de la unicidad de sus elementos, ya que normalmente tenemos duplicados. Si este fuera el caso y quisiéramos eliminarlos existe una transformación que produce un nuevo RDD conteniendo únicamente elementos distintos, llamada `distinct`.

La operación más sencilla entre conjuntos es la `union`, que produce un nuevo RDD con los datos de ambas estructuras de origen. Al contrario de lo que sucede con la operación matemática de unión, si existen duplicados en las estructuras de datos de entrada, el resultado contendrá duplicados.

Spark También permite la intersección (`intersection`) que devolverá sólo aquellos elementos presentes en ambos RDD. En este caso, los duplicados desaparecerán. El rendimiento de esta operación de intersección es mucho peor que el de la unión, al requerirse una operación de mezcla (`shuffle`) para identificar aquellos elementos comunes entre ambas estructuras de datos.

La operación `subtract` elimina aquellos elementos de un RDD presentes en un segundo. Es decir el resultado serán aquellos valores presentes en el primer RDD pero no en el segundo. Al igual que en la transformación de intersección, Es necesario un *shuffle*.

También es posible calcular el producto cartesiano entre dos RDD. Esta transformación produce todas las posibles combinaciones de pares (a, b) en las que `a` pertenece al primer RDD, y `b` al segundo. Esta transformación puede ser interesante si consideramos calcular el grado de similitud entre todos los posibles pares de una estructura de datos, como por ejemplo las preferencias de los usuarios. Esta transformación es posiblemente la más cara de llevar a cabo en Spark.

Acciones

La acción más común que se lleva acabo sobre un RDD es la operación `reduce`, que toma una función como argumento que opera sobre una pareja de elementos y

devuelve sólo uno del mismo tipo. El ejemplo más sencillo es la función +, que podemos utilizar para sumar todos los elementos de nuestro RDD. La operación `reduce` se usa para sumar fácilmente todos los elementos de un RDD, contarlos o llevar acabo otros tipos de agregación.

Ejemplo 9. Operación `reduce` *en Scala y su equivalente en notación anónima sobre iterables.*

```
1. scala> val sum = rdd.reduce((x, y) => x + y)
2. scala> val sum = rdd.reduce(_ + _)
```

La siguiente operación popular en las acciones en Spark es `fold`, que también toma una función como argumento, siguiendo la misma notación que `reduce`, pero además permite especificar una valor inicial para la función. Este valor inicial juega el papel de elemento identidad de la operación; es decir, aplicarlo a cualquiera de los valores que toma la función no altera su valor (p.ej.: 0 es el elemento identidad de la suma).

Tanto `fold` como `reduce` requieren que el tipo devuelto como resultado sea del mismo tipo que todos los elementos de RDD sobre el que operan. La operación que nos libera de esta restricción es `aggregate`, que utiliza la misma sintaxis que `fold` usando también un valor inicial. Tras él, se especifica una función que combina los elementos del RDD con un acumulador. Y finalmente, una segunda función mezcla los dos acumuladores, asumiendo que cada nodo acumula sus propios resultados localmente. Un uso claro de este tipo de métodos es el cálculo del promedio de un RDD sin usar `map` seguido de `fold`:

Ejemplo 10. Ejemplo de `aggregate`*.*

```
1. scala> val result = input.aggregate((0, 0))(
2.   (acc, value) => (acc._1 + value, acc._2 + 1),
3.   (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
4. scala> val = result._1 / result._2.toDouble
```

Existen otras acciones sobre RDD que envían algunos o todos los datos al driver en forma de colección O simplemente un valor. La operación más común de traer todos los datos a nuestro driver es utilizar la operación `collect()`, con la que debe tenerse especial cuidado de que todo sus contenidos quepan en la memoria de una sola máquina en la que se ejecuta el driver. Si lo que queremos es acceder a *n* elementos de nuestro RDD, debe utilizarse la operación `take(n)`. Debe decirse que este método intenta minimizar el número de particiones a las que accede, Y por tanto, el orden de los elementos retornados puede no ser el esperado. Otras operaciones que interesantes son, `top()`, que respeta el orden predefinido de nuestros datos, o `takeSample`, que permite muestrear con o sin reemplazo.

Este tipo de operaciones son muy útiles para hacer tests unitarios O depurado del código, pero pueden suponer un cuello de botella cuando se trabaja con grandes cantidades de datos.

En otras ocasiones se persigue efectuar algún tipo de acción sobre los elementos del RDD pero si devolver ningún resultado al driver. Un ejemplo de esto puede ser la inserción de registros en una base de datos. Para este tipo de situaciones, `foreach()` permite hacer transformaciones sobre cada elemento del RDD sin tener que traerlos a la memoria del driver de forma local.

Para finalizar dos operaciones estándar sobre RDD qué hacen exactamente lo que su nombre indica: `count` devuelve el número de elementos de una colección, y `countByValue` qué devuelve un mapa con los valores únicos y la cuenta de sus apariciones.

2.3 Persistencia

Como ya se ha explicado, Spark lleva acabo evaluación perezosa de los RDD, pudiendo darse el caso de que un mismo RDD sea necesario utilizarse varias veces. Si no hacemos nada al respecto, Spark recalcular a el RDD y todas sus dependencias cada vez que se derive una acción sobre él, lo que puede resultar especialmente caro en algoritmos iterativos.

Ejemplo 11. Ejemplo de recalculo de un RDD sin persistencia.

```
1. val result = input.map(x => x*x)
2. println(result.count())  // se calcula result para producir ‘count’
3. println(result.collect().mkString(",")) // result se recalcula de nuevo
```

Para evitar este hecho se solicita a Spark que convierta el dato en persistente. Cuando hacemos esto, cada nodo responsable calcular su parte de un RDD, la almacena internamente. Si un nodo que tiene datos marcados como persistentes falla, Spark recalculando las particiones perdidas de esos datos cuando sean necesarias. También es posible replicar estos datos en múltiples nodos de manera que la gestión de fallos sea mucho más rápida. Existen varios niveles de almacenamiento de datos persistentes, siendo posible especificar si queremos que Spark serialice o no los datos en disco, en memoria, o en combinaciones de ambas.

