Identificación, captura y almacenamiento de datos masivos

Introducción al cálculo distribuido

Jordi Nin

PID_00237566

Tiempo mínimo previsto de lectura y comprensión: **3 horas**

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Es un hecho natural que cada día generamos más y más datos, y que su captura, almacenamiento y proceso es básico en una gran variedad de situaciones, ya sean en el ámbito empresarial como en el ámbito de la investigación.

Para conseguir este objetivo, es necesario habilitar un conjunto de tecnologías que permitan llevar a cabo todas las tareas necesarias en el proceso de análisis de grandes volúmenes de información o *big data*. Estas tecnologías impactan en casi todas las áreas de las tecnologías de la información y comunicaciones, también conocidas como TIC. Desde el desarrollo de nuevos sistemas de almacenamiento de datos, como serían las memorias SSD (*Solid State Disk*) que permiten acceder de forma eficiente a grandes conjuntos de datos, o el desarrollo de nuevas redes de computadores, basadas en fibra óptica, que permiten compartir gran cantidad de datos entre múltiples servidores de forma eficiente, hasta nuevas metodologías de programación que permiten a los programadores usar estos nuevos componentes de hardware de forma sencilla.

En general, toda infraestructura para el procesamiento de *big data* requiere de una gran cantidad de servidores. Cada uno de estos servidores dispone de varias unidades centrales de proceso (CPU), una gran cantidad de memoria principal de acceso aleatoria (RAM) y un conjunto de discos duros para el almacenamiento estable de información. El principal objetivo de una infraestructura de *big data* es que todos estos elementos (CPUs, memoria y disco) sean accesibles de forma distribuida haciendo transparente para el usuario su uso, y proveyendo una falsa sensación de centralidad, es decir, para el programador solo hay una única CPU, memoria y disco.

Para permitir esta abstracción es necesario que los datos se encuentren redundados en diferentes servidores, y que el entorno de programación permita distribuir los cálculos que se han de realizar de forma sencilla y eficiente.

En primer lugar, debemos capturar y almacenar los datos disponibles. Actualmente, los sistemas de gestión de archivos distribuidos más habituales que encontramos en infraestructuras de *big data* son bases de datos relacionales, como Oracle, PostgreSQL o IBM BD2; bases de datos NoSQL como Cassandra, Redis o MongoDB; y sistemas de ficheros de texto como HDFS (Hadoop Distributed File System), que permite almacenar grandes volúmenes de datos. Estos sistemas son los encargados de almacenar y permitir el acceso a los datos de forma eficiente.

En segundo lugar, encontramos dos entornos de procesamiento de datos predominantes en el mercado actual:

- **Hadoop MapReduce**. Desarrollado por Google y que basa su forma de procesar datos en dos sencillas operaciones: la operación *map*, que distribuye el cómputo junto con sus correspondientes datos a los diferentes servidores, y la operación *reduce* que combina todos los resultados parciales obtenidos en las diferentes operaciones *map*. La principal limitación de este modelo de procesamiento es el uso intensivo que hace del disco duro. Esto hace que sea ineficiente en muchos casos, pero es extremadamente útil en otros.

- **Apache Spark**. Desarrollado por Matei Zaharia, que aunque se basa en la misma idea de "divide y vencerás" de Hadoop MapReduce, utiliza la memoria principal para distribuir y procesar los datos. Esto le permite ser mucho más eficiente cuando tiene que realizar cálculos iterativos (que se repiten continuamente). Esta capacidad le convierte en el sustituto perfecto de Hadoop MapReduce cuando este no es válido.

Para ambos entornos, los dos grandes retos a los que se enfrentan son:

1) Desarrollar algoritmos que sean capaces de trabajar únicamente con una parte de los datos y que sus resultados sean asociativos y fácilmente combinables.

2) Distribuir los datos de forma eficiente entre los servidores para no saturar la red durante el procesamiento.

En este material extenderemos todas estas ideas y veremos cómo es posible afrontar todos estos retos.

Objetivos

En los materiales didácticos de este módulo encontraremos las herramientas indispensables para asimilar los siguientes objetivos:

1. Comprender los diferentes componentes hardware de un sistema de procesamiento de datos masivos.

2. Conocer el *stack* de software típico de gestión de un sistema de procesamiento de datos masivos.

3. Entender cómo se almacenan los datos masivos en un sistema de archivos distribuido.

4. Ser capaz de diferenciar los diferentes tipos de procesamiento distribuido. Modelo *batch* (por lotes) frente modelo *streaming* (secuencial).

