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

Índice

Resumen

Aquí el resumen

Palabras clave

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

Abstract

resumen en ingles

Keywords

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

Capítulo 1

¿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 2

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 [1] 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 3

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.

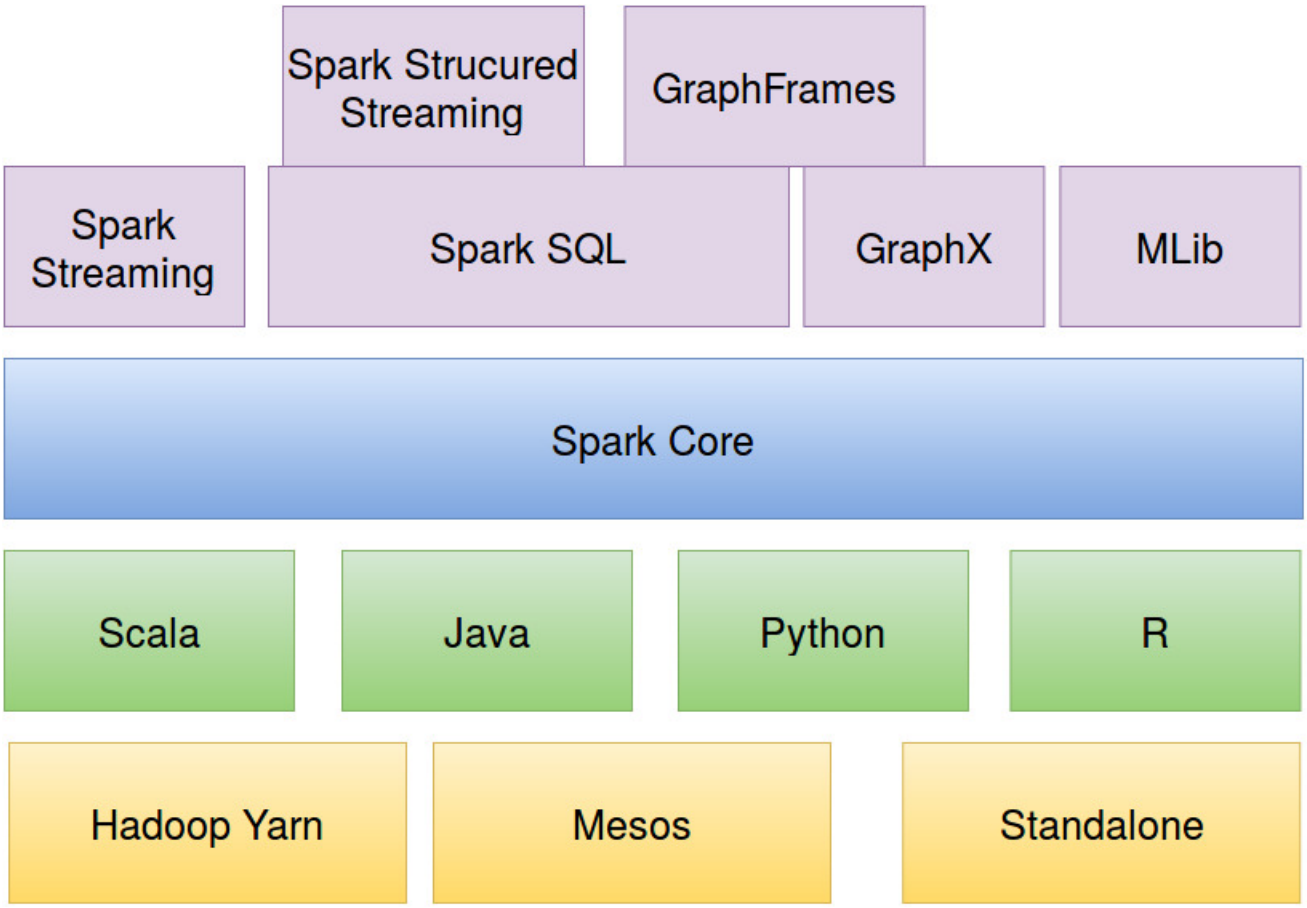

Figura 3.1: Ecosistema de Spark

Capítulo 4

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.

4.1. RDD de pares clave valor

Son un tipo especial de RDD [2], 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.

4.2. Propiedades de los RDD

Ahora que ya hemos estudiado la definición de los RDDs en general, veamos las cinco propiedades internas [3] 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.