Ejemplo 12. Ejemplo RDD persistente.

```
1. val result = input.map(x => x*x)
2. result.persist(Storagelevel.DISK_ONLY)
3. println(result.count())
4. println(result.collect().mkString(","))
```

Cuando se intentan hacer persistentes demasiados datos como para que quepan en memoria, Spark automáticamente despreciará aquellas particiones más antiguas utilizando una política de caché LRU.

3 Pares Clave-Valor

El trabajo con pares clave-valor es una de las tareas más típicas a realizar con Spark. Son probablemente el componente principal de la mayoría de las agregaciones y trabajos de ETL que se realizan sobre cualquier conjunto de datos. Spark cuenta con un conjunto especial de operaciones sobre RDD formadas por parece clave en vano que le permiten trabajar con cada clave en paralelo o reagrupar datos en una red de nodos formando un cluster.

3.1 Creación de RDD de pares

Existen muchas maneras de crear un RDD de pares. La más sencilla de todas es literal sobre nuestros datos con una función `map` y devolver tuplas, como en el siguiente ejemplo, en el que se leen líneas de texto Y se devuelven tu planes en las que la clave es la primera palabra de cada línea y el valor es la línea entera.

Ejemplo 13. Construcción de un RDD de pares.

```
1.  val pairs = lines.map(x => (x.split(" ")(), x))
```

Los RDD de pares siguen siendo RDD por lo que podríamos aplicar el resto de funciones que aplicamos de manera normal a cualquier otro RDD. Por ejemplo, podríamos aplicar la función para eliminar todas aquellas líneas cuya clave cumpla una condición dada.

Ejemplo 14. Filtrado de todas las líneas de mas de 20 caracteres en un RDD de pares.

```
1.  pairs.filter{ case (key, value) => value.length < 20) }
```

3.1.1 Agregaciones

Cuando un conjunto de datos puede expresarse en forma de padres clave-valor, es muy común querer producir estadísticas agregadas de los elementos que

comparten la misma clave. Este tipo de operaciones que combinar valores con la misma clave son transformaciones ya que devolverán un RDD como resultado.

`reduceByKey` es muy similar a `reduce`, con la salvedad de que ejecuta varias operaciones `reduce` en paralelo, una para cada clave del conjunto de datos, y donde cada operación combinan los valores que comparten la misma clave. Bajo estas mismas premisas, Spark también cuenta con la operación `foldByKey`, que en este caso, guarda similitudes con la operación `fold`.

Estas dos operaciones son muy similares al concepto de `combiner` de MapReduce, con el que se especificaba que cada máquina combinada de manera local las claves que contenía la partición sobre la que trabajaba, antes de efectuar el cálculo total de todas las claves del conjunto de datos para todas las particiones del cluster. En este caso el usuario no necesita especificar la utilización de un combiner. Si lo que se desea es personalizar el comportamiento de esta etapa, Spark ofrece un método más general llamado `combineByKey`.

Para ilustrar estas operaciones, se recurre normalmente al ejemplo del conteo de palabras de un gran fichero de texto. En el siguiente ejemplo, podemos ver el resultado, utilizando la operación `reduceByKey`:

Ejemplo 15. Conteo de palabras usando `reduceByKey`.

```
1.  val input = sc.textFile("s3://...")
2.  val words = input.flatMap(x => x.split(" "))
3.  val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

Es importante resaltar que `combineByKey` es la función de agregación por clave más general de todas. Esto significa que la mayoría de los *combiners* están implementados sobre esta función. Igualmente importante, `combineByKey`

permite retornar valores que no sean del mismo tipo que nuestros datos de entrada.

Para entender mejor esta función, debe saberse que `combineByKey` recorre todos los elementos de una partición, y cada elemento tendrá una clave que podrá no haber sido vista anteriormente, o corresponderse con la de un elemento previo, siempre dentro de la partición. Si se trata de nuevo elemento, `combineByKey` aplicará la función que se le pasa, llamada `create Combiner()`, para crear el valor inicial del acumulador de esa clave. Si por el contrario se trata de un valor que ya ha sido visto anteriormente lo que utilizará será la función `mergeValue`, a la que se le pasara el valor actual de la acumulador para esa clave y el nuevo valor.

Como cada partición es procesada de manera independiente, el proceso podrá acabar muy probablemente con varios acumuladores para una misma clave. Por ello, cuando finalmente se mezclan los resultados de cada partición, si dos o más particiones tienen un acumulador con la misma clave sus acumuladores se mezclan utilizando la función provista por el usuario `mergeCombiners`.

Ejemplo 16. Promedio de valores por clave usando `combineByKey`.

```
1. val result = input.combineByKey(
2.   (v) => (v, 1),
3.   (acc, v) => (acc._1 + v, acc._2 + 1),
4.   (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
5. result.map {
6.   case (clave, (suma, cuenta)) => (clave, suma / cuenta.toFloat)
7. }.collectAsMap().map(println(_))
```

La importancia de métodos capaces de agregar valores por clave, como `combineByKey`, reside en que juegan un papel fundamental en la implementación de algoritmos muy importantes como KMeans o Naïve Bayes, en los que es necesario efectuar una operación estadística sobre cada clave. El ejemplo anterior implementa un promedio por clave Y vamos a intentar explicar a continuación los

detalles de este método, en el que la utilización de tres lambdas dificultan su comprensión.

El método `combineByKey` debe producir la suma y la cuenta de valores para cada una de las claves presentes en la variable `input`.

Crear el *combiner*

```
(v) => (v, 1)
```

El primer argumento es una función a la que llamar la primera vez que aparece una clave en el conjunto de datos de entrada. Lo que esta función recibe es el valor de la tupla clave-valor. Como lo que queremos producir al final es un par de la forma `(suma, cuenta)`, el primer paso de esta agregación será emitir un par de la forma (v, 1), donde `v` es el valor es la primera ocurrencia de cada clave, y uno es el valor inicial de la cuenta de elementos asociados a esa clave.

Mezclar un valor (`mergeValue`)

```
(acc, v) => (acc._1 + v, acc._2 + 1)
```

La siguiente función que necesitamos se utilizará cada vez que un *combiner* encuentre un nuevo valor para su clave. En este caso todo lo que necesitamos hacer es sumar el valor de la tupla al acumulador encargado de producir la suma total de todos sus elementos (`acc._1 + v`), y añadir uno al acumulador que cuenta el número de elementos (`acc._2 + 1`).

Mezclar los *combiners* (`mergeCombiners`)