1. Qué entendemos por datos masivos o *big data*

Datos masivos o ***big data*** es el concepto con el cual hacemos referencia a la **identificación**, **captura**, **almacenamiento y procesamiento de grandes cantidades de datos** y a su vez a los procedimientos empleados para extraer conocimiento válido dentro de esos datos. Generalmente, en los documentos científico-técnicos en español se usa directamente el término en inglés *big data*, tal y como aparece en el articulo seminal de Viktor Schönberger "*Big data*: la revolución de los datos masivos".

Referencia bibliográfica

Viktor Mayer-Schönberger; Kenneth Cukier. "Big Data: A Revolution That Will Transform How We Live, Work and Think".

> Datos masivos es un término que hace referencia a un volumen de datos suficientemente grande que supera la capacidad del software habitual para ser capturados, administrados y procesados en un tiempo razonable. El volumen a partir del cual los datos empiezan a considerarse masivos crece constantemente. En 2012 se estimaba su tamaño de entre una docena de terabytes hasta varios petabytes de datos en un único conjunto de datos.

Uno de los principales problemas del *big data* es, que a diferencia de los sistemas gestores de bases de datos tradicionales, no se limita a fuentes de datos con una estructura determinada y sencilla de identificar, capturar, almacenar y procesar. Generalmente diferenciamos tres tipos de fuentes de datos masivos. Nótese que esta clasificación se aplica tanto a datos masivos como no masivos:

- **Datos estructurados (*Structured Data*).** Datos que tienen bien definidos su longitud y su formato, como las fechas, los números o las cadenas de caracteres. Se almacenan en formato tabular. Ejemplos de este tipo de datos son las bases de datos relacionales y las hojas de cálculo.

- **Datos no estructurados (*Unstructured Data*).** Datos que en su formato original, carecen de un formato específico. No se pueden almacenar en un formato tabular porque su información no se puede desgranar en un conjunto de tipos básicos de datos (números, fechas o cadenas de texto). Ejemplo de este tipo de datos son los documentos PDF o Word, documentos multimedia (imágenes, audio o video), correos electrónicos, etc.

- **Datos semiestructurados (*Semistructured Data*).** Datos que no se limitan a un conjunto de campos definidos como en el caso de los datos estructurados, pero a su vez contienen marcadores para separar sus diferentes elementos. Es una información poco regular como para ser gestionada de una forma estándar (tablas). Este tipo de datos poseen sus propios metadatos (datos que definen como son los datos) semiestructurados que describen

los objetos y sus relaciones, y que en algunos casos están aceptados por convención, como por ejemplo los formatos HTML, XML o JSON.

Muchas veces encontramos el concepto de *big data* relacionado con cuatro conceptos diferentes conocidos como "las 3 V del big data", más una cuarta V adicional.

- **V de Volumen**. Este es primer aspecto que se nos viene a la cabeza cuando pensamos en el *big data*, y nos dice que los datos tienen un volumen demasiado grande para que sean gestionados de la forma tradicional en un tiempo razonable.

- **V de Velocidad**. Aun y cuando los datos no sufren variaciones muy frecuentes, su análisis puede llevar horas e incluso días con técnicas tradicionales sin ser esto un gran problema. No obstante, en el ámbito del *big data* la cantidad de información crece tan de prisa, que el tiempo de procesamiento de la información se convierte en un factor fundamental para que dicho tratamiento aporte ventajas que marquen la diferencia.

- **V de Variedad**. Como hemos descrito anteriormente, el *big data* no procesa únicamente datos estructurados. Técnicamente, no es sencillo incorporar grandes volúmenes de información a un sistema de almacenamiento cuando su formato no está perfectamente definido. En este escenario nos encontramos con infinidad de tipos de datos que se aglutinan dispuestos a ser tratados y es por ello que frente a esa variedad aumenta el grado de complejidad tanto en el almacenamiento como en su procesamiento.

- **V de Veracidad**. Cuando disponemos de un alto volumen de información que crece a gran velocidad y que dispone de una gran variedad en su estructura, es inevitable dudar del grado de veracidad que estos datos poseen. Para ello, se requiere realizar limpieza y verificación en los datos para así asegurar que generamos conocimiento sobre datos veraces.

Fichero HTML

El *HyperText Markup Language* es un lenguaje de programación que se utiliza para el desarrollo de páginas de Internet.

Fichero XML

Un fichero XML (del inglés *eXtensible Markup Language*) es un tipo de documento semiestructurado, compuesto por datos elementales pero de definición no previamente conocida, que incluye etiquetas para describir su propia definición.

Fichero JSON

JSON (*JavaScript Object Notation*, en inglés) es un estándar abierto basado en texto diseñado para el intercambio de datos legible por humanos, que permite representar estructuras de datos simples y listas asociativas.

