INTRODUCCIÓN A APACHE SPARK CON SCALA

UNIVERSIDAD COMPLUTENSE DE MADRID

GRADO EN MATEMÁTICAS, FACULTAD DE CIENCIAS MATEMÁTICAS

ERNESTO VILLANUEVA LÁZARO

Trabajo Fin de Grado

Director: Luis Llana

Madrid, **Julio del 2018**

Autorización de Difusión

Julio de 2018

EL abajo firmante, matriculado en el Grado de Matemáticas de la Facultad de Ciencias Matemáticas, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, tanto el presente Trabajo Fin de Grado: "Introducción a Apache Spark con Scala." y el código, realizado durante el curso académico 2017-2018 bajo la dirección de Luis Fernando Llana Díaz con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo.

Ernesto Villanueva Lázaro

Índice

Agradecimientos

A mi familia, por todo su apoyo.

A María, por su ayuda y consejo.

A Stratio, por la formación que recibí al iniciar mi vida laboral, en especial, a Jorge López Malla, por animarme a resolver ejercicios, incluidos en esta memoria, de registros de vuelos en EEUU a partir de un dataset público.

Resumen

Estamos generando más datos que nunca: según *World Wide Web Size* [1], se llegó a alcanzar en el 2016 la cifra de un Zettabyte, o, lo que es lo mismo, 1.099.511.627.776 GB. En 2017, durante cada minuto se realizaron más de 3.8 millones de búsquedas en Google, se escucharon más de 1.5 millones de canciones en Spotify, se enviaron más de 29 millones de mensajes por whatsapp, 400 horas de video se subieron a Youtube... Frente a estas cifras, el uso de las herramientas tradicionales no resulta el más apropiado para abordar análisis de datos.

Cuando la cantidad de información es demasiado grande para ser tratada en una sola máquina, ha de paralelizarse. Spark es un framework de procesamiento distribuido en memoria que hace uso del paradigma de programación MapReduce para realizar computación distribuida en un clúster. En esta memoria nos familiarizaremos con esta herramienta, aprenderemos cómo trabajar con Spark y cómo Spark opera.

Palabras clave

Spark, Big data, dato, scala, hadoop, Apache, clúster, RDD, Dataset, DataFrames, Shuffle

Abstract

We are generating more data than ever: according to the World Wide Web Size, the figure of a Zettabyte, that is, 1,099,511,627,776 GB, was reached in 2016. In 2017, each minute, more than 3.8 million searches on google, more than 1.5 million songs were listened to on Spotify, more than 29 million messages were sent by whatsapp, 400 hours of video are uploaded to YouTube ... Against this data, the use of traditional tools is not the most appropriate for data analysis.

When the amount of information is too large to be treated by a single machine, it must be parallelized. Spark is a distributed processing framework in memory that makes use of the MapReduce programming paradigm to perform distributed computing in a cluster. In this report, by becoming familiar with this tool, we will learn how to work with Spark and how Spark works

Keywords

Spark, Big data, data, scala, hadoop, Apache, clúster, RDD, Dataset, DataFrames, Shuffle

Capítulo 1

Introducción

La Historia siempre ha tomado como referencia acontecimientos de vital importancia que han introducido cambios fundamentales en la historia de la humanidad; podemos encontrar activos que sobresalen con respecto al resto, algunos tan relevantes para su época que incluso los bautizan: Edad de Piedra, Edad del Bronce, Edad del Hierro. Más recientemente podemos encontrar el carbón, o tras él el petróleo, pero si tuviéramos que destacar unos elementos actuales por encima de otros para señalarlos como el epicentro de nuestra sociedad, serían los datos, fuente de la información, y, por tanto, del poder.

En la economía digital actual los datos constituyen el activo más valioso de muchas empresas. Nos estamos adentrando en la era del Big Data, prueba de esto es que hasta 1.2 ZB de datos IP circularon a través de Internet en 2016 [2] o el crecimiento sin precedentes en la velocidad a la que son producidos; en los últimos dos años se ha creado el 90 por ciento de los datos que hay en el mundo [3].

En este contexto surgen nuevos términos , como "huella digital", empleado para referirnos al rastro que dejamos al navegar e interactuar con la red, que abarca desde qué páginas visitamos, a cuánto tiempo pasamos en ellas y dónde clicamos, nuevas empresas, o alguna ya existente, que han sabido aprovechar la importancia del dato, como Amazon, Netflix, Facebook o Google, nuevas leyes [4] que actualizan las ya existentes para adecuarlas al nuevo contexto, nuevas profesiones dedicadas a explotar y aprovechar el valor del dato, como los *Data Scientist* y los *Data Developer*, así como multitud de herramientas con las que hacerlo, como Hadoop Mapreduce, Apache HBase, Apache Hive, Apache Kafka, Apache Mesos, Apache Pig...

En esta memoria vamos a introducirnos en el framework Apache Spark, herramienta que , como veremos, nos permite manejar grandes cantidades de información de manera rápida y versátil, ofreciéndonos la posibilidad de trabajar con datos estructurados, semi-estructurados o no estructurados, capaz de tratar con multitud de formatos como CSV, JSON, procesar datos en streaming, aplicar machine learning sobre nuestra información o hacer estudios sobre grafos y, además, podremos hacer esto en diferentes lenguajes. La manera que hemos encontrado más adecuada de trabajar con Spark es a través de su lenguaje nativo Scala, y que en los últimos años se ha convertido en el lenguaje basado en JVM más popular tras Java [5].

Para ello, partiremos desde cero, estudiando los conceptos clave para entender cómo trabaja Spark y finalizaremos con unos ejemplos donde poder poner en práctica lo aprendido. En el transcurso de esta memoria nos apoyaremos especialmente en dos libros de referencia, Spark the Definitive Guide [6] y High Perfomance Spark [7], entre otros muchos.

Aunque Spark fue diseñado pensando en ser usado sobre un clúster, para la realización de este trabajo se ha utilizado un ordenador personal.

1.1. Metodología

El trabajo se va a realizar en cuatro fases:

En la primera, presentaremos a Spark, haremos un repaso sobre su historia, los motivos que llevaron a su creación y estudiaremos los componentes que configuran el ecosistema Spark.

En la segunda, estudiaremos las abstracciones que usa Spark para manejar datos, cómo operar sobre ellas y empezaremos a entender el impacto que tiene la manera de organizar nuestro código en Spark.

En tercer lugar, vamos a estudiar qué sucede cuando se ejecuta código en Spark, cómo se organizan sus distintos elementos en un clúster y la manera que tiene de transformar el código de un usuario en un trabajo realizado.

Acabaremos poniendo algunos conceptos en práctica mediante el estudio de vuelos realizados en EEUU.

Capítulo 2

¿Qué es Spark?

Responder a esta pregunta sin divagaciones es posible: Apache Spark es un motor de código abierto de computación en memoria unificado, con un conjunto de bibliotecas para el procesamiento paralelo de datos en clústeres de computadoras. Sin embargo, no sería lícito para su entramado significativo que no nos adentrásemos en tal definición, pues Spark posee varias características relevantes para el mundo de la computación.

Veamos algunas de sus propiedades:

- Es compatible con Python, Java, R y Scala, su lenguaje nativo.

- Utiliza commodity hardware, lo que supone que pueda ser utilizado desde en un ordenador portátil a en clúster de miles de servidores, pues no requiere de computadores avanzados para su utilización.

- Tiene la capacidad de realizar la mayoría de las operaciones en memoria (sin necesidad de escribir a disco). Por esta razón, podemos afirmar que está dotado de mayor rapidez de la que tienen otras opciones, como HadoopMapReduce.

- Ofrece una plataforma unificada que admite una amplia gama de tareas para el análisis de datos, como la carga de datos en crudo, las consultas SQL, el machine learning, el cómputo en streaming o el tratamiento de grafos, todo esto a través del mismo motor

de computación y con un conjunto consistente de APIs.

- La filosofía que persigue Spark con esta visión unificada es la de la combinación de diferentes tipos de bibliotecas y procesamientos dispares, tal y como viene exigiendo el mundo real. Referente a las bibliotecas, cabe decir que además de las bibliotecas estándares, de las que hablamos en el siguiente punto, Spark también admite bibliotecas externas publicadas por las comunidades de código abierto.

- Las bibliotecas estándares de Spark, son la mayor parte del proyecto de código abierto. Pese a que las bibliotecas han crecido para ampliar su funcionalidad, el motor central Spark (Sparkcore) apenas se ha modificado desde la fecha de su lanzamiento.

- Su estructura homogénea permite realizar análisis de manera más simple y más eficiente.

- Es importante señalar que Spark, como motor de computación, no tiene como fin almacenar datos, sino manejar la carga de datos de sistemas de almacenamiento y la realización de cálculos.