```
(acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
```

Finalmente necesitamos una función que sea capaz de mezclar dos *combiners*. En el ejemplo propuesto, con tuplas de la forma `(suma, cuenta)`, lo único que

necesitamos hacer es sumar el primer y segundo valor de cada acumulador. Recordemos que esta operación mezcla el resultado producido por cada combiner de manera distribuida, pero sólo entre aquellos que operan sobre la misma clave.

Finalmente, para calcular la media de cada clave, si utiliza una función `map` que divide la suma por la cuenta de elementos.

3.1.2 Ajuste del nivel de paralelismo

Hasta ahora se ha hablado de cómo todas nuestras transformaciones se ejecutan de manera distribuida, pero uno se ha explicado como decide Spark cómo dividir el trabajo. Cada RDD tiene un número fijo de particiones que determinan el grado de paralelismo que puede alcanzarse cuando se ejecuta una operación sobre él. Las operaciones de agregación hola agrupamiento utilizar un número específico de particiones que Spark infiere a partir del cluster, pero siempre es posible ajustar el nivel de paralelismo para mejorar el rendimiento.
La mayoría de los operadores vistos hasta ahora acéptame un segundo parámetro especifica dicho número de particiones.

También es posible redefinir cómo se ha efectuado el particionado de un RDD mediante el método `repartition()`, que vuelve a mezclar todos los datos entre los nodos del cluster para crear un nuevo conjunto de particiones. Evidentemente, esta operación es tremendamente cara. Existe además una versión optimizada del método `repartition()`, llamada `coalesce()` que intenta evitar el movimiento de datos pero sólo en el caso en el que el número de particiones decrezca. Para conocer el número de particiones que está empleando un RDD se utiliza el método `rdd.partitions.size()`.

3.1.3 Agrupaciones

Con los datos para los que se puede fijar una clave el caso de uso más común es agruparlos por esa clave –por ejemplo, listar todas las llamadas de un cliente de telefonía. La transformación más común en estos casos es `groupByKey`, que nos devolverá un RDD formado por claves (del tipo `K`), y un `Iterable` del mismo tipo (`V`) que los valores (`RDD[(K, Iterable[V])]`).

Sin embargo, en muchas otras ocasiones nuestros datos no se presentaran en forma de parejas, o querremos agruparlos atendiendo criterios más complejos que la concordancia con una clave. En estos casos se utiliza la transformación `groupBy` en la que se especifica una lambda que se aplicará cada elemento del RDD para determinar la clave en función del resultado.

También es posible agrupar datos que contengan la misma clave provenientes de múltiples RDD, usando una función llamada `cogroup()`. Ilustremos su funcionamiento con un ejemplo.

Ejemplo 17. Ejemplo de funcionamiento de `cogroup()`.

```
1.  scala> val rdd1 = sc.parallelize(Array((1,b), (2,b), (1,b), (3,b)))
2.  scala> val rdd2 = sc.parallelize(Array((1,c), (2,c), (1,c), (3,c)))
3.  scala> rdd1.cogroup(rdd2).collect
4.  res2: Array(
      (1,(CompactBuffer(b, b),CompactBuffer(c, c))),
      (2,(CompactBuffer(b),   CompactBuffer(c))),
      (3,(CompactBuffer(b),   CompactBuffer(c))) )
```

En el ejemplo, se construyen dos RDD de pares formados por claves numéricas (1, 2, 3) y letras como valores asociados a cada clave (a, b, c). `Cogroup()` enlaza los dos RDD a través de la clave que tienen en común (`RDD[(K, (Iterable[V], Iterable[W]))]`), y coloca los datos correspondientes en cada clave en un par de listas iterables con todos los elementos enlazados.

Una aplicación clara de la función `cogroup` es la implementación de `join`, pero también pueden utilizarse para inplementar la intersección por clave. Resta mencionar que `cogroup` puede tratar más de dos RDD en una sola llamada.

3.1.4 Join

Una de las transformaciones más útiles en Spark sobre con juntos de datos con clave es su unión (join). El operador `join` de Spark implementa un `inner join`, o intersección entre conjuntos (aquellos elementos presentes en ambos conjuntos que comparten la misma clave). No bastante, también es posible efectuar la operación de `join` por la izquierda y por la derecha (`leftOuterJoin`, `rightOuterJoin`). El primero de ellos da como resultado un RDD con todas las entradas formadas por las claves del RDD de origen. El valor asociado con cada clave es una tupla formada el valor del RDD de origen, y un `Option` para el valor en el RDD de destino, que nos indica que puede (Some) o no existir (None).

El comportamiento del otro `join` es idéntico al `leftOuterJoin` pero intercambiando los papeles entre el RDD de origen y de destino.

3.1.5 Ordenación

Siempre es posible ordenar un RDD formado por pares clave-valor, si puede definirse una relación de orden sobre las claves. Una vez que los datos hayan sido ordenados, cualquier llamada posterior para su recolección en driver o su escritura en disco producirá un resultado ordenado.

3.1.6 Acciones

Existen acciones específicas que pueden llevarse a cabo sobre los RDD formados por pares clave-valor. Entre las más comunes se encuentran:

- `countByKey`: Cuenta el numero de elementos para cada clave
- `collectAsMap`: Convierte los pares en mapas (Map) que permitan la ejecución más eficiente de funciones de búsqueda.
- `lookup(key)`: Devuelve todos los valores asociados a la clave dada.

3.2 Particionado de Datos

Una de las características más interesantes de Spark guarda relación con como se puede controlar el particionado de los datos entre los nodos que forman un cluster. En un sistema distribuido la comunicación es un recurso caro, por tanto la distribución eficiente de los datos que minimice el tráfico de red puede mejorar notablemente el rendimiento. Del mismo modo que un programa no distribuido debe elegir correctamente las estructuras de datos más eficientes, los programas hechos en Spark pueden llegar a controlar cómo sus RDD son particionados para reducir la comunicación. El particionado no tiene porque ser útil en todas las aplicaciones. Por ejemplo, si un RDD sólo se va a leer una vez no tiene sentido particionarlo por adelantado. Cuando si tiene sentido es cuando un conjunto de datos es utilizado varias veces por operaciones que utilizan sus claves internas.

En Spark es posible la partición de todos aquellos RDD formados por pares clave-valor, de manera que se agrupen los elementos basándose en una función aplicada sobre cada clave. Aunque no se proporciona el control sobre qué claves van a qué nodo, si se permite especificar qué conjunto de claves permanecerán juntas en el mismo nodo.

Considérese por ejemplo una aplicación te mantiene en memoria una tabla con información de usuarios (un identificador de usuario y, por simplificar, otro campo

con información asociada a dicho usuario). Esta aplicación combina periódicamente dicha tabla con un pequeño fichero que representa los eventos asociados a algunos de esos usuarios, en los últimos cinco minutos (por ejemplo, el fichero podría contener el identificador del usuario y la información de en qué enlace ha hecho clic). Podríamos, por ejemplo, querer contar cuantos usuarios han visitado un determinado enlace que no estuviera presente entre la lista de aquellos a los que está suscrito. La aproximación más directa para resolver este problema puede ser la que se gusta en el siguiente ejemplo:

Ejemplo 18. Ejemplo de aplicación en Spark