V de Veracidad

Las tres primeras V (Volumen, Velocidad y Variedad) aparecen en la definición original de *big data*, más adelante se incorporó la cuarta. Aun a día de hoy, esta cuarta V no está aceptada por todo el mundo.

1.1. Utilidad: ¿dónde encontramos *big data*?

La respuesta es sencilla: en una gran variedad de ámbitos. Algunos ejemplos son los siguientes:

1) Redes Sociales. Su uso que cada vez está más extendido, provoca la tendencia a que sus usuarios incorporen cada vez más una gran parte de su actividad y la de sus conocidos. Las empresas utilizan toda esta información con muchas finalidades. Por ejemplo para realizar estudios de marketing, evaluar su reputación o incluso cruzar los datos de los candidatos a un puesto de trabajo determinado.

2) Consumo. Amazon es líder en ventas cruzadas. Gran parte de su éxito se basa en el análisis masivo de datos basando en los patrones de compra de un usuario cruzados con los datos de compra de otro, creando así anuncios personalizados y boletines electrónicos que incluyen justo aquello que el usuario quiere en ese instante.

3) Salud y medicina. En 2009, el mundo experimentó una pandemia de gripe A, también conocida como gripe porcina o H1N1. El website Google Flu Trends fue capaz de predecirla gracias a los resultados de las búsquedas de palabras clave en su buscador. Flu Trends usó los datos de las búsquedas de los usuarios que contienen "Influenza-Like Illness Symptoms" (síntomas parecidos a la enfermedad de la gripe) y los agregó según ubicación y fecha, siendo capaz de predecir la actividad de la gripe hasta con dos semanas de antelación más que los sistemas tradicionales.

4) Política. Barak Obama fue el primer candidato a la presidencia de Estados Unidos en basar toda su campaña electoral en los análisis realizados por su equipo de *big data*. Este análisis le permitió vencer al candidato republicano Mitt Romney con el 51,06% de los votos, siendo esta una de las elecciones presidenciales más disputadas.

Lectura recomendada

Para saber más sobre la campaña electoral de Barak Obama podéis leer el siguiente artículo: Obama Used Big Data to Rally Voters

5) Telefonía. Las compañías de telefonía utilizan la información generada por los teléfonos móviles (posición GPS y los CDR –*Call Detail Record*–) para estudios demográficos, planificación urbana, etc.

Call Detail Record

Call Detail Record (CDR) es un registro de datos generado en la comunicación entre dos teléfonos fijos o móviles que documenta los detalles de la comunicación (ex. llamada telefónica, mensajes de texto, etc.). El registro contiene varios atributos como la hora, duración, estado de finalización, número de la fuente o número de destino.

1.2. La información no es conocimiento

La última de las cuatro V del *big data* nos advierte de que no todos los datos que capturamos tienen valor. Ya hace mucho tiempo Albert Einstein dijo que "la información no es conocimiento". ¡Cuánta razón tenía! Los datos necesitan ser procesados y analizados para que se les pueda extraer el valor que contienen.

Es fundamental conseguir la tecnología, tanto hardware como software, para transformar los datos en información, además de la habilidad analítica humana para transformar la información en conocimiento, de modo que con dicho conocimiento se puedan optimizar los procesos en los negocios.

1.3. ¿Cómo procesamos toda esta información?

Como hemos introducido anteriormente, una de las principales características del *big data* es la capacidad de procesar una gran cantidad de datos en un tiempo razonable. Esto es posible gracias a la computación distribuida.

La **computación distribuida** es un modelo para resolver problemas de computación masiva utilizando un gran número de ordenadores organizados en clústeres incrustados en una infraestructura de telecomunicaciones distribuida.

Las principales características de este modelo de cómputo son:

- La forma de trabajar de los usuarios de la infraestructura de cómputo debe ser similar a la que tendrían en un sistema centralizado.
- La seguridad interna en el sistema distribuido y la gestión de sus recursos es responsabilidad del sistema operativo y de sus sistemas de gestión y administración.
- Se ejecuta en múltiples servidores a la vez.
- Ha de proveer un entorno de trabajo cómodo para los programadores de aplicaciones.
- Dispone de un sistema de red que interconecta los diferentes servidores de forma transparente al usuario.
- Ha de proveer transparencia en el uso de múltiples procesadores y en el acceso remoto.
- Diseño de software compatible con varios usuarios y sistemas interactuando al mismo tiempo.

Múltiples servidores

En general cuando nos referimos a un conjunto de servidores hablaremos de clúster.