- Puede ser utilizado en una gran variedad de sistemas de almacenamiento persistentes, incluidos sistemas de archivos distribuidos, como Hadoop HDFS, buses de mensajes, como Apache Kafka, o sistemas de almacenamiento en la nube, como Azure Storage o Amazon S3.

Capítulo 3

Breve Historia sobre Spark

Spark surge, en primera instancia, en la universidad UC Berkeley en 2009 como proyecto de investigación doctoral de Matei Zaharia dirigido por Ion Stoica. Por aquel entonces, Hadoop MapReduce era el motor de programación paralela dominante. Un año más tarde se publicó un artículo titulado "Spark: Cluster Computing with Working Sets", por Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker e Ion Stoica.

El objetivo perseguido con la creación de Spark era suplir aquellos casos de uso en los que MapReduce resultaba ineficiente. Para ello, trabajaron con usuarios de Hadoop MapReduce con el fin de detectar los puntos fuertes y débiles de esta tecnología, llegando a la conclusión de que debían mejorar los procesos iterativos, aquellos en los que se requieren múltiples pases para procesar los datos y aquellos procesos que requieren una gran cantidad de consultas.

Para solventar estos problemas, Spark se basó en la programación funcional, diseñando de este modo una API sobre un nuevo motor, capaz de realizar cálculos en memoria sobre los datos. Tras esto, su siguiente objetivo fue crear un conjunto de bibliotecas a través de las cuales se pudieran escribir aplicaciones big data con diferentes enfoques usando el mismo framework. Así nacieron MLib, GraphX y Spark Streaming.

En 2013 el proyecto se donó a la Apache Software Foundation. En ese mismo año, Matei Zaharia, Ion Stoica, Ali Godsi, Reynold Xin, Patrick Wendell, Andy Konwiski y Scot Shenker fundaron Databriks, empresa con la que fortalecer el proyecto. Al año siguiente, Databricks obtuvo un nuevo récord mundial [8] en la ordenación a gran escala usando Spark. Actualmente, Databriks desarrolla una plataforma basada en web para trabajar con Spark.

Además, organiza la conferencia más grande sobre Spark: Spark Summit.

Desde su creación, Spark no ha dejado de ir sumando nuevas APIs y librerías con las que ampliar o mejorar su funcionalidad, como las APIs estructuradas o GrapFrame.

Capítulo 4

Componentes de Spark

Spark se basa en un componente principal (Core) sobre el que existen algunos componentes que extienden su uso. Los más destacables son los siguientes:

Spark Core

Como hemos adelantado en las anteriores líneas, se trata del núcleo de Spark, la base del procesamiento paralelo y distribuido. Aquí reside la funcionalidad principal de Spark para gestionar de una manera eficiente la memoria, planificar tareas, recuperarse ante fallos, entre otras posibilidades. Además, es el responsable de todas las funcionalidades de entrada y salida fundamentales. Tiene API en Scala, Java, Python y R. Los distintos componentes de Spark que presentaremos a continuación usan la lógica del Core para adaptarse a sus necesidades.

Es el corazón, en sentido metafórico, de Spark, pues una ventaja o desventaja en él implicará un beneficio o una pérdida en los otros módulos. Además, en este componente se encuentra el API de las colecciones de datos RDD, concepto básico de trabajo en Spark en el que nos adentraremos en el siguiente capítulo.

Spark SQL

Spark SQL es el paquete de Spark para trabajar con datos estructurados. Permite hacer consultas distribuidas a través de SQL, así como la variante Apache Hive de SQL, llamada HiveQueryLanguage (HQL), y admite muchas fuentes de datos, incluidas tablas Hive, Parquet y JSON. Además de proporcionar una interfaz SQL para Spark, Spark SQL permite a

los desarrolladores mezclar consultas SQL con manipulaciones de datos programáticos compatibles con RDD en Python, Java y Scala, todo dentro de una sola aplicación, combinando SQL con análisis complejos. Fue introducido en Spark 1.0

Spark Streaming y Spark Structured Streaming

Spark Streaming es el componente de Spark que permite el procesamiento de streams de datos. Para procesar datos en tiempo real utiliza una secuencia continua de datos de entrada. Ayuda a realizar análisis de transmisión ingiriendo datos en mini-batches, donde pueden ser procesados. Su API es muy similar a la del Sparkcore, facilitando su aprendizaje.

Recientemente, desde Spark 2.2, existe una nueva API de procesamiento en stream, Spark Structured Streaming. Está construida sobre el motor de Spark SQL, utiliza las API estructuradas y pretende ofrecer una mejora importante en cuanto a rendimiento y uso con respecto a su predecesor.

MLlib

Se trata de la biblioteca que contiene la funcionalidad de machine learning. Proporciona múltiples tipos de algoritmos de aprendizaje automático, así como funcionalidades de soporte tales como evaluación de modelos e importación de datos. Todos sus métodos están diseñados para escalar a través de un clúster.

Graphx y GraphFrames

Graphx es el motor de cálculos en paralelo de grafos de Spark. GraphX también proporciona varios operadores para manipular grafos (por ejemplo, subgraph y mapVertices) y una biblioteca de algoritmos de grafos comunes que veremos en los casos de uso.

A partir de Spark 1.4, Spark cuenta con GrapFrames, un motor nuevo para cálculos sobre grafos. Se trata de un paquete que extiende la funcionalidad de GraphX, debido a que esta usa DataFrames en lugar de RDD, por lo que aprovecha la optimización de estos, además de extender los lenguajes con los que puede ser usado (Scala, Java y Python).

Gestor de recursos

Spark está diseñado para escalar en un clúster, y de manera eficiente, de uno a muchos miles

de nodos. Spark admite tres administradores de clústeres distintos: Hadoop YARN, Apache Mesos y StandaloneCluster Manager, lo que maximiza su flexibilidad.

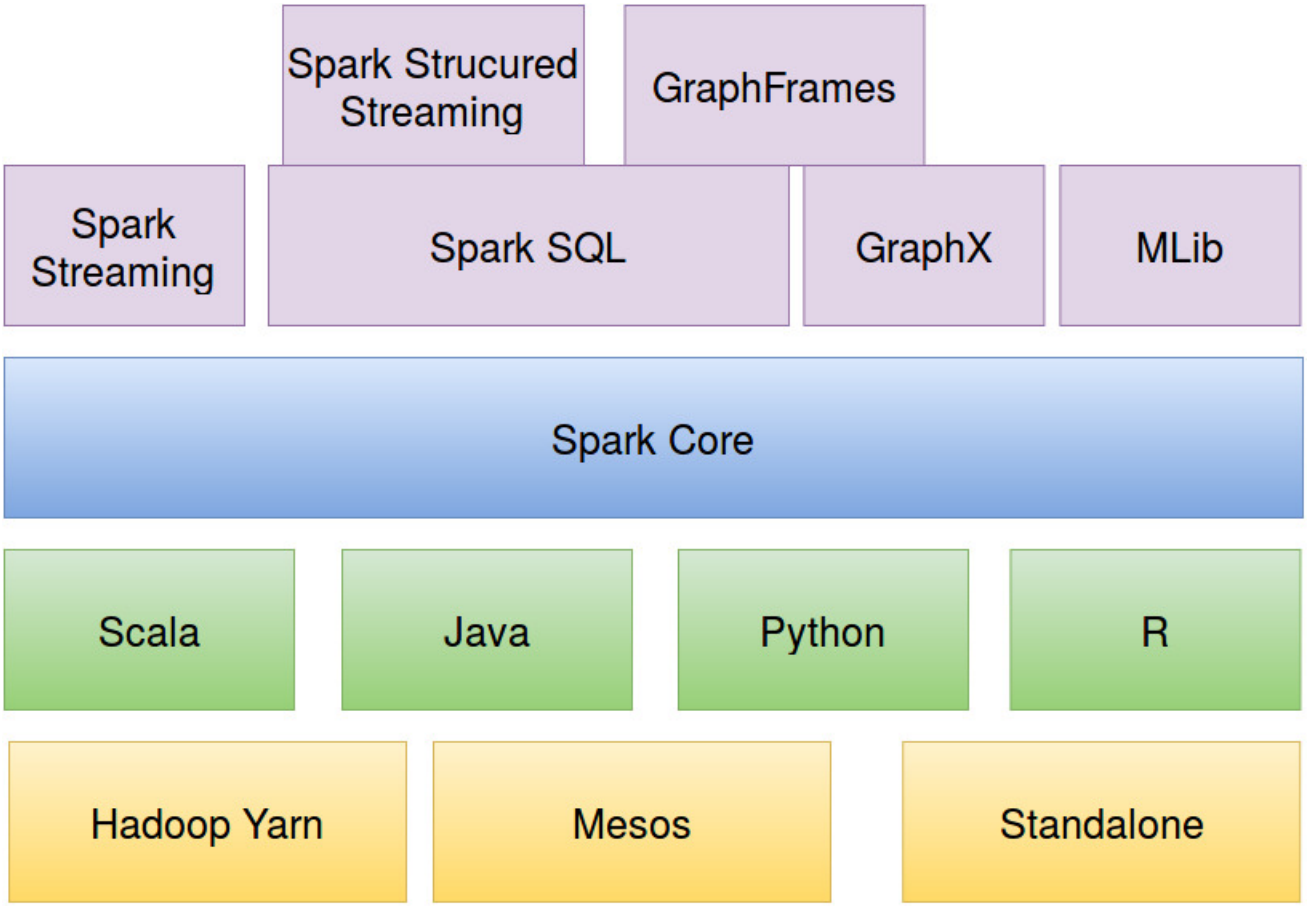

Figura 4.1: Ecosistema de Spark

4.1. Spark shell

Se trata de una consola de lenguaje interactiva con la que poder trabajar de una manera rápida y sencilla con Spark. Está disponible en Scala y Python.