```
1.  val sc = new SparkContext(...)
2.  val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()
3.
4.  // Funcion que procesa un fichero de logs con eventos
5.  def processNewLogs(logFileName: String) {
6.    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
7.    val joined = userData.join(events)// RDD de (UserID, (UserInfo, LinkInfo))
8.    val offTopicVisits = joined.filter {
9.      case (userId, (userInfo, linkInfo)) =>
10.       !userInfo.topics.contains(linkInfo.topic)
11.   }.count
12.   println("Visitas a enlaces no suscritos: " + offTopicVisits)
13. }
```

En el ejemplo anterior, en la línea 7, se unen los dos conjuntos de datos: el que contiene la información del usuario y es que contiene los eventos generados por el usuario. Para ello se utiliza la clave el identificador de usuario. El resultado es un conjunto de datos en el que el identificador de usuario es la clave para acceder a una dupla que contiene la información preferencias (`userInfo`) Y la información de los enlaces que has visitado (`linkInfo`). En la línea 8, se filtra el conjunto de datos resultantes, quedándose tan sólo con aquellos enlaces que no aparezcan entre sus preferencias (línea 10).

El código del ejemplo es sencillo de interpretar pero a la vez ineficiente ya que cada vez que se llama a la función se realiza una unión (`join`) pero sin ningún

conocimiento de cómo se encuentran particionadas las claves. Por defecto, esta operación calculará el hash de las claves de ambos conjuntos, y enviará los elementos con el mismo hash a la misma máquina del cluster, para unir todos los elementos con la misma clave en ese nodo, de forma local. Como del planteamiento del problema se interpreta que la tabla con las preferencias de usuario es mucho más grande que el fichero de eventos que debe procesarse cada cinco minutos, puede entenderse fácilmente que el código del ejemplo supone un desperdicio tremendo de recursos. Esto es así porque la tabla entera debe procesarse para calcular el hash y después mezclarse por todos los nodos del cluster cada vez que la función invocada.

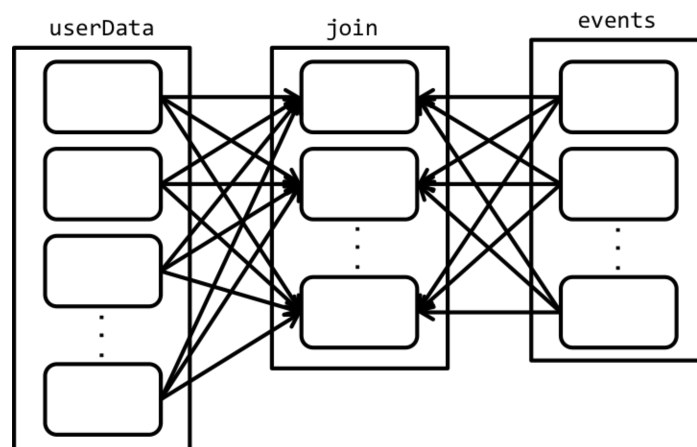

Figura 4

Solucionar este inconveniente es sencillo: debemos utilizar la transformación `partitionBy` para que el hash y el particionado se hagan sólo una vez, al principio del programa.

Ejemplo 19. Particionado creado por el usuario

```
1.  val sc = new SparkContext(...)
2.  val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
3.                   .partitionBy(new HashPartitioner(100))
4.                   .persist()
```

El resto del código del ejemplo no necesita ningún cambio adicional. La principal ventaja de haber introducido el particionado a priori es que cuando llamamos a la función de `join`, Spark ya sabe a qué máquina debe ser enviado cada elemento, reduciéndose así drásticamente el tráfico de red necesario.

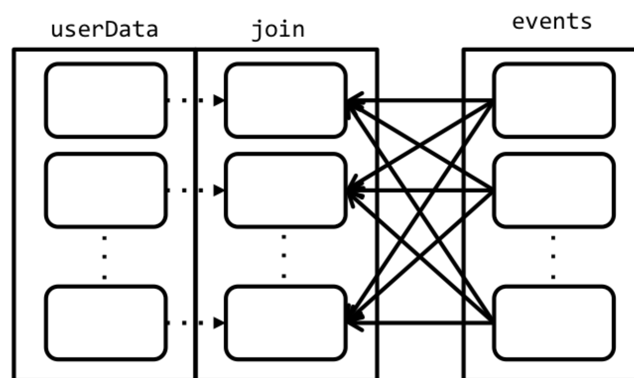

Figura 5

Nótese que `partitionBy` es una transformación y por tanto siempre devolverá un nuevo RDD. Los RDD son invariantes y por ello es importante hacerlos persistentes y accesibles a través de una variable reutilizable (`userData`). El número 100 utilizado para especificar el número de particiones especificará cuantas tareas correrán en paralelo sobre el RDD. Como recomendación general Como este número de veras ser al menos tan grande como el número de núcleos de CPU presentes en el cluster.

El efecto de predefinir las particiones de los datos es que muchas otras cooperaciones también se verán beneficiadas porque esta optimización, como por ejemplo `sortByKey` o `groupByKey`. Sin embargo, si efectuamos una transformación `map()` sobre el RDD, Spark olvidará toda la información de particionado porque el resultado teórico de este tipo de operaciones es modificar la clave de cada registro.

3.2.1 Particionado de un RDD

Es posible averiguar como está particionado un RDD en cualquier momento, consiguiendo el valor de la propiedad `partitioner`. Si ha sido definida, contendrá un objeto del tipo `Partitioner` que es básicamente una función que especifica que partición va cada clave. Conocer Y manejar esta propiedad permite comprobar desde la consola de Spark cómo afectan las distintas operaciones al particionado Y así comprobar que el resultado de cualquier programa es el esperado.

Ejemplo 20. Ejemplo de creación de un conjunto de datos particionado.