Aunque el uso de la computación distribuida facilita mucho el trabajo, los métodos tradicionales de procesado de datos no son válidos para estos sistemas de cálculo. Para conseguir el objetivo de procesar grandes conjuntos de datos, Google en 2004 creó la metodología de procesado de datos MapReduce, motor que está actualmente detrás de los procesamientos de datos de Google. Pero fue el desarrollo Hadoop MapReduce, por parte de Yahoo, lo que propicio un ecosistema de herramientas *open source* de procesamiento de grandes volúmenes de datos.

Open source

Open source (o código abierto) es el término con el que se conoce al software distribuido y desarrollado libremente. El código abierto tiene un punto de vista más orientado a los beneficios prácticos de compartir el código que a las cuestiones éticas y morales, las cuales destacan en el llamado software libre.

La innovación clave de MapReduce es la capacidad de ejecutar un programa, dividiéndolo y ejecutándolo en paralelo a la vez, a través de múltiples servidores sobre un conjunto de datos inmenso que también se encuentra distribuido.

Aunque la aparición de MapReduce cambió completamente la forma de trabajar y facilitó la aparición del *big data*, también poseía una gran limitación: únicamente es capaz de distribuir el procesamiento a los servidores copiando los datos a procesar a través de su disco duro. Esta limitación hace que este paradigma de procesamiento sea poco eficiente cuando es necesario realizar cálculos iterativos.

Ejemplo de cálculo iterativo

Cálculo de los pesos de una recta de regresión, o en general cualquier método de estimación de parámetros basado en el descenso del gradiente.

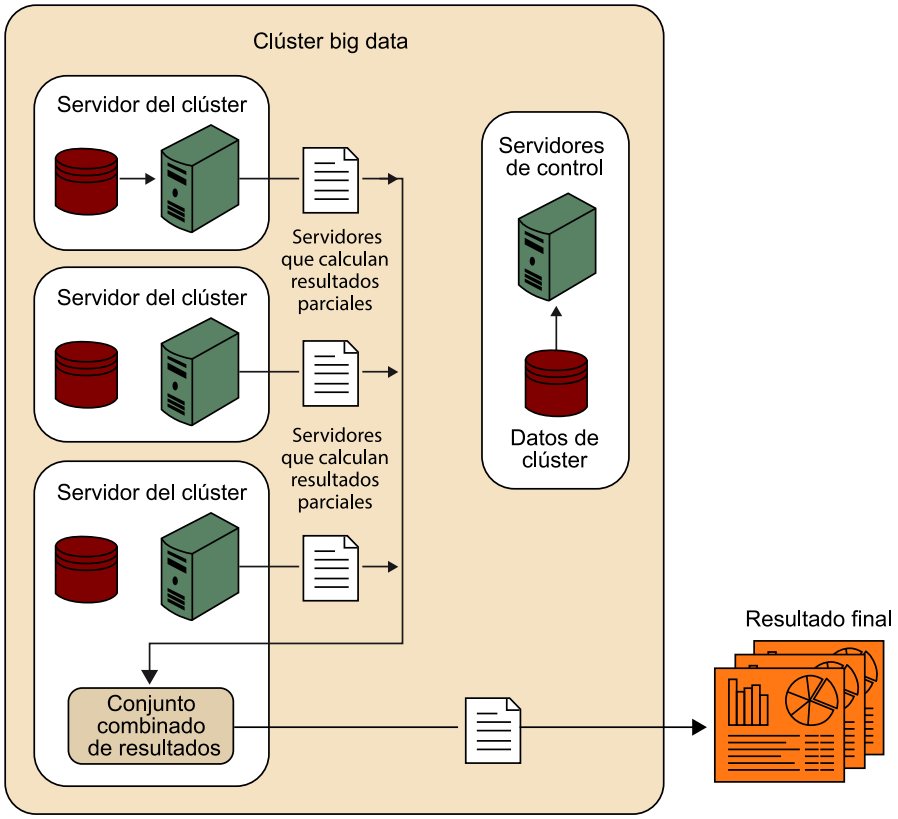
Posteriormente, en el año 2014, Matei Zaharia creó Apache Spark que solucionaba las limitaciones de MapReduce permitiendo distribuir los datos a los servidores usando su memoria principal o RAM. Esta innovación ha permitido que muchos algoritmos de procesamiento de datos puedan ser aplicados de forma eficiente a grandes volúmenes de datos de forma distribuida.

Hablaremos con mucho más detalle de estas dos tecnologías en los apartados posteriores.

2. Estructura general de un sistema de *big data*

Aunque todas las infraestructuras de *big data* tienen características específicas que adaptan el sistema a los problemas que tienen que resolver, todas comparten unos componentes comunes, como se describe en la figura 1.