4.3. 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*.

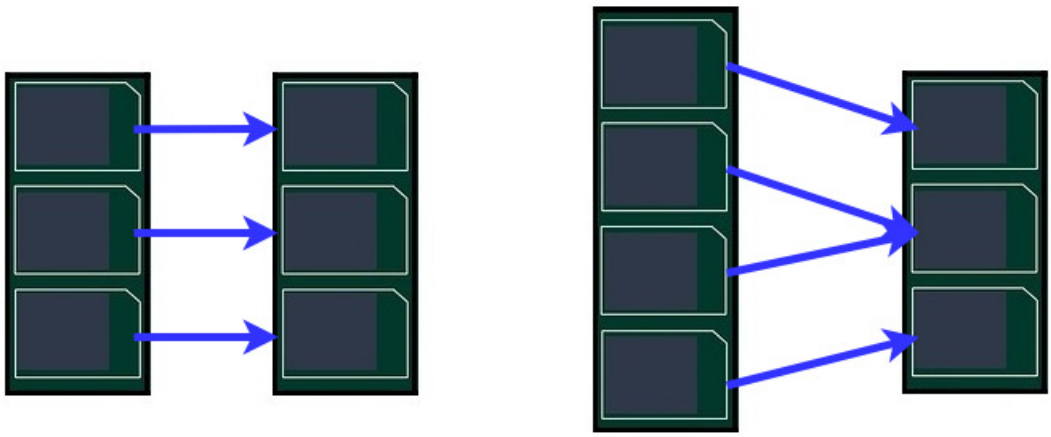

Figura 4.1: Transformaciones con dependencias narrow

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*.

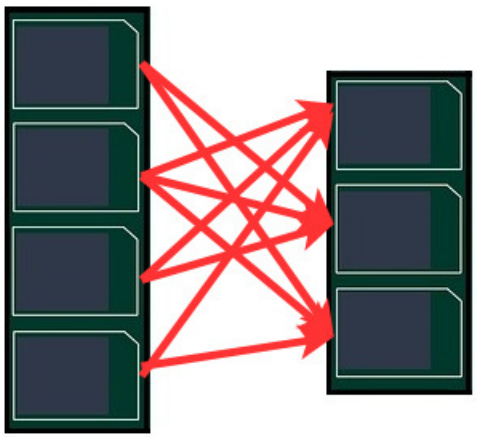

Figura 4.2: Transformaciones con dependencias wide

4.4. 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.

4.4.1. 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.

4.4.2. 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 5

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 acti-

vos 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.

5.1. 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.

5.2. 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.

5.3. 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 tareasnecesarias 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.

5.4. 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 [4] 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.