```
1.  scala> val pairs = sc.readTextFile("hdfs://pairs.csv").map(_.split(","))
2.  scala> val partitioned = pairs.
                 partitionBy(new spark.HashPartitioner(20)).persist
3.  scala> partitioned.partitioner
    res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5192873d)
```

En el ejemplo anterior, se lee un fichero de pares y se establece que el número de particiones sea 20. Como se quiere utilizar el conjunto de datos en operaciones futuras dentro del mismo programa, si añade la llamada a la función `persist`, de modo que el RDD no tenga que hacer recalculado cada vez. Si no se hiciera esto, las acciones futuras sobre el RDD tendrían que evaluar completamente el valor de `partitioned`, lo que provocaría que tuviera que recalcularse el hash del contenido de `pairs`, además de ser re-particionada cada vez.

3.2.2 Operaciones que afectan o se benefician del particionado

La mayoría de las operaciones que se realizan con Spark implican la mezcla de datos entre los nodos que componen un cluster. A partir de la versión 1.0, las operaciones que se benefician de un óptimo particionado de datos son:

- `cogroup`
- `groupWith`
- `join`
- `leftOuterJoin`

- `rightOuterJoin`
- `groupByKey`
- `reduceByKey`
- `combineByKey`
- `lookup`

De entre las operaciones anteriores, aquellos que operan sobre un único RDD, como `reduceByKey`, hacerlo sobre una estructura pre-particionada provocará que cada clave sea calculada localmente en una única máquina, reduciendo el tráfico de red al envío del valor final calculado localmente desde cada nodo hacia el master.

Por otro lado, para las operaciones que implican dos RDD, como `cogroup` o `join`, el pre-particionado evitará que al menos uno de ellos no sea mezclado y enviado por todo el cluster (*shuffle*). Y si los dos RDD de la operación cuentan con el mismo `partitioner` encontrándose los dos cacheados en la misma máquina, o uno de ellos aún no ha sido calculado, tampoco tendrá lugar el *shuffle*.

Spark conoce internamente cómo afecta cada operación al particionado, y por ello configura automáticamente el objeto `partitioner` de los RDD creados con operaciones que particionan los datos. Por ejemplo, una función `join` sobre dos RDD que vaya seguida de una operación `reduceByKey` se beneficiará de que la localización de las claves resultantes de la primera operación es conocida, provocando que la segunda sea mucho más rápida.

Sin embargo existen transformaciones en las que no puede garantizarse un particionado óptimo y conocido de antemano. Por ejemplo, si se hace `map` sobre un RDD particionado de pares clave-valor, la función que se pasa a la función `map` puede, en teoría, cambiar la clave de cada elemento, por lo que el resultado no podrá contar con un `partitioner`.

Spark no analizar cada operación para comprobar si las operaciones retienen las claves. En su lugar, da la opción de utilizar operaciones que sí las retienen como `mapValues` y `flatMapValues`, que si garantizan que la clave de cada tupla permanece tras la transformación.

3.2.3 Ejemplo: PageRank

Para ilustrar las ventajas de particionar los datos, se mostrará a continuación una pequeña implementación del algoritmo PageRank, que asigna una medida de la importancia de una página en función de cuántas páginas cuentan con un enlace hacia ella. PageRank es un algoritmo iterativo que efectúa muchas uniones entre conjuntos, por lo que constituye un buen ejemplo de las ventajas del particionado de datos.

El algoritmo mantiene dos conjuntos de datos: uno formado por tuplas (`pageID`, `linkList`), en el que `linkList` es la lista de enlaces que contiene una página, y otro con tuplas (`pageID`, `Rank`), con la evaluación que se le da a cada página. La descripción de alto nivel sería como sigue:

1. Inicializa el ranking de cada página a 1.0
2. En cada iteración, cada página `p` envía una contribución `rank(p)/numNeighbors(p)` a sus vecinas (las páginas a las que enlaza).
3. Fija el ranking de cada página a `0.15 + 0.85 * contributionsReceived`.

Los últimos dos pasos se repiten durante un número no determinado de iteraciones necesarias para hacer que converja, aunque típicamente este valor es 10.

El código que implementa el algoritmo en Spark se muestra a continuación.

Ejemplo 21. Algoritmo PageRank en Scala