4.2. Spark UI

Es una interfaz web con la que podemos monitorizar nuestras aplicaciones en Spark.

Gracias a todo este ecosistema de librerías y aplicaciones podemos considerar a Spark como una compleja y completa herramienta con la que podemos trabajar en campos muy diversos dentro del Big Data, ofreciendo además versatilidad a la hora de elegir el lenguaje con el que hacerlo.

Capítulo 5

RDD

Los Resilient Distributed Datasets, RDD, si atendemos a sus siglas, son colecciones de datos distribuidos, de tipado estático, inmutables y de evaluación perezosa usadas por Spark para realizar trabajos cuyo cometido es ir realizando una sucesión de funciones sobre estos RDD con el fin de obtener un resultado.

Con estas líneas nos pueden asaltar no pocas cuestiones. ¿A qué tipo de colecciones de datos nos referimos? ¿Dónde se distribuyen? ¿Qué quiere decir que son inmutables? ¿Y de evaluación perezosa? Esclarezcamos esta definición de RDD descomponiéndola para asimilar así mejor su contenido:

Las colecciones de datos que forman los RDD no son solo cualquier tipo de objeto de Python, Java o Scala, sino que además pueden ser objetos definidos por el usuario. Los datos que conforman el RDD se dividen en particiones que son repartidas entre los diferentes nodos esclavos del clúster. Con inmutables nos referimos a que son solo de lectura, puesto que no se prestan a modificaciones, dado que los RDD van siendo creados a partir de la transformación de otro RDD, de la distribución de una colección de objetos, como puede ser un array, en el driver o de la lectura de datos en memoria, o bien, en algún otro tipo de almacenamiento. Para finalizar este desglose léxico, atendemos al término "evaluación perezosa". Con él, afirmamos que los RDD son calculados únicamente cuando los datos finales deben computarse, esto es, cuando sobre ellos recae una acción, como veremos más adelante.

5.1. RDD de pares clave valor

Son un tipo especial de RDD [9], compuestos por pares de clave - valor (key-value), lo cual significa que están formados por una lista de tuplas cuyo primer elemento corresponde a la clave y el segundo es el valor que se le asocia a la clave. Podrían ser comparados, pues, con diccionarios o mapas en otros lenguajes de programación, como Python o Java. Spark habilita muchas funciones específicas para trabajar con este tipo de RDDs, ya que le posibilitan actuar en cada clave en paralelo, o reagrupar datos en la red.

5.2. Propiedades de los RDD

Ahora que ya hemos estudiado la definición de los RDDs en general, veamos las cinco propiedades internas [10] que caracterizan un RDD en particular: la lista de objetos de partición, la lista con las dependencias de los RDDs padres, una función, una ubicación y el particionador:

- La lista de objetos de partición que componen el RDD puede ser consultada mediante la función *partitions()*, que devuelve un array con los objetos de partición, que componen las partes del conjunto de datos distribuidos.
- La lista con las dependencias de los RDDs padres, la cual puede ser consultada mediante la función *dependencies()*. Las dependencias pueden ser de dos tipos: narrow o wide. Las primeras corresponden a particiones que dependen de un pequeño subconjunto de particiones del padre, mientras que las segundas son aquellas en las que la partición ha sido creada mediante una reorganización de todos los datos del padre.
- Una función para calcular los elementos de una partición p, dados iteradores para sus particiones padres *iterator(p, parentIters)*. No es común que un usuario llame directamente a esta función. Lo normal es que sea usada por Spark cuando se calculan acciones.
- Una ubicación preferencial de cada partición. Para consultar la localidad de los datos en una partición p podemos usar la función *preferredLocations(p)*, que devuelve una secuencia de string que nos informa sobre cada uno de los nodos donde se almacena la partición p.
- Un particionador. Mediante la función *partitioner()* obtenemos información acerca de si en el RDD existe una relación entre los datos y la partición asociada a ellos, como un

hashPartitioner. *partitioner()* devuelve un optiontype de Scala, siendo None el resultado en aquellos RDD que no sean de tipo Clave Valor.

Las tres primeras en conjunto constituyen la ruta desde un RDD hasta su RDD raíz, se conocen como linaje y suponen que cada partición de los datos contenga la información necesaria para ser recalculada, haciendo a Spark tolerante a fallos. Las otras dos son opcionales y se utilizan como optimizadores.

5.3. Funciones sobre RDDs

Las operaciones que pueden realizarse sobre RDDs se engloban en dos tipos: transformaciones y acciones.

Las acciones son aquellas operaciones que devuelven algo que no es un RDD. Devuelven información al driver o escriben datos en sistemas de almacenamiento externo. Dado el carácter perezoso de los RDD, son necesarias para la evaluación de un programa de Spark. Algunas acciones mandan la información al driver, por lo que el resultado de cualquier acción debe caber en su memoria. Por este motivo, es preferible evitar acciones que devuelvan el total de los datos al driver, como *collect()*, y usar en su lugar otras que retornen una cantidad elegida por el usuario, como *first()*, *take(n)*, *count*, etcétera.

Las transformaciones son las operaciones que tienen como resultado un RDD, suponen el concepto básico de trabajo en Spark y constituyen la mayor parte de la potencia de la API de Spark. Dado que los RDD son inmutables y están tipados estáticamente, si intentamos realizar una transformación, por pequeña que sea, en un RDD nuestro RDD original no se verá afectado, lo que obtendremos es un nuevo RDD con una nueva definición de sus propiedades. Además, son calculadas de manera perezosa, por lo que su cómputo será ejecutado cuando al RDD resultante se le aplique una acción. Por lo general, las transformaciones tienen un comportamiento a nivel de elemento, actuando de uno en uno de manera paralela en las diferentes particiones de un RDD.

Las transformaciones se dividen en dos categorías: transformaciones con dependencias *narrow* y transformaciones con dependencias *wide*. Para un buen desarrollo de aplicaciones en Spark, es de vital importancia entender las diferencias entre estas dos categorías, ya que una aplicación construida sin tenerlas en consideración supondrá un impacto muy negativo a nivel

de eficiencia.

Las transformaciones con dependencias *narrow* son aquellas en las que los datos del RDD padre (el RDD sobre el que se aplica la transformación) no necesitan ser mezclados a nivel de partición para calcular el RDD hijo (el RDD resultante de aplicar la transformación al RDD padre), por lo que pueden ser ejecutadas en un subconjunto de datos sin necesidad de información acerca del resto de particiones. De esta manera, el RDD hijo tendrá un número finito de dependencias en las particiones del RDD padre. Transformaciones que se encuentran en esta categoría son *map*, *flatMap*, *filter*.