Figura 1. Ejemplo de estructura general de un posible sistema de *big data*



La primera característica común que hay que destacar es que cada servidor del clúster posee su propio disco, memoria RAM y CPU. Esto permite crear un sistema de cómputo distribuido con ordenadores heterogéneos y de propósito general, no diseñados de forma específica para crear clústeres. Esto reduce muchos los costes de estos sistemas de computación, tanto de creación como de mantenimiento. Todos estos servidores se conectan a través de una red local. Esta red se utiliza para comunicar los resultados que cada servidor calcula con los datos que almacena localmente en su disco duro. La red de comunicaciones puede ser de diferentes tipos, desde Ethernet a fibra óptica, dependiendo de cómo de intensivo sea el intercambio de datos entre los servidores.

Red Ethernet

Ethernet es un estándar de redes de área local para computadores con acceso al medio por detección de la onda portadora y con detección de colisiones (CSMA/CD). Su nombre viene del concepto físico de *ether*.

En la figura 1 observamos tres tipos de servidores:

- Los servidores que calculan resultados parciales. Estos servidores se ocupan de hacer los cálculos necesarios para obtener el resultado deseado en los datos que almacenan en su disco duro.

- Los servidores que combinan los diferentes resultados parciales para obtener el resultado final deseado. Estos servidores son los encargados de almacenar durante un tiempo los resultados finales.

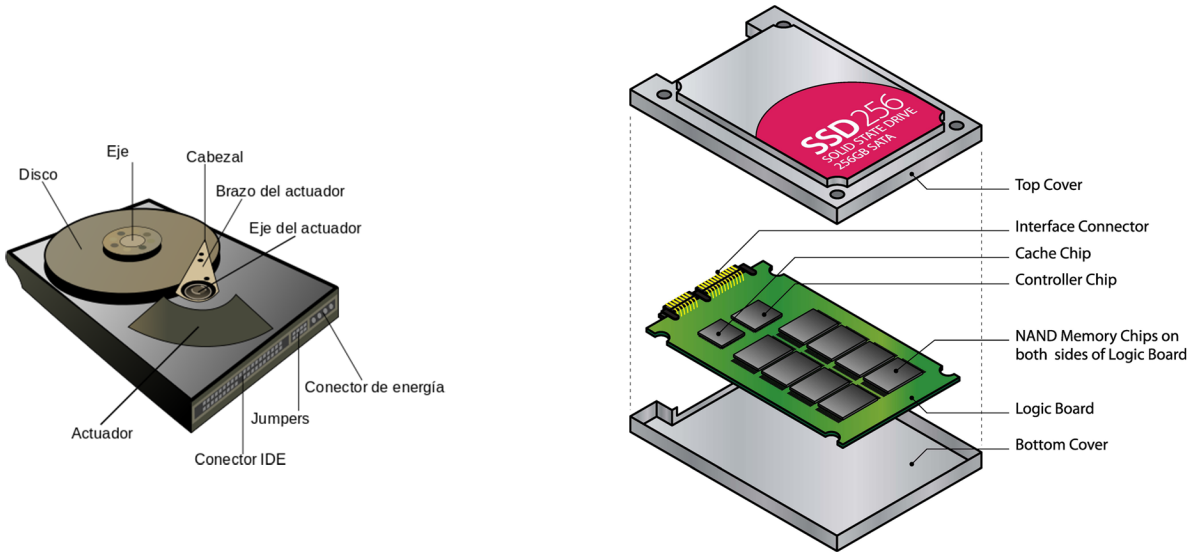
- Los servidores de control o gestores de recursos, que aseguran que el uso del clúster sea correcto y que ninguna tarea sature los servidores. También se encargan de inspeccionar que los servidores funcionen sin errores. En el caso de que detecten un mal funcionamiento, fuerzan el reinicio del servidor y avisan al administrador que hay problemas en ciertos nodos.

El uso combinado de este conjunto de servidores es posible, como veremos durante en este módulo didáctico, gracias al uso de un sistema de ficheros distribuido y la implementación de los cálculos en un entorno de programación distribuida, como por ejemplo Hadoop o Spark.

3. Sistema de archivos

El **sistema de archivos** o **sistema de ficheros** es el componente encargado de administrar y facilitar el uso del sistema de almacenamiento, ya sea basado en discos duros magnéticos (HDD, *Hard Disk Drive*) o en memorias de estado sólido (SDD, *Solid State Disk*). En general, es extraño en sistemas de *big data* usar almacenamiento terciario como DVD o CD-ROM. En la figura 2 podemos observar una comparación entre ambas tecnologías.

Figura 2. Comparación entre un disco duro magnético y una memoria de estado sólido.