```
1.  // La lista de enlaces esta guardada en un 'objectFile'
2.  val links = sc.objectFile[(String, Seq[String])]("links")
3.    .partitionBy(new HashPartitioner(100))
4.    .persist()
5.  // Inicializa cada ranking a 1.0
6.  var ranks = links.mapValues(v => 1.0)
7.  // Ejecuta el algoritmo durante 10 iteraciones PageRank
8.  for (i <- 0 until 10) {
9.    val contributions = links.join(ranks).flatMap {
10.     case (pageId, (links, rank)) =>
11.       links.map(dest => (dest, rank / links.size))
12.   }
13.   ranks = contributions.reduceByKey(_ + _).mapValues(v => 0.15 + 0.85 * v)
14. }
15. ranks.saveAsTextFile("rankings")
```

El algoritmo comienza inicializando los rankings al valor 1.0. Lo hace empleando la transformación `mapValues` que produce otro RDD que mantiene el particionado de claves del RDD original (`links`). El cuerpo del algoritmo comienza con una unión entre los rankings y la lista de enlaces por página (recordemos que ambos conjuntos tienen la misma clave, `pageID`), para obtener en un mismo RDD la lista de enlaces y el ranking por página (línea 10, `(pageId, (links, rank))`). A continuación se emplea la transformación `flatMap` para crear un conjunto con la lista de contribuciones a enviar a cada página (línea 11, `(dest, rank / links.size)`). Finalmente, todas las contribuciones son agregadas por clave (línea 13), y se calcula el valor final del ranking de acuerdo a la fórmula expresada al comienzo de esta sección.

Aunque el código en sí es simple, hace varias cosas para asegurarse que los RDD son particionados de forma eficiente, minimizando la comunicación.

1. En cada iteración se hace un `join` con el RDD estático que contiene los enlaces (`links`). Debido a su naturaleza invariante, se particiona en la línea 3, para evitar la operación de *shuffle* por todo el cluster. En la práctica, el

RDD de enlaces tenderá a ser mucho mayor que el que forman los rankings, así que esta optimización evitará mucho trafico de red, si lo comparamos sobre todo con lo que podría ser su alternativa en MapReduce.

2. Por la misma razón se utiliza `persist` en la línea 4, asegurando que se mantiene este RDD en la memoria de cada nodo a lo largo de todas las iteraciones.
3. Cuando se crean los rankings por primera vez en la línea 6, se utiliza `mapValues` en lugar de `map` para preservar el particionado existente en el RDD origen (`links`), de manera que la primera operación de `join` sea mucho más eficiente.
4. En el cuerpo se utiliza la función `reduceByKey` seguida de `mapValues`; dado que el resultado de la primera redefine la ubicación y el particionado de las claves, y la segunda operación no, el siguiente `join` será mucho más eficiente.

3.2.4 Particionado personalizado

Spark proporciona dos funciones de particionado, `HashPartitioner` y `rangePartitioner`. Ambas son adecuadas para la mayoría de los casos de uso, pero en cualquier caso, Spark también permite que la función del objeto `Partitioner` pueda ser adaptada a necesidades especiales. Para ilustrar esta capacidad se modificará el ejemplo anterior para el `Partitioner` tenga en cuenta el dominio de la URL de la página que se quiere evaluar, en lugar de aplicar una función de *hash* no informada sobre la cadena de texto que compone dicha URL.

Para implementar un `Partitioner` a medida debe crearse una clase que herede la clase `org.apache.spark.Partitioner`, que implemente los siguientes tres métodos:

- `numPartitions: Int`, que devolverá el número de particiones que se hayan creado.
- `getPartition(key: Any): Int`, que devolverá el identificador de partición para una clave dada.
- `equals()` que implementará la función de igualdad estándar de Java, y que permitirá a Spark probar este `Partitioner` contra otras instancias del mismo para decidir si dos RDD deben ser particionados de la misma manera.

El código con el `Partitioner` a medida se muestra a continuación:

Ejemplo 22. Partitioner a medida

```
1. class DomainNamePartitioner(numParts: Int) extends Partitioner {
2.   override def numPartitions: Int = numParts
3.
4.   override def getPartition(key: Any): Int = {
5.     val domain = new Java.net.URL(key.toString).getHost()
6.     val code = (domain.hashCode % numPartitions)
7.     if (code < 0) code + numPartitions // no negativos.
8.     else code
9.   }
10.
11.   // Java equals para que Spark compare nuestro Partitioner
12.   override def equals(other: Any): Boolean = other match {
13.     case dnp: DomainNamePartitioner =>
14.       dnp.numPartitions == numPartitions
15.     case _ => false
16.   }
17. }
```

El método `equals` utiliza el operador `match` para comprobar si el otro `Partitioner` con el que se le está comparand es del mismo tipo que este (`DomainNamePartitioner`). Si es así se comprueba si el número de particiones que ambos producen son iguales.

Utilizar el `Partitioner` que se acaba de crear es sencillo, ya que basta con pasarlo como argumento a la función `partitionBy()`.

4 Ejemplo de uso real de Spark

Spark es una tecnología que permite el procesado de grandes volúmenes de datos de forma rápida, entendiendo por "procesado" la aplicación de cualquier método de cálculo que permita la extracción de conclusiones sobre los datos. Dentro de esta categoría, podemos encontrar

- Transformaciones. Se procesan colecciones de, típicamente, logs con el objeto de producir estructuras de datos más compactas que permitan realizar análisis más orientados.
- Agregaciones de datos. Típicamente se recorren grandes colecciones de datos para producir agregados estadísticos que sirvan para monitorizar un conjunto de indicadores clave.
- Análisis. Con el resultado de los dos procesos anteriores se suelen conducir análisis por medio de algoritmos de búsqueda de patrones o de clasificación que permitan explicar comportamientos.
- Predicción. Como estado final, y sobre el análisis de datos en tiempo real es posible efectuar predicciones en las que la ventana de tiempo utilizada puede variar entre unos pocos segundos o un día. Para ello, los algoritmos suelen ser más complejos y deben mantener una información de estado mucho más completa que permita producir una predicción.

Actualmente existen varios proyectos dentro de Telefónica que utilizan Spark para el análisis de colecciones masivas de datos. A continuación se describen a muy alto nivel, algunos de los resultados obtenidos en el proyecto de Vídeo, donde los datos de entrada lo componen las interacciones de los usuarios de televisión con el servicio a través de su descodificador y mando a distancia, además de las colecciones de datos que describen los distintos contenidos que se emiten en cada cadena en cada momento distinto del día.

4.1 Transformaciones

La transformación típica en Spark es la conversión de colecciones de datos en estructuras más compactas, en lo que comúnmente se conoce como fase de ETL, correspondiéndose ésta con la de transformación.

En el caso de vídeo en Telefónica, las sesiones de interacción con el servicio de televisión pueden presentarse de forma muy compleja y siempre con un gran volumen. Por ello, se utiliza Spark para convertir varios formatos de entrada en los que cada línea de un fichero de log, puede ser un heartbit emitido por el servicio cada n segundos, o un *click* en el mando a distancia en lo que se denomina internamente, una sesión de televisión. Estas sesiones intentan comprimir todas esas líneas en una sola que recoja el período de tiempo en el que un usuario determinado ha estado viendo un canal.

De este modo, se consigue pasar de una estructura de datos difícil de correlar a otra en la que el usuario es la clave, y las distintas ocurrencias son los diferentes programas que ha estado viendo. Al hilo del ejemplo que se mostraba en la sección anterior con el algoritmo PageRank, este conjunto de datos voluminoso se

correspondería con los enlaces (`links`), pudiendo componerse de varias decenas de millones de ellas para un solo día.

En el otro extremo están los contenidos emitidos en un único día. Aunque voluminoso, no deja de ser un conjunto de datos varios ordenes de magnitud inferior al de sesiones, por lo que se corresponderá (siguiendo con el símil con PageRank) con el conjunto de rankings.

El caso típico de transformación toma los datos de un repositorio *en bruto*, tipo HDFS, para compactarlos y encajar en un modelo más apropiado para las fases posteriores, que además residirá en un almacén de datos como Hive o HBase. Y todo ello desde un módulo en Spark que realiza todas esas funciones con una única llamada desde consola.

4.2 Agregaciones

Las agregaciones más típicas persiguen el cálculo de audiencias, como caso general. Para ello deben cruzarse (`join`) los conjuntos de datos correspondientes a las sesiones de televisión de toda la planta de usuarios, con los conjuntos de datos que contienen información de los contenidos visualizados. Estos conjuntos auxiliares pueden ser numerosos y servir para identificar nombres y categorías de canales o programas.

Las estrategias para llevar a cabo estas uniones de datos son simples pero deben perseguir ser muy eficientes en tiempo y producir conjuntos de datos agregados muy compactos que puedan servir para alimentar una herramienta típica de visualización de datos.

Ejemplo 23. Cálculo de la audiencia por canal.