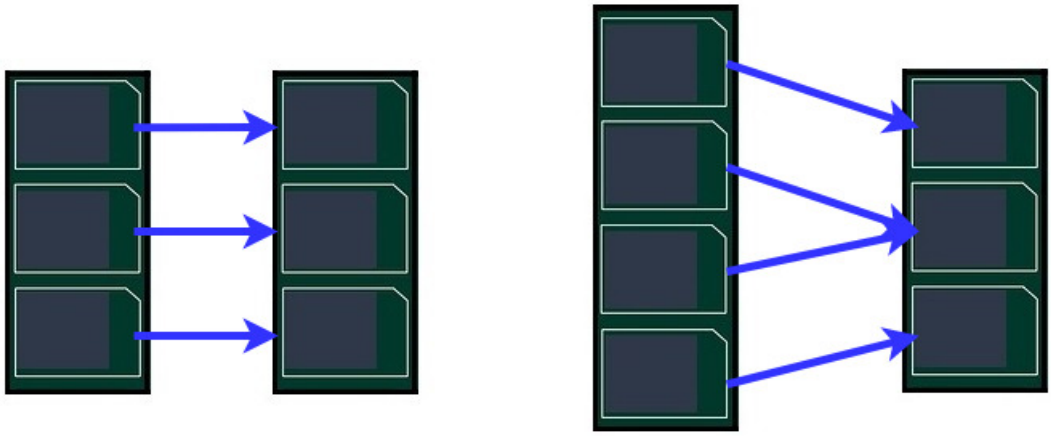

Figura 5.1: Transformaciones con dependencias narrow

Las transformaciones con dependencias *wide*, por el contrario, son aquellas que requieren un particionado particular de los datos, siendo necesaria la consulta de datos en las distintas particiones del RDD padre, implicando una mezcla en los datos de dichas particiones para computar el RDD hijo; esta operación de compartición de información a través de distintas particiones es conocida en Spark como shuffle, y la estudiaremos detenidamente más adelante. Ejemplos de estas transformaciones son *sort*, *reduceByKey*, *gruoupByKey*.

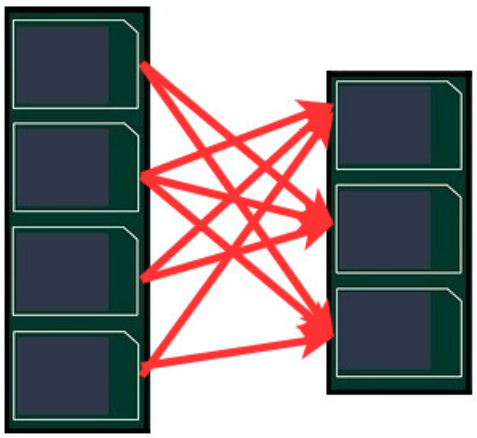

Figura 5.2: Transformaciones con dependencias wide

5.4. APIs estructuradas

Hasta ahora, cuando hemos hablado acerca de las estructuras de datos en Spark lo hemos hecho en términos de RDD. No obstante, hoy por hoy no es la herramienta más utilizada. En este apartado vamos a hablar de los DataFrames y los Datasets, que constituyen las APIs estructuradas de Spark, que son las más usadas en la actualidad.

Podemos pensar en los DataFrames y en los Datasets como una tabla distribuida de datos con filas y columnas. El esquema, propiedad inherente a estos, es la lista que define las columnas y los tipos dentro de estas columnas. El usuario puede definirlo, o puede ser leído por una fuente de datos. Los DataFrames y los Datasets comparten algunas semejanzas con los RDD, como por ejemplo, su inmutabilidad y su evaluación perezosa.

5.4.1. Datasets

Están presentes en el ecosistema Spark desde Spark 1.6. Representan conjuntos tipados de datos. En ellos, la comprobación del tipado se realiza en tiempo de compilación. Sólo están disponibles para los lenguajes basados en JVM, es decir, para Scala y para Java, dado que

los tipos que admite son los tipos de Java o case class de Scala.

5.4.2. Los DataFrames

Se trata de la API estructurada más usada actualmente y aparecieron en la versión Spark 1.3. Se suele decir que se trata de conjuntos no tipados, lo cual es incorrecto. La comprobación del tipado especificado se realiza en tiempo de ejecución. Internamente, Spark trata a los DataFrames como Datasets de tipo *row*, que permite liberarse de los costes de recolección de basura y creación de instancias de objetos que implican los tipos de JVM. Están disponibles en cuatro lenguajes: Java, Python, Scala y R.

Capítulo 6

Arquitectura de Spark

El término "arquitectura" en informática remite a la estructura lógica y física de los componentes de una computadora. Si nos detenemos a estudiar la arquitectura de Spark, la primera característica identificativa es su uso de la conocida arquitectura maestro-esclavo o maestro-trabajador, que, como bien indica su terminología, consta de un nodo maestro y muchos nodos esclavos.

La arquitectura de Spark se compone de dos procesos: el driver y los executors. Ambos serán descritos más adelante, pero aclaremos por ahora que el driver es el coordinador central, el cual se comunica con el cluster manager, encargado de orquestar los nodos esclavos, donde corren los executors y que, además, el driver y los executors se ejecutan en sus propios procesos de Java.

Para ir dibujando nuestra arquitectura necesitamos preguntarnos cómo se ponen en funcionamiento las aplicaciones en Spark. El proceso es el siguiente: Las aplicaciones de Spark se ejecutan como conjuntos de procesos independientes en un clúster, coordinados por el objeto SparkContext en su programa principal, el driver. Cuando ejecutamos Spark en un clúster, SparkContext se conecta al cluster manager, encargado de asignar recursos en el clúster. Una vez conectado, Spark adquiere executors en los nodos del clúster, que son procesos que ejecutan cúlculos y almacenan datos para su aplicación. A continuación, envía su código de aplicación (definido por archivos JAR o Python pasados a SparkContext) a los esclavos. Por último, SparkContext envía tareas a los esclavos para que se ejecuten.

Spark puede ejecutar varias aplicaciones en un clúster al mismo tiempo. Las aplicaciones están programadas por el administrador del clúster (cluster manager) y corresponden a un SparkContext. Cada aplicación obtiene sus propios agentes executors, que permanecen activos durante toda la aplicación y ejecutan tareas en varios subprocesos. Este hecho tiene la ventaja de aislar aplicaciones entre sí, tanto en el lado de la programación (cada driver programa sus propias tareas), como en el lado del executor (las tareas de diferentes aplicaciones se ejecutan en diferentes Máquinas Virtuales Java). Sin embargo, también significa que los datos no se pueden compartir en diferentes aplicaciones Spark sin escribirlo en un sistema de almacenamiento externo.

6.1. Driver

Como hemos señalado antes, el driver es el coordinador central, ya que un programa Spark se ejecuta en el nodo del driver y este envía instrucciones a los executors. Es un proceso JVM que ejecuta la función main() de la aplicación y crea el SparkContext para una aplicación de Spark. También aloja el DAG Scheduler y el Task Scheduler, de los que hablaremos no muy tarde.

El driver debe ser direccionable a través de la red desde los nodos esclavos, en tanto que es el que escucha y acepta las conexiones entrantes de sus executors a lo largo de su vida útil. Además, el driver es el encargado de traducir el código de usuario en un trabajo específico y de crear tareas convirtiendo aplicaciones en pequeñas unidades de ejecución.

6.2. Executors

Son agentes distribuidos responsables de la ejecución de las tareas, siendo ellos los que procesan los datos, los mantienen en memoria o los almacenan en disco.

Cada aplicación tiene sus propios executors. Los executor se ejecutan durante toda la vida de una aplicación de Spark. Pueden ejecutar múltiples tareas a lo largo de su vida, tanto de forma secuencial como paralela. Suelen ser de asignación estática. No obstante, en caso de

que así lo requiramos, podemos configurarlos para hacer la asignación dinámica.

También informan de las métricas parciales mediante el uso del hilo del transmisor Heartbeat. Cuando un executor se inicia, se registra con el driver, comunicándose directamente con él para ejecutar tareas.

Sobre la ubicación de los executor, cabe señalar que residen en los nodos esclavos del clúster, que son cualquier nodo que pueda ejecutar código de aplicación en el clúster. Cada executor es su propia JVM, y un executor no puede abarcar múltiples nodos, aunque un nodo puede contener varios executors.

6.3. SparkContext

El SparkContext es una variable de entorno que configura los servicios internos y establece una conexión con un entorno de ejecución de Spark. Es decir, con el clúster. Una vez que se crea un SparkContext, puede usarlo para crear RDD, acumuladores y variables broadcast, acceder a los servicios de Spark y ejecutar trabajos (hasta que se detenga SparkContext). Proporciona acceso al clúster a través del cluster manager. SparkContext es esencialmente un cliente del entorno de ejecución de Spark y actúa como el maestro de su aplicación.

SparkContext determina cuántos recursos se asignan a cada executor. Como veremos más adelante, cuando se lanza un trabajo de Spark, cada executor tiene ranuras (slots) para ejecutar las tareas necesarias para calcular un RDD. De esta forma, podemos pensar en un SparkContext como un conjunto de parámetros de configuración para ejecutar trabajos de Spark. Estos parámetros (por ejemplo, la URL del nodo maestro) están expuestos en el objeto SparkConf, utilizado para crear un SparkContext.