De forma breve podemos decir que un disco duro tradicional (figura 2*a*) se compone de uno o más platos o discos rígidos, unidos por un mismo eje que gira a gran velocidad dentro de una caja metálica sellada. Sobre cada plato, y en cada una de sus caras, se sitúa un cabezal de lectura/escritura que flota sobre una delgada lámina de aire generada por la rotación de los discos. Para leer o escribir información primero se ha de buscar la ubicación donde realizar la operación de lectura o escritura en la tabla de particiones ubicada al principio del disco. Luego se ha de posicionar el cabezal en el lugar adecuado, y finalmente, leer o escribir la información de forma secuencial. Como estos dispositivos de almacenamiento no disponen de acceso aleatorio, se suelen considerar sistema de almacenamiento lento. En cambio las memorias de estado sólido (figura 2*b*), sí que disponen de acceso aleatorio a la información almacenada y en general son bastante más rápidas, pero tienen una vida útil relativamente corta y que se reduce drásticamente si realizamos muchas operaciones de escritura.

Volviendo a los sistemas de archivos, decimos que sus principales funciones es la asignación de espacio a los archivos, la administración del espacio libre y del acceso a los datos almacenados. Los sistemas de archivos estructuran la información almacenada en un dispositivo de almacenamiento de datos, que luego será representada ya sea textual o gráficamente utilizando un gestor de archivos.

La estructura lógica de los archivos suele representarse de forma jerárquica o en "árbol" usando una metáfora basada en la idea de carpetas y subcarpetas para organizar los archivos con algún tipo de orden. Para acceder a un archivo se debe proporcionar su **ruta** (orden jerárquico de carpetas y subcarpetas) y su **nombre** de archivo seguido de una **extensión** (por ejemplo, .txt) que indica el contenido del archivo.

Ruta completa de un fichero de datos

home/user/mydata/data.csv es la ruta completa con su nombre de archivo y extensión de un fichero de datos.

Cuando trabajamos con grandes volúmenes de información, un único dispositivo de almacenamiento no es suficiente y debemos utilizar sistemas de archivos que permitan gestionar múltiples dispositivos. Esta característica descrita de esta forma no es exactamente lo que necesitamos, de hecho cualquier sistema de archivos moderno permite almacenar información en diversos dispositivos de una forma más o menos transparente al usuario. Realmente lo que necesitamos es un sistema de archivos que nos permita gestionar múltiples dispositivos distribuidos en diferentes nodos (ordenadores) conectados entre ellos utilizando un sistema de red.

Cuando requerimos de este nivel de distribución ya no todos los sistemas de archivos nos sirven. Un **sistema de archivos distribuido** o **sistema de archivos de red** es un sistema de archivos de ordenadores que sirve para compartir archivos, impresoras y otros recursos como un almacenamiento persistente en una red de ordenadores. El sistema NFS (de sus siglas en inglés, *Network File System*) fue desarrollado por Sun Microsystems en el año 1985 y es un sistema estándar y multiplataforma que permite acceder y compartir archivos en una red heterogénea como si estuvieran en un solo disco. Pero, ¿es esto realmente lo que necesitamos? La respuesta es no. Este sistema nos permite acceder a una gran cantidad de datos de forma distribuida, pero sufre un gran problema: los datos no están almacenados en el mismo sitio donde se han de realizar los cálculos, lo que provoca que cada vez que hemos de ejecutar un cálculo, los datos tienen que ser copiados por la red de un nodo a otro. Este proceso es lento y costoso.

Para solucionar este problema se creó el Hadoop Distributed File System (HDFS), que es un sistema de archivos distribuido, escalable y portátil para el *framework* de cálculo distribuido Hadoop. En el siguiente subapartado detallaremos su funcionamiento y sus principales componentes.

Logo del Hadoop Distributed File System (HDFS)

3.1. Hadoop Distributed File System (HDFS)

HDFS es un sistema de ficheros **distribuido**, **escalable y portátil** escrito en Java y creado especialmente para trabajar con ficheros de gran tamaño. Una de sus principales características es un **tamaño de bloque muy superior al habitual** para no perder tiempo en los accesos de lectura. Los ficheros que normalmente van a ser almacenados o ubicados en este tipo de sistema de ficheros siguen el patrón "Write once read many" (escribe una vez y lee muchas). Por lo tanto, está especialmente indicado para procesos *batch* de grandes ficheros, los cuales solo serán escritos una vez y, por el contrario, serán leídos gran cantidad de veces para poder analizar su contenido profundamente.