```
1.  def channelAudience(sessions:RDD[Sessions]):
2.      RDD[(String, IndexedSeq[Long])] = {
3.    sessions.map { session =>
4.      ((session.channelName, session.userID),watchingUsers(session.interval))
5.    }.reduceByKey(_ + _).map {
6.      case ((channel, _), vector) => (channel, vector)
7.    }.reduceByKey(_ + _)
8.  }
```

El anterior es un ejemplo de cálculo de número de usuarios que ven cada canal, para un conjunto de sesiones dado. Las sesiones se pasan como argumento a la función, y son un RDD con varias decenas de millones de sesiones de televisión correspondientes a todos los usuarios activos durante un período de tiempo cualquiera. Lo que la primero hace la función es crear un tupla nueva en la que la clave es otra tupla compuesta por el nombre del canal y el identificador de usuario. Como valor, se recurre a una función externa que compone un vector en el que se interpreta el intervalo de tiempo que compone la sesión del usuario de tal manera que se devuelve una secuencia de minutos durante los que un usuario ha estado viendo un canal. Esta secuencia es un vector de con tantas posiciones como unidades de tiempo conformen el período para el que se está analizando la audiencia. Si es un día, y la unidad es el minuto, serán 1440 posiciones, en las que tomarán un valor de 1 los minutos en los que se haya estado viendo el canal, y 0 en caso contrario.

La línea 5 suma todos aquellos vectores aportados por un mismo usuario y cadena (las claves), de forma que podamos saber cuanto tiempo ha estado viendo una cadena determinada. En la línea 6, sin embargo, el usuario deja de ser relevante por lo que desparece de la clave, y en su lugar lo que permanece es el identificador del canal. Como valor, tenemos la suma de minutos (línea 7) que han aportado todos los usuarios que han visto ese canal.

En este ejemplo Spark nos proporciona una manera sencilla y rápida de procesar un volumen enorme de información en una cantidad de tiempo muy pequeña (< 1 min. En un cluster de 8 cores). El tiempo de desarrollo de este código es pequeño, si lo comparamos con sus equivalentes MapReduce, y la mantenibilidad mucho mayor.

La resultado es que Spark, hasta ahora, nos ha proporcionado la manera de colapsar conjuntos de datos con un volumen considerable en estructuras más compactas y eficientes, sobre las que efectuar agregaciones sencillas que no sólo son fáciles de componer, si no además poco costosas de ejecutar.

4.3 Análisis

Al igual que se acceden a las funciones de agregación, es posible utilizar MLlib para llevar a cabo análisis más complejos con algoritmos iterativos que permitan efectuar clasificaciones, como KMeans, por ejemplo.

El ejemplo más típico sobre los datos de sesiones de vídeo es el de la búsqueda de clusters que agrupen comportamientos similares de consumo de televisión. Para ello, los datos de entrada se compondrán de la agregación, por usuario de cuantos minutos de televisión de cada tipo de programa o canal se consumen, por unidad de medida temporal (días, semanas, o meses). Este vector de entrada sirve perfectamente para buscar grupos para los que la distancia entre ellos sea mínima, y por tanto podamos decir que presentan un comportamiento similar.

De nuevo, Spark nos proporciona todas las herramientas para llevar a cabo el camino desde el dato en crudo hasta el resultado final, con una única pila de

componentes tecnológicos que además se benefician de la creación de funciones comunes y conocimiento de los datos extremo a extremo.

4.4 Predicciones

Spark incluye a través de su librería de métodos de aprendizaje máquina (MLLib) varios algoritmos de aprendizaje supervisado. Este tipo de algoritmos son utilizados en dos fases: una primera en la que se entrena el algoritmo con los datos para los que se quiere aprender una respuesta determinada a cada uno de los estímulos o señales de entrada; y una segunda etapa en la que se muestran al algoritmo datos nunca vistos anteriormente, para que intente predecir la respuesta asociada a ellos, utilizando para ello la experiencia acumulada en el proceso de aprendizaje.

Las señales o estímulos de entradas son conjuntos de datos que componen un patrón determinado formado por varias variables. Y la respuesta asociada a un patrón no es más que una etiqueta o clasificación de entre varias posibilidades, que permite fijar una categoría para una combinación dada de valores de entrada.

La mayoría de métodos de aprendizaje supervisado incluidos en Spark (Linear regresión lineal, regresión Logística, naïve Bayes, árboles de decisión o SVM lineal) se basan en un método de cálculo conocido como método del descenso del gradiente [6] y descenso de gradiente estocástico [5]. Este último es una versión optimizada del método más general.

Dejando a un lado los detalles de preparación de datos (escalado o normalizado), el entrenamiento de un modelo de regresión logística supone una secuencia de pasos tan sencilla como la que se muestra en el siguiente ejemplo:

Ejemplo 24. Construcción de un modelo basado en regresión logística con Spark.