6.4. DAG Sheduler

El DAGSheduler, a quien ya habíamos introducido, es el responsable de manejar la ejecución de una aplicación en Spark, dividiéndola en trabajos, etapas y tareas. Estos conceptos los

estudiaremos detenidamente más adelante para así centrarnos a lo largo de este subapartado en comprender cómo actúa el DAGScheduler de Spark y por qué este comportamiento supone una mejora con respecto a otros sistemas.

Se trata de la capa de planificación de alto nivel [11] que implementa la programación orientada a la etapa. Lo hace calculando un DAG (directed aciclid graph) de etapas para cada trabajo, realizando un seguimiento de los RDD, a la par que busca un cronograma mínimo para ejecutar el trabajo. Por lo tanto, es el encargado de transformar un plan de ejecución lógico (es decir, el linaje de RDD de dependencias) en un plan de ejecución físico. Además de crear un DAG de etapas, DAGScheduler determina las ubicaciones más apropiadas para ejecutar cada tarea, teniendo en cuenta el estado actual de la memoria caché, evitando así recalcular RDD que hayan sido cacheados. También se ocupa de borrar las estructuras de datos una vez hayan finalizado las tareas que dependen de ellos, optimizando así el uso de memoria. Por último, bajo su responsabilidad está la recuperación ante fallas debidas a la pérdida de archivos durante el shuffle. Lo hace volviendo a computar las etapas afectadas en las particiones perdidas.

Este modelo es una generalización del modelo MapReduce, el cual crea un DAG con dos estados, Map y Reduce. El cambio de etapas en un DAG en Spark viene marcado por una transformación con dependencias wide, por lo tanto, puede canalizar transformaciones con dependencias narrow juntas, sin necesidad de escribir a disco entre cada una, obteniendo así un mejor rendimiento que el de MapReduce, que necesita escribir en disco los resultados entre las etapas Map y Reduce.

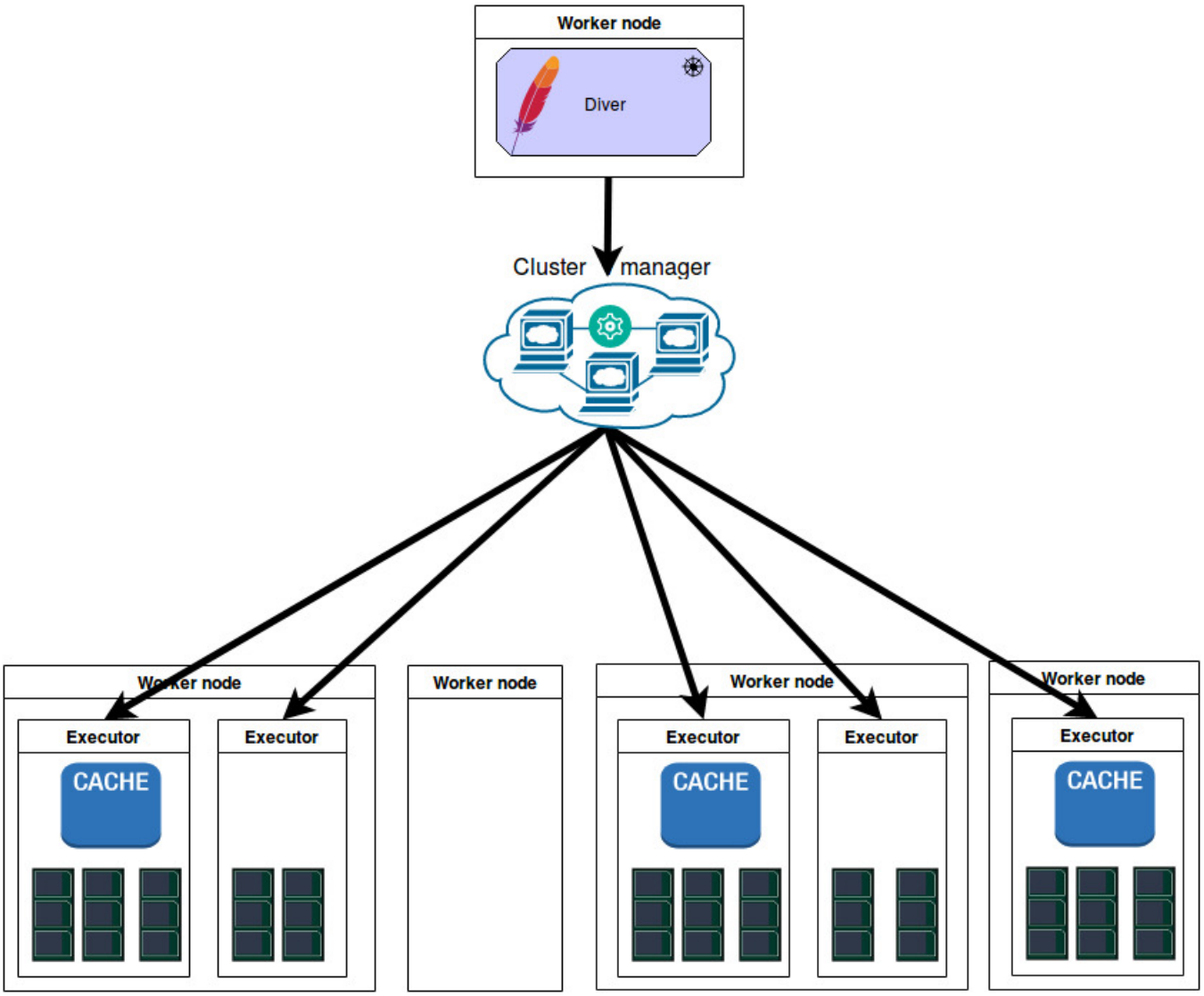

Figura 6.1: Arquitectura de Spark

Capítulo 7

Cómo opera Spark

En este apartado vamos a tratar qué sucede internamente cuando estamos trabajando con Spark. Para iniciarnos en dicha materia debemos familiarizarnos antes con algunos conceptos propios de Spark.

7.1. Tareas

Las tareas son unidades de trabajo individuales , cada una enviada a una máquina, es decir, una tarea no se puede ejecutar en más de un executor, cada tarea atacará a una partición específica de los datos. Dicho de otra manera, una tarea es simplemente una unidad de cálculo aplicada a una unidad de datos (partición). Particionar los RDD en un mayor número de particiones significa que se pueden ejecutar más en paralelo, siempre y cuando el clúster tenga recursos suficientes. Spark no puede ejecutar más tareas a la vez que el número de núcleos por executor por la cantidad de executors.El número de tareas por etapa corresponde al número de particiones en el RDD de salida de esa etapa.

Cada tarea realiza internamente los mismos pasos:

1. Obtener su entrada, ya sea desde el almacenamiento de datos (si el RDD es un RDD de entrada), un RDD existente (para datos ya almacenados en caché), o salidas de shuffle.

2. Realizar la operación necesaria para calcular los RDDs que representa.

3. Escribirá la salida de shuffle, en almacenamiento externo o de vuelta al driver (si es el RDD final de una acción como *count()*).

El conjunto de tareas producidas por una transformación con dependencias *narrow* se denomina etapa.

7.2. Etapas

Para ser más eficiente, Spark agrupa las operaciones que pueden ser aplicadas en una sola partición, y, consecuentemente, en una única máquina sin que sea necesaria comunicación en red entre los nodos esclavos, esto es, cuando no se requiere información de otras máquinas. Es decir, canaliza juntas las transformaciones con dependencias *narrow*, mientras que las transformaciones con dependencias *wide* suponen el límite entre una etapa y otra.

En consecuencia, entre una etapa y el siguiente se necesita comunicación con el driver, por lo que se han de ejecutar en secuencia. Sin embargo, en casos donde no se requiera información de otra etapa, por ejemplo, aplicar operaciones a dos RDD sin ningún padre común, se pueden calcular en paralelo.