Por tanto, en HDFS tenemos un sistema de archivos distribuido y especialmente optimizado para almacenar grandes cantidades de datos. De este modo, los ficheros serán divididos en bloques de un mismo tamaño y distribuidos entre los nodos que forman el clúster de datos (los bloques de un mismo fichero se ubicarán en nodos distintos). Esto nos facilitará el cómputo en paralelo y nos evitará desplazar grandes volúmenes de datos entre diferentes nodos de una misma infraestructura de *big data*.

Tamaño mínimo de información

El tamaño mínimo de información a leer en HDFS es de 64 MB, mientras que en los sistemas de archivo no distribuido este tamaño no suele superar los centenares de KBytes.

Las arquitecturas HDFS tiene dos tipos de nodos, diferenciados completamente según el rol o la función que vayan a desempeñar a la hora de ser usados. Los dos tipos de nodos HDFS son los siguientes:

Proceso batch

Se conoce como sistema por lotes (*batch*) la ejecución de un programa sin el control o supervisión directa del usuario. Este tipo de programas se caracterizan porque su ejecución no precisa ningún tipo de interacción con el usuario.

- **Namenode (JobTracker)**. Este tipo de nodo, del que solo hay uno por clúster, es el más importante ya que es responsable de la topología de todos los demás nodos y, por consiguiente, de gestionar el espacio de nombres. El espacio de nombres (*namespace*, en inglés) indica la ubicación (ruta) donde se encuentran los datos. Concretamente indica el nombre del *rack* (y más precisamente, del *switch*) donde está el nodo con los datos.

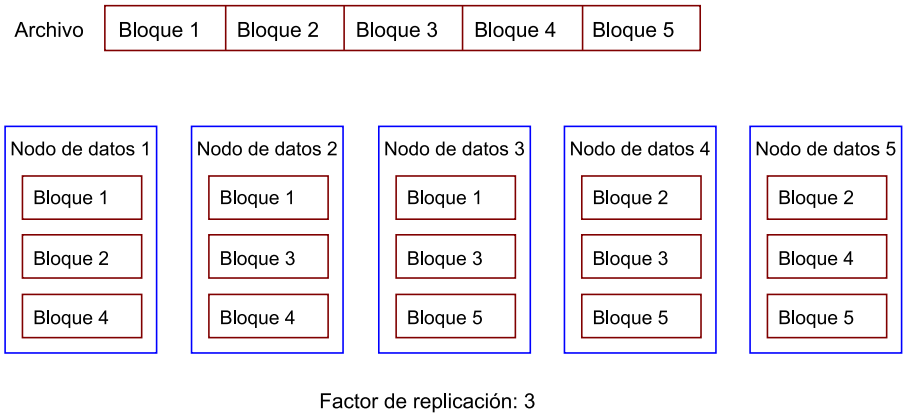
- **Datanodes (TaskTracker)**. Este tipo de nodos, de los que normalmente van a existir varios, son los que realizan el acceso a los datos propiamente dicho. En este caso, almacenan los bloques de información y los recuperan bajo demanda.

Simplificando, se puede considerar el JobTracker como el nodo principal o director de orquesta, mediante el cual se va a distribuir el tratamiento y procesado de los ficheros en los TaskTracker, o nodos worker, que realizarán el trabajo. Una tarea muy importante del sistema de ficheros HDFS es definir correctamente el número de réplicas de cada uno de los archivos de datos. Este valor indica cuantas copias hay en el clúster de cada fichero: a más copias menos necesidad de desplazar datos entre los *datanodes* pero menos espacio para almacenar datos. Es un valor que tiene que definirse de forma correcta.

Ejemplo de almacenamiento en HDFS

En la figura 3, se ha definido el valor del número de replicas a 3. Este número permite que cada *datanode* posea más del 50% de la información, por lo tanto no tiene que solicitar una gran cantidad de información a los otros *datanodes*. Por otro lado, permite que podamos terminar los procesos en curso, incluso si fallan dos nodos de los cinco que dispone el clúster.

Figura 3. Ejemplo de almacenamiento en HDFS



3.2. Bases de datos NoSQL

Hasta hace unos años, las bases de datos relacionales ha sido la única alternativa a los sistemas de ficheros para almacenar grandes volúmenes de información. Este tipo de bases de datos utilizan SQL (lenguaje de consulta estructurado) como lenguaje de referencia. Este tipo de bases de datos siguen las reglas ACID. Estas propiedades ACID permiten garantizar que los datos son almacenados de forma fiable y cumpliendo con un conjunto de reglas de integridad definidas sobre una estructura basada en tablas que contienen filas y columnas.