```
1.  import org.apache.spark.mllib.classification.LogisticRegressionWithSGD
2.
3.  // Construir el modelo de regresión logística
4.  val model_lr = LogisticRegressionWithSGD.train(trainData, numIterations=100)
5.
6.  // Predecir resultados usando el modelo y datos de 'test'
7.  val labelsAndPreds_lr = testData.map { point =>
8.    val pred = model_lr.predict(point.features)
9.    (pred, point.label)
10. }
11. val m_lr = eval_metrics(labelsAndPreds_lr)._2
12. println("precision = %.2f, recall = %.2f, F1 = %.2f, accuracy = %.2f".
      format(m_lr(0), m_lr(1), m_lr(2), m_lr(3)))
```

La línea 4 construye un modelo, y las líneas 7 a 10 evalúan las predicciones obtenidas con el mismo. El resultado final se muestra por consola en la línea 12.

Por tanto, una vez más, la gran ventaja que nos ofrece Spark está en la capacidad de cómputo sobre grandes volúmenes de datos, entendiendo por cómputo en este caso, la utilización de métodos de aprendizaje cuyo uso ha estado normalmente circunscrito a máquinas únicas o a complejos sistemas basados en *grids*, para los que la paralelización de este tipo de métodos era una tarea ardua y compleja.

4.5 SparkSQL

Spark SQL es el modulo ofrecido por Spark para tratar con información estructurada. Es decir, aquellos casos de usos en los que sea necesario interactuar con una base de datos pueden beneficiarse de este modulo. La mayor ventaja, de nuevo, radica en que el código que permite lanzar queries convive con el resto del código y los casos de uso que se han visto hasta ahora.

El resultado de cualquier operación con Spark SQL es una abstracción nueva de Spark llamada DataFrame para la que existe el método `toRDD`, si queremos convertirla al ya familiar RDD. Un DataFrame es una estructura distribuida

construida con RDD en la que se ha tenido especial cuidado en la organización de los datos por columnas.

Por supuesto, casi todo el API disponible para los RDD, también lo está para los *DataFrames*, por lo que el modelo de programación no se resiente. Es entre 10 y 100 veces más rápido que cualquier consulta hecha de forma nativa en Hive, principalmente por el cacheado de datos en memoria, reutilizando los mecanismos heredados de los RDD.

Un ejemplo en el que se combinan las lecturas de datos de Hive con Spark SQL y el entrenamiento de un modelo de regresión logística:

Ejemplo 25. Combinación de Spark SQL y MLLib.

```
1.  val trainingDataTable = sql("""
2.    SELECT e.action, u.age, u.latitude, u.longitude
3.    FROM Users u
4.    JOIN Events e
5.    ON u.userId = e.userId""")
6.
7.  val trainingData = trainingDataTable.map { row =>
8.    val features = Array[Double](row(1), row(2), row(3))
9.    LabeledPoint(row(0), features)
10. }
11.
12. val model = new LogisticRegressionWithSGD()
```

El código del ejemplo anterior muestra perfectamente el potencial de la flexibilidad de Spark al combinar la lectura de un conjunto de datos a través de una *query* SQL, con el entrenamiento de un modelo de aprendizaje supervisado que emplea esos mismos datos.

5 Conclusión

Spark ha surgido con mucha fuerza desde el otoño de 2013 (versión 0.7) para ir desplazando poco a poco a un paradigma que parecía que había llegado para quedarse, que no es otro que Hadoop MapReduce. Sin duda, la ventaja del modelo que mas ha interesado a sus adeptos es su tremenda velocidad de proceso, comparado con MapReduce. Pero en el momento en el que Spark surgió, existían muchas otras alternativas que prometían alcanzar mejoras sustanciales de velocidad (Tez, Impala, Hive 13, etc.,), que sin embargo no acabaron de atraer tanto interés y fuerza de desarrollo, como Spark.

Al margen de las mejoras de velocidad, la adopción inicial de Spark venía acompañada de la necesaria travesía por el desierto del aprendizaje de un lenguaje nuevo: Scala. Este podría haber sido motivo suficiente para despertar recelos y hacer que todo el proyecto fracasará. Sin embargo, un lenguaje relativamente nuevo como Scala, basado en la JVM, que permitía expresar la mayoría de las transformaciones que se querían realizar sobre los datos (en la mayoría de los casos de uso de Big Data) como funciones, sin profundizar en la parte imperativa, es para el autor, la mayor de las ventajas.

Es cierto que Spark puede utilizarse con Python o Java, pero la expresividad que puede alcanzarse con Scala no es posible alcanzarla con sus competidores. La inferencia estática de tipos, la programación no defensiva y la no mutabilidad

exigida por Spark pueden parecer grandes peajes, pero los sistemas para los que se desarrolla en Spark donde el tiempo de computo puede costar dinero el código debe ser tan robusto como sea posible, Scala es la solución. Por otro lado, la extensión con tipos de datos algebraicos [9] también suponen una diferencia importante frente a Python. La expresividad y posibilidades que pueden alcanzarse con el uso de librerías que implementan algebras abstractas para, por ejemplo, agregación de datos, como AlgeBird [8] (de Twitter) es enorme. Con todo, no es necesario concluir aquí cuál es el mejor lenguaje para programar en Spark ya que podemos citar entre sus virtudes la apertura a más de un paradigma. Este hecho también ha influido en su rápida adopción.

Finalmente, lo más obvio es lo más interesante, y quizá por eso es su característica más potente: la paralelización transparente. Para procesar un gran volumen de datos, sobre todo de forma iterativa, Spark oculta todos los detalles involucrados en el paso de mensajes, el modelo de actores, la distribución, etc. Tan sólo es necesario pensar en una única abstracción, los RDD. El API con el que interactuamos con ellos hace creer al programador que trata con colecciones, para las que casi el único requisito es su serialización.

Con todo, Spark es sin duda una alternativa viable y robusta para la construcción de sistemas Big Data extremo a extremo.

6 Referencias

[1] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., & Stoica, I. (2008, December). Improving MapReduce Performance in Heterogeneous Environments. In OSDI (Vol. 8, No. 4, p. 7).

[2] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010, June). Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (Vol. 10, p. 10).

[3] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., ... & Stoica, I. (2012, April). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association.

[4] Databricks. www.databricks.com

[5] Método estocástico de descenso del gradiente. https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[6] Método de descenso del gradiente. https://en.wikipedia.org/wiki/Gradient_descent

[7] Databricks Scala style guide. https://github.com/databricks/scala-style-guide

[8] Algebird. https://github.com/twitter/algebird

[9] Functional Scala: Algebraic datatypes. https://gleichmann.wordpress.com/2011/01/30/functional-scala-algebraic-datatypes-enumerated-types/