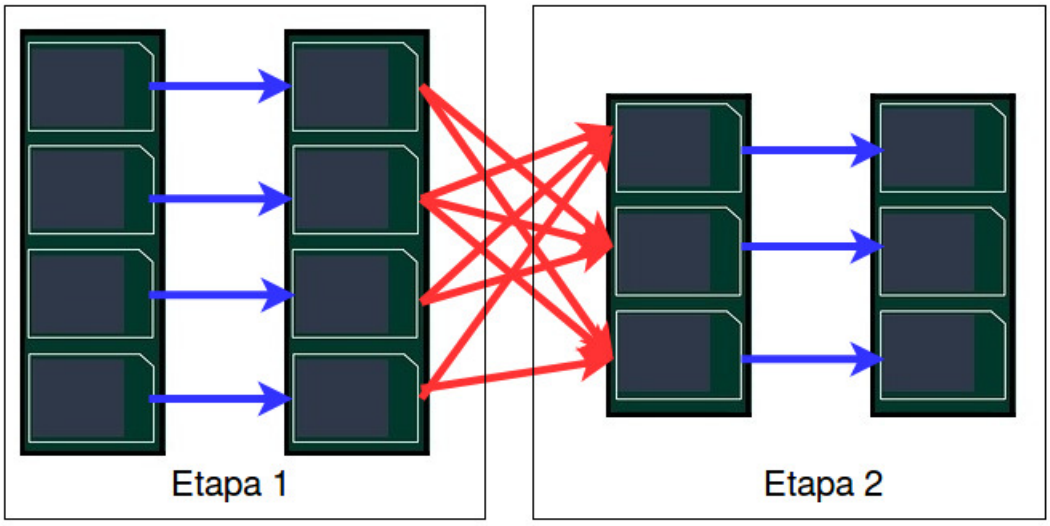

Figura 7.1: Límite entre dos etapas

Podemos clasificar las etapas en dos categorías diferentes, según lo que marque su final:

- ShuffleMapStage, que escribe los archivos de salida de shuffle.
- ResultStage, para la etapa final que ejecuta una acción.

Las agrupaciones de etapas fruto de una acción se conocen como trabajo. Las etapas suelen compartirse en varios trabajos, si estos trabajos reutilizan los mismos RDD.

7.3. Trabajo

Suponen el elemento más alto de la jerarquía de ejecución de Spark. Cada trabajo de Spark corresponde a una acción, y suponen el desencadenante para comenzar el cómputo sobre los RDD, debido al carácter perezoso de estos.

7.4. Aplicación

Por último, definimos a cada uno de los programas en los que se crea un SparkContext como una aplicación de Spark. Entre diferentes aplicaciones no se comparte memoria ni recursos.

Ahora que ya conocemos las distintas fases por las que pasa toda aplicación de Spark, falta por abordar una cuestión. Como hemos visto, la memoria entre etapas no puede ser compartida por no estar en la misma máquina virtual, por lo que nos surge la pregunta de cómo compartir información entre etapas. La respuesta sería escribiendo a disco, operación conocida como shuffle.

7.5. Shuffle

Según la documentación oficial de Spark, el shuffle es "el mecanismo de Spark para redistribuir los datos de manera que se agrupen de manera diferente en las particiones". Es conveniente aclarar que no todo movimiento de datos por el clúster es considerado como shuffle. Las acciones, escrituras o lecturas de una fuente externa no se consideran shuffle. La fase de shuffle representa una repartición física de los datos, esto es, conlleva escritura en disco. Esto contradice la definición inicial que dimos de Spark como motor de computación en memoria, pero se trata de momentos concretos en los que debido a una transformación con dependencias *wide*, es decir, cuando necesite combinar los datos de distintas etapas, o por saturación de memoria, en vez de mantener los datos cacheados los escribe temporalmente a disco, tras lo cual se genera una nueva etapa con un nuevo conjunto de particiones.

Su comportamiento varía dependiendo del Shuffle Manager que usemos y cada uno tendrá su Caso de Uso. Tratar los diferentes shuffle manager va más allá de lo que pretendemos abordar en este texto, por lo que únicamente nos centraremos en las propiedades generales.

Operaciones que pueden causar shuffle son aquellas que implican una repartición de los datos entre particiones, como repartition y coalesce, operaciones *ByKey*, excepto *countByKey* y operaciones de mezcla de RDDs como *cogroup* y *join*.

Se trata de una operación muy costosa, pues debe leer todas las particiones para encontrar

todos los valores de todas las claves, y luego unir los valores de las particiones para calcular el resultado final de cada valor, esto conlleva escritura y lectura en disco, serialización de datos y movimiento de entrada y salida por la red [12].

La fase de shuffle se desarrolla en dos partes: map y reduce. En la parte de map, Spark genera archivos intermedios que son escritos en disco en la ruta local de cada nodo. Estos archivos son conservados hasta que los RDD a los que corresponden ya no se utilizan, de esta manera, en caso de fallo no es necesario que vuelvan a ser calculados. Cada tarea map escribe un archivo por cada tarea Reduce. El directorio de almacenamiento temporal en disco puede ser configurado en la propiedad spark.local.dir que se configura a nivel de application. Además, spark brinda la opción de comprimir los archivos de salida Map, especificados por el parámetro *spark.shuffle.compress*.

Un efecto de lado, provocado por esta persistencia en disco durante la fase de shuffle, es que la ejecución de un nuevo trabajo sobre los datos que ya se han pasado por la fase shuffle, no vuelve a ejecutar la parte map del shuffle. Debido a que los archivos shuffle ya se habían escrito anteriormente en disco, Spark sabe que puede usarlos para ejecutar únicamente la parte reduce del shuffle. Pese a esta optimización automática, para un mejor rendimiento es preferible realizar su propio almacenamiento en caché con el método caché, que le permite controlar exactamente qué datos guardar y dónde.

Durante la fase de reduce, Spark requiere que todos los datos quepan en memoria. Cuando la memoria requerida de cada tarea reduce excede la asignada, se emite una excepción de falta de memoria y se debe interrumpir todo el trabajo. Para evitar quedarse sin memoria, se debe especificar un valor suficientemente alto para la cantidad de tareas reduce.

Llegados a este punto ya estamos en disposición de poder analizar qué sucede internamente durante una aplicación de Spark:

El código escrito por un usuario define un DAG de RDD. En primer lugar, Spark comprueba si el código escrito por el usuario es válido. Tras esto, Spark lo convierte en un flujo lógico de operaciones, el plan lógico [13] (también conocido como DAG de dependencias). Cuando se realiza una acción este plan lógico se pasa a través de SparkContext al DAG Scheduler, que lo usa para crear un plan de ejecución físico, el DAG de etapas, que consiste en tareas agrupadas en etapas, las cuales son enviadas como TaskSets a una implementación TasScheduler subyacente, que es el encargado de enviar las tareas para su ejecución en el clúster a través

del cluster manager [14]. En este punto Spark ejecuta este plan en el clúster.

Spark cuenta con mecanismos internos para optimizar tanto el plan lógico como el plan físico, y también realiza optimizaciones en tiempo de ejecución. Estos difieren en función de si estamos tratando con las APIs estructuradas o con RDDs. En cualquier caso, el plan físico siempre es ejecutado sobre RDDs.

Capítulo 8

Ejemplos con spark

En esta sección abordaremos la parte práctica sobre Spark. El proyecto ha sido construido usando Apache Maven y, como ya se ha dicho antes, usaremos Scala (ver. 2.11.12) y trabajaremos con la última versión de Spark (Spark-Core 2.3.1, Spark Sql 2.3.1, Spark Graphx 2.3.1 y Spark GraphFrames 0.5.0).

Los distintos ejemplos tratados en las siguientes líneas suponen una extracción del total realizado para este trabajo; el resto pueden ser descargados del siguiente repositorio de github: https://github.com/ErnestoVLUCM/Origin/tree/develop

8.1. Introducción a Scala

Antes de afrontar los distinto ejemplos de Spark que trataremos en el siguiente bloque, resulta imperativo ver unas nociones básicas acerca del lenguaje Scala. Se trata de un lenguaje creado por Martin Odersky en 2003. Es un lenguaje puro de programación orientada a objetos [15], lo que implica que todas las variables son tratadas como objetos, y los operadores como métodos. Además, es un lenguaje de programación funcional. El código de Scala se ejecuta sobre JVM y es posible el uso de librerías de Java.

Cuando programemos en Scala, por lo general lo haremos usando variables inmutables, usando la sentencia val para declararlas, aunque es posible trabajar con variables mutables, usando la sentencia var; veamos cómo hacerlo:

```
//We define an immutable String
val hello = "hola"

//We define a mutable String specifying the typing
var world: String = "mundo"

println(hello + world)
holamundo

world = "a todos"

println(hello + world)
holaa todos
println(s"$hello $world")
hola a todos
```

Al definir funciones en Scala es necesario indicar el tipo de los parámetros de entrada. El tipo de los parámetros de salida no es necesario indicarlo, a excepción de que se trate de una función recursiva:

```
def mult1(x: Int) = {
  x * x
}

def mult2(x: Int): Int = {
  x * x
}

def multDefaultValue(x: Int = 1) = {
  x * x
}

def sumRecursive(n: Int): Int = {
  if (n == 0) 0 else n + sumRecursive(n - 1)
}
```

```
def sumTailRecursive(n: Int): Int = {
  def sum(n: Int, acc: Int): Int = {
    if(n == 0) acc
    else sum(n - 1, acc + n)
  }
  sum(n, 0)
}
```

En Scala no disponemos de las funciones *for* o *while*; cuando queramos iterar sobre una lista, lo haremos usando la función *map()*. Además, Scala nos permite reducir la cantidad de código que escribimos con atajos conocidos como *syntactic sugar*:

```
val list: List[Int] = List(1, 2, 3)

list.map(elemento => mult2(elemento))

list.map(mult2(_))
```

En la siguiente sección trataremos varias veces con tuplas, así que veamos cómo estas se definen en Scala y distintas formas de manipularlas:

```
val tupla: List[(Int, String)] = List((1,"a"), (2,"b"), (3,"c"))

tupla.map(tupla =>
        (sumTailRecursive(tupla._1), tupla._2.toUpperCase))

tupla.map{case(firstElement, secondElement) =>
        (sumTailRecursive(firstElement),
        secondElement.toUpperCase())}

tupla.map{case(firstElement, _) =>
        sumTailRecursive(firstElement) }
```

Hasta aquí las nociones básicas que resultan necesario cubrir antes de continuar. Con lo que hemos visto podemos intuir que Scala es un lenguaje intuitivo y legible. Cuenta con la ventaja de heredar las librerías de Java, con el añadido de evitar efectos de lado si definimos las variables de manera inmutable.