Sin embargo con los requerimientos del mundo del *big data* nos encontramos que las bases de datos relacionales no pueden manejar el tamaño, la complejidad de los formatos o la velocidad de entrega de los datos que requieren muchas aplicaciones. Un ejemplo de aplicación sería Twitter, donde millones de usuarios acceden al servicio de forma concurrentes tanto para consultar como para generar nuevos datos.

Lenguaje SQL

SQL es el acrónimo en inglés de *structured query language*. El SQL es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en ellas.

Estas nuevas aplicaciones han propiciado la aparición de nuevos sistemas de bases de datos, llamados NoSQL, que permiten dar una solución a los retos de escalabilidad y rendimiento que representa el *big data*.

Reglas ACID

En el contexto de bases de datos, ACID (acrónimo inglés de atomicity, consistency, isolation, durability) son una serie de propiedades que tiene que cumplir todo sistema de gestión de bases de datos para garantizar que las transacciones sean fiables. Para una explicación más detallada consultar siguiente entrada de la Wikipedia: http://es.wikipedia.org/wiki/ACID.

El concepto NoSQL agrupa diferentes soluciones para almacenar diferentes tipos de datos, desde tablas a grafos, pasando por documentos, imágenes o cualquier otro formato. Cualquier base de datos NoSQL es distribuida y escalable por definición. Hay numerosos productos disponibles, muchos de ellos *open source*, como Cassandra, que basa su sistema de almacenamiento en guardar la información en forma de columna y generando un conjunto de índices asociativos que le permiten recuperar grandes bloques de información en un tiempo muy bajo.

Las bases de datos NoSQL no suponen que nunca más se vuelva a usar el lenguaje SQL, sino que simplemente aportan soluciones alternativas que mejoran el rendimiento de los sistemas gestores de bases de datos para determinados problemas y aplicaciones. Por este motivo, NoSQL también se asocia al concepto "**not only SQL**". NoSQL no prohíbe el lenguaje estructurado de consultas. Si bien es cierto que algunos sistemas NoSQL son totalmente no-relacionales, otros simplemente evitan funcionalidades relacionales concretas como esquemas de tablas fijas o ciertas operaciones del álgebra relacional. Por ejemplo, en lugar de utilizar tablas, una base de datos NoSQL podría organizar los datos en objetos, pares clave-valor o incluso tuplas secuenciales.

El principio que siguen este tipo de bases de datos es el siguiente: como en determinados escenarios no es posible utilizar bases de datos relacionales, no queda otro remedio que relajar alguna de las limitaciones inherentes de este tipo de sistemas de almacenamiento. Por ejemplo, podemos pensar en colecciones de documentos con campos definidos de forma no estricta, que incluso pueden ir cambiando en el tiempo, en lugar de tablas con filas y columnas con un formato prefijado. En cierto modo, incluso podríamos llegar a pensar que un sistema de este tipo no es ni siquiera una base de datos entendida como tal, sino un sistema de almacenamiento distribuido para gestionar datos dotados de una cierta estructura que puede ser extremadamente flexible.

En general hay cuatro tipos de bases de datos NoSQL, dependiendo de cómo almacenan la información:

- **Clave-valor**. Este formato es el más típico. Podemos entenderlo como un HashMap donde cada elemento está identificado por una llave única, lo que permite la recuperación de la información de manera muy rápida. Normalmente el valor se almacena como un objeto binario y su contenido no es importante para el clúster.

- **Basada en documentos**. Este tipo de base de datos almacena la información como un documento (generalmente con una estructura simple como JSON o XML) y con una clave única. Es similar a las bases de datos clave-valor, pero con la diferencia que el valor es un fichero que puede ser entendido por el clúster y puede realizar operaciones sobre los documentos.

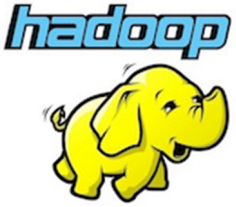
- **Orientadas a grafos**. Hay otras bases de datos que almacenan la información como grafos donde las relaciones entre los nodos es lo más importante. Son muy útiles para representar información de redes sociales.

- **Orientadas a columnas**. Guardan los valores en columnas en lugar de filas. Con este cambio ganamos mucha velocidad en lecturas, ya que si se requiere consultar un número reducido de columnas, es muy rápido hacerlo. La principal contrapartida es que no eficiente para realizar escrituras.

HashMap

Un HashMap es una colección de objetos, como un vector o *arrays*, pero sin orden. Cada objeto se identifica mediante algún identificador apropiado. El nombre *hash*, hace referencia a una técnica de organización de archivos llamada *hashing* o dispersión en el cual se almacenan los registros en una dirección que es generada por una función que se aplica sobre la clave del registro.

4. Sistema de cálculo distribuido



Logo de