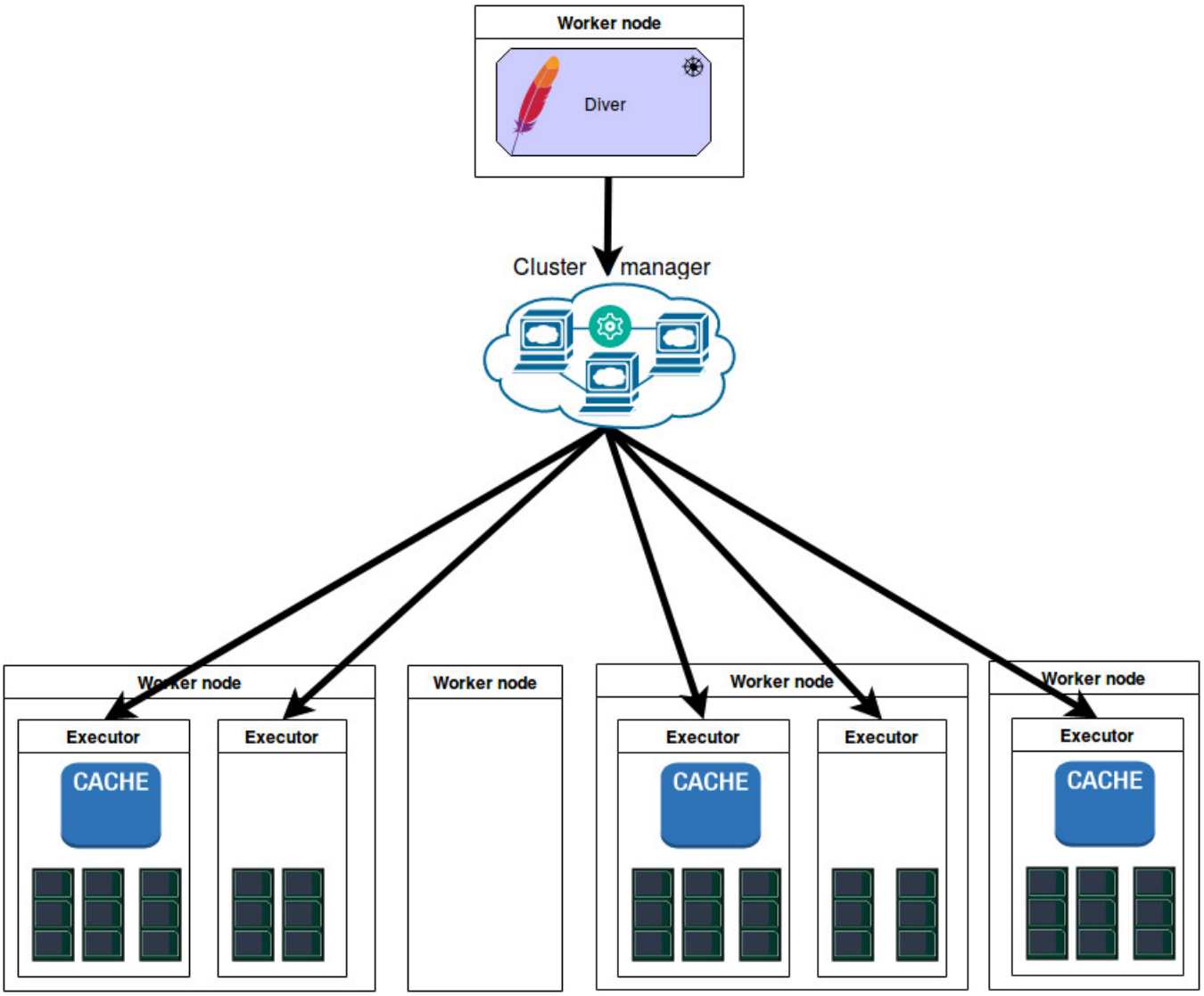

Figura 5.1: pie de foto

Capítulo 6

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.

6.1. 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.

6.2. 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.

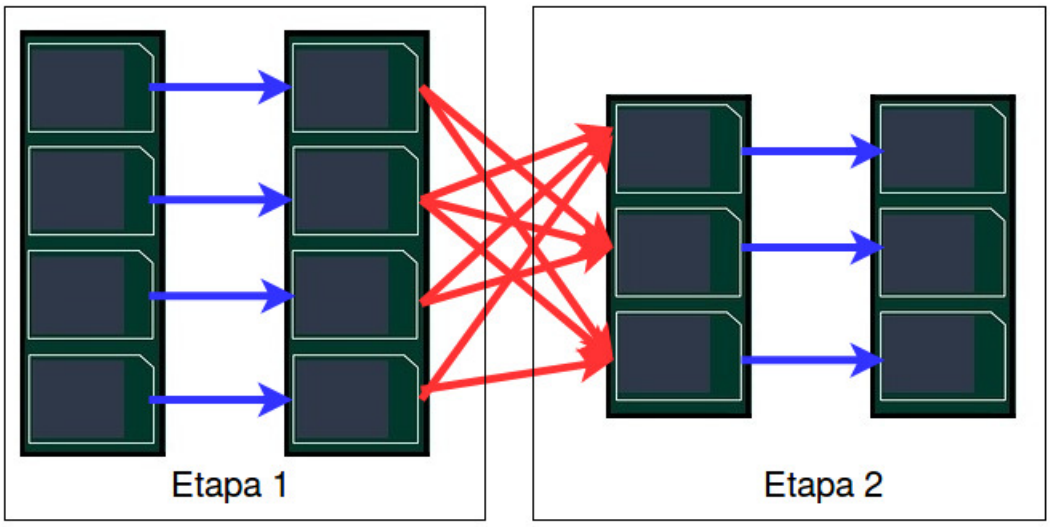

Figura 6.1: pie de foto

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.

6.3. 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.

6.4. 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.

6.5. 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.

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 [5] (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 [6]. 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 7

Codigo

Capítulo 8

Conclusiones

Bibliografía

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

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

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

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

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

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