Aunque no hemos hablado de ello en esta sección, por carecer de relevancia de cara a prepararnos para los próximos ejemplos, Scala ofrece funcionalidades muy interesantes como el control de casos usando *pattern matching* (similar a las guardas en Haskell) o el uso de implícitos, entre otros, haciendo de él un lenguaje muy potente.

8.2. Word Count

Como no podía ser de otra manera, empezaremos viendo cómo implementar un programa para contar palabras, el *"print ("hola mundo")* de la programación paralela:

```
@transient lazy val sparkSession: SparkSession =
  SparkSession
    .builder()
    .appName("WordCount")
    .config("spark.master", "local")
    .getOrCreate()

val sc = sparkSession.sparkContext
sc.setLogLevel("ERROR")

val path = "/home/evl/Escritorio/flights/data/Quijote.txt"
val quijote: RDD[String] = sc.textFile(path)

// amount word
val quijoteWordsCount = quijoterDD.flatMap(line
      => line.split(" ")).count
println(s"en el quijote hay: $quijoteWordsCount palabras")
```

El procedimiento es sencillo. Para empezar, cargamos el texto usando sparkContext. Al leer el texto, lo que obtenemos es un RDD compuesto por las líneas del documento, por lo que lo primero que debemos hacer es separar esas líneas en palabras. Esto lo conseguimos con la función split(" ") al recorrer el RDD usando la función flatMap(). A continuación, simplemente tendremos que contar el número de palabras con la función count().

Si lo que queremos es contar el número de apariciones de una palabra en concreto, utilizaremos la siguiente función:

```
    val word = "Dulcinea"
    val quijoteCount = wordCount(quijote, word)
    println(s"En el Quijote aparece $quijoteCount veces la
        palabra $word")

  def wordCount(rDD: RDD[String], key: String): Long = {
    val words = rDD.flatMap(_.split(" "))
    words.map(_.toLowerCase().replaceAll("\\P{L1}",""))
        .filter(_ == key.toLowerCase).count()
  }
```

Aquí, tras partir el RDD en palabras, no basta simplemente con encontrar coincidencias con la palabra buscada, puesto que en ese caso estaríamos perdiendo registros. Para evitar esto, haremos la comprobación pasando a minúsculas la palabra a buscar y las palabras del texto, además de descartar cualquier carácter especial que no sean letras, como comas, puntos o signos de exclamación, mediante el uso de una expresión regular.

No podemos repetir la misma palabra en posiciones tan cercanas. Este primer contacto con Spark nos muestra lo intuitivo y directo que resulta frente a competidores como Hadoop MapReduce.

8.3. RDD de pares clave valor

Una vez expuesto el problema más básico de la programación paralela, continuaremos familiarizándonos con la programación mediante RDD de pares clave valor. Para ello analizaremos un csv con registros de vuelos en EEUU.

En primer lugar, cargamos el csv en formato DataFrame usando la función read de sparkSession:

```
  @transient lazy val sparkSession: SparkSession =
    SparkSession.builder().appName("Sparkflights")
    .config("spark.master", "local").getOrCreate()
```

```
val path =  "~/2001.csv"

val flightsDF = sparkSession.read.format("csv")
      .option("path", path).option("header", "true")
      .option("inferSchema", "true").load.cache()
```

Empezamos calculando las n rutas más repetidas a lo largo de un año:

```
def getTopNflights(flightsDF: DataFrame, n: Int):
      Array[((String, String), Int)] = {

  val sqlContext = flightsDF.sqlContext
  import sqlContext.implicits._
  flightsDF.map(flightRow => {
    val source = flightRow.getAs[String](origin)
    val destination = flightRow.getAs[String](dest)
    ((source, destination), 1)
  }).rdd.reduceByKey(_ + _).sortBy(_._2,
  ascending = false).take(n)
}
```

Lo hacemos extrayendo los campos correspondientes al aeropuerto origen y al aeropuerto destino y usándolos como clave en una tupla cuyo valor asociado es 1. Al realizar esta operación lo que obtenemos es un Dafaset, por eso lo pasamos a RDD mediante la función rdd(). Este paso inicial lo repetiremos en las siguientes funciones, pues nos permite definir de manera rápida e intuitiva RDDs de pares clave valor a partir de un DataFrame. Llegados a este punto, simplemente tenemos que sumar todos los valores asociados a las claves, es decir, todas las veces que se ha producido un vuelo desde un aeropuerto origen A hasta un aeropuerto destino B; para ello usamos la función *reduceByKey()*. Como buscamos los n vuelos que más se han repetido, ordenamos el RDD en función de los valores asociados a las claves; gracias al azúcar sintáctico de Scala podemos hacerlo escribiendo muy poco. Ya solo resta quedarnos con los n primeros resultados usando la función *take()*.

Continuamos estudiando para cada mes el aeropuerto con mayor media de vuelos que han despegado desde él:

```
def getMaximunAvgByMonth(flightsDF: DataFrame):
        RDD[(Int,(String,Double))] = {

  val sqlC = flightsDF.sqlContext
  import sqlC.implicits._

  val fMonthKeySum = flightsDF.map(flightRow =>
    ((flightRow.getAs[Int]("Mont"),
    flightRow.getAs[String]("Origin")),1))
    .rdd.reduceByKey(_+_).cache()

  val fMonthSum = fMonthKeySum.map{case((month, _), count) =>
    (month,count)}.reduceByKey(_+_)

  val fMonthFligths = fMonthKeySum.map{case
        ((month, airport), count) =>
    (month, (airport, count))}

  fMonthKeySum.unpersist()
  val fmonthJoin = fMonthFligths.join(fMonthSum)

  val fmonthAvg = fmonthJoin.mapValues{case
        ((airport, airportCounter), monthCounter) =>
        (airport, (monthCounter/airportCounter).toDouble)}

  fmonthAvg.reduceByKey((current, next) =>
      if (current._2 < next._2) current
      else next)
}
```

En este caso, en primer lugar, agrupamos por mes y aeropuerto y contamos cuántos vuelos hay para cada clave. Como este RDD lo usaremos dos veces lo cacheamos. Para hacer la media necesitamos saber cuántos vuelos se han producido en cada mes, este resultado lo almacenamos en la variable *fMonthSuma*. La información que nos falta para poder calcular la media por mes es un RDD de clave mes y valor una tupla con cada aeropuerto y la cantidad de vuelos despegados desde él en cada mes; para obtener esta información simplemente tenemos que reordenar el rdd *fMonthKeySum*. Continuamos uniendo los dos RDDs mediante

la operación *join()*. LLegados a este punto, para calcular la media únicamente tenemos que recorrer los valores del RDD dividiendo los resultados obtenidos anteriormente, la cantidad de vuelos al mes entre la cantidad de vuelos con salida en cada aeropuerto. Por último, para quedarnos con el aeropuerto con mayor media por mes, recorremos todas las claves de dos en dos con la función *reduceByKey*, quedándonos con los registros con mayor media.

Con este ejemplo cerramos los ejercicios sobre el Core de Spark. Resultan ser muy representativos en la programación con RDDs. Esta manera de programar no es la más usual, pero sí sirve como una buena primera toma de contacto con Spark.

8.4. GraphFrames

En este apartado abordaremos el mismo dataset con el que hemos estado trabajando durante la sección anterior, pero, en esta ocasión, construiremos un grafo usando la librería GraphFrames y veremos alguno de los algoritmos y consultas que nos habilita.

Empezamos con la construcción del grafo. Para ello, la siguiente función recibirá como parámetro el DataFrame con los registros de vuelos creado en el apartado anterior:

```
def createGraph(flightsDF: DataFrame): GraphFrame = {

  val dayOfYear = (year:Int, month:Int, day:Int) =>
      new DateTime().year.setCopy(year).monthOfYear
      .setCopy(month).dayOfMonth.setCopy(day)
      .dayOfYear().get

  val flightsDS = flightsDF.map(flightRow => {
    val source = flightRow.getAs[String]("Origin")
    val destination = flightRow.getAs[String]("Dest")
    val distance = flightRow.getAs[Int]("Miles")
    val year = flightRow.getAs[Int]("Year")
    val month = flightRow.getAs[Int]("Month")
    val dayOfWeek = flightRow.getAs[Int]("DayOfWeek")
    val date = dayOfYear(year, month, dayOfWeek)
```

```
      (source, destination, date, distance)})

    val colNames = Seq(src, dst, date, miles)
    val edg = flightsDS.toDF(colNames: _*)

    val aiports = flightsDS.rdd.flatMap{case (source, dest, _,_)
        => Seq(source, dest)}.distinct()
    val vertexID = aiports.distinct().toDF("id")

    GraphFrame(vertexID, edg)
  }
```

Gracias a la librería *joda* podemos obtener el día del año de manera sencilla; esto nos será útil para filtrar por fechas más adelante. Para crear un grafo en GraphFrames necesitamos dos DataFrames, uno con la información de los vértices y otro con las aristas, siendo preciso que el campo con el que identificaremos los vértices sea denotado por **"id"**, mientras que en el caso de las aristas el nodo origen sea definido como **"src"** y el nodo destino como **"dst"**, no existiendo restricciones para el resto de los campos. En este caso nuestro DataFrame correspondiente a los vértices tendrá únicamente el campo identificador de los nodos, correspondiendo estos a los distintos aeropuertos del fichero, mientras que el DataFrame que define las aristas informará con el aeropuerto de partida, de llegada, el día del año en el que se produce el vuelo y la distancia recorrida.

Una vez que tenemos creado el GraphFrame, podemos utilizar las funciones que nos proporciona esta librería, que nos permite, por ejemplo, calcular de manera sencilla el nodo con mayor ratio entre el grado de entrada y el de salida:

GraphFrames nos habilita la posibilidad de hacer consultas mediante patrones [16]. Veámoslo con un ejemplo a través del cual buscaremos triángulos dirigidos en nuestro grafo:

```
def getMinDistTriangles(graph: GraphFrame): Dataset[Row] = {
  val triangles = graph.find("(a)-[ab]->(b);
      (b)-[bc]->(c); (c)-[ca]->(a)")
  triangles.where("ab.date < bc.date")
      .where("bc.date < ca.date")
}
```

Con la expresión (a)-[ab]->(b) en la función *find()* obtenemos todas las conexiones entre dos nodos genéricos a y b. Con el resto de la función *find()* estamos definiendo un triángulo de vértices a b c. Por último, lo interesante sería contemplar únicamente aquellos casos en los que el triángulo se haya recorrido en orden cronológico; lo hacemos empleando la función *where()*.

GraphFrames nos proporciona multitud de algoritmos sobre grafos [17]. Veamos, por ejemplo, cómo calcular el famoso algoritmo que usa Google (http://infolab.stanford.edu/ backrub/-google.html) para estimar la importancia de cada nodo del grafo, en función de los nodos que llegan a él y de la importancia de estos:

```
def getPageRankSorted(graph: GraphFrame): Dataset[Row]={

  val ranks = graph.pageRank.resetProbability(0.15).
      maxIter(10).run()

      ranks.vertices.orderBy($"pagerank".desc)
              .select("id", "pagerank")
}
```

Como podemos observar en este ejemplo, gracias a la librería GraphFrames tenemos acceso de una manera sencilla a potentes algoritmos sobre grafos.

Capítulo 9

Conclusiones

Durante el presente trabajo nos hemos ido adentrando en el framework Apache Spark, con la intención no solo de aprender a usarlo, sino también de conocer sus características internas, sus distintos elementos y cómo estos se relacionan entre sí, conocimientos necesarios para optimizar nuestras aplicaciones en Spark. Con lo abordado a lo largo de estas páginas, hemos aprendido que uno de los puntos fuertes de Spark es la canalización en las transformaciones narrow, obteniendo así una velocidad muy por encima con respecto a sus predecesores en aquellos casos en los cuales no necesitemos conocer la información ubicada en los distintos nodos del clúster, así como el rico abanico de posibilidades que nos ofrece Spark, yendo desde el machine learning hasta el procesamiento en streaming , pasando por el tratamiento de grafos.

Además, gracias a los conocimientos adquiridos ahora somos conscientes de los puntos delicados que tiene una aplicación Spark, destacando el cuidado que se debe tener al realizar una acción o cómo la operación shuffle influye en nuestros procesos.

Cabe destacar, que al mismo tiempo que hemos ido tomando contacto con Spark, nos hemos adentrado en el lenguaje Scala, que nos proporciona una acertada integración de los paradigmas de programación funcional y de la programación orientada a objetos, gracias a lo cual ha sido adoptado por muchas compañias como Twitter, LinkedIn o Sony, entre otras muchas.

Sin embargo, encontramos puntos en los que ahondar en esta memoria. Pese a que el objetivo principal se ha cumplido, entender los mecanismos del core de Spark, queda abierto el estudio en profundidad de las distintas librerías que componen esta herramienta, así como

un tratamiento más práctico de los distintos elementos estudiados.

Concluyo que este proyecto ha resultado una primera toma de contacto perfecta con una herramienta tan potente, compleja y viva como Spark; espero poner en práctica todo lo aprendido en mi vida laboral , así como continuar investigando y profundizando en esta materia.

9.1. Trabajo futuro

Partiendo de lo abordado a lo largo de esta memoria, se podría seguir estudiando las distintas librerías de Spark, así como las mejoras introducidas con las APIs estructuradas [18] [19]. Además, sería interesante darle un enfoque más práctico, trabajando con un clúster, y realizar flujos de trabajo integrando herramientas adicionales como pueden ser Kafka [20] o ElasticSearch [21].

Bibliografía

[1] WorldWideWebSize.com | The size of the World Wide Web (The Internet), Jun 2018. [Online; accessed 27. Jun. 2018].

[2] The Zettabyte Era: Trends and Analysis, Jun 2018. [Online; accessed 27. Jun. 2018].

[3] 10 Key Marketing Trends for 2017, Jul 2017. [Online; accessed 27. Jun. 2018].

[4] RGPD, May 2016. [Online; accessed 27. Jun. 2018].

[5] PYPL PopularitY of Programming Language index, Jun 2018. [Online; accessed 27. Jun. 2018].

[6] Bill Chambers and Matei Zaharu. *Spark: The Definitive Guide: Big data processing made simple*. O'Reilly UK Ltd., Apr 2018.

[7] Rachel Warren Holden Karau. *High Perfomance Spark*. O'Reilly, 2017.

[8] Apache Spark officially sets a new record in large-scale sorting, Nov 2014. [Online; accessed 26. Jun. 2018].

[9] M. Macías, M. Gómez, R. Tous, and J. Torres. *Introducción a Apache Spark*. Oberta UOC Publishing, 2015.

[10] Rishi Yadav. *Spark Cookbook*. Packt Publishing, Jul 2015.

[11] eBay/Spark, Jun 2018. [Online; accessed 26. Jun. 2018].

[12] RDD Programming Guide - Spark 2.3.1 Documentation, Jun 2018. [Online; accessed 26. Jun. 2018].

[13] Understand DAG and Physical Execution Plan in Apache Spark, Jun 2018. [Online; accessed 26. Jun. 2018].

[14] Directed Acyclic Graph DAG in Apache Spark - DataFlair, Apr 2017. [Online; accessed 26. Jun. 2018].

[15] Alvin Alexander. *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. O'Reilly Media, Aug 2013.

[16] Motif Analysis using Apache Spark GraphFrames, Jun 2018. [Online; accessed 26. Jun. 2018].

[17] User Guide - GraphFrames 0.5.0 Documentation, May 2017. [Online; accessed 26. Jun. 2018].

[18] Catalyst: A Query Optimization Framework for Spark and Shark - Databricks, Jun 2018. [Online; accessed 27. Jun. 2018].

[19] Deep Dive into Spark SQL's Catalyst Optimizer, Apr 2015. [Online; accessed 27. Jun. 2018].

[20] Real-Time End-to-End Integration with Apache Kafka in Apache Spark's Structured Streaming, Apr 2017. [Online; accessed 27. Jun. 2018].

[21] Apache Spark support | Elasticsearch for Apache Hadoop [6.3] | Elastic, Jun 2018. [Online; accessed 27. Jun. 2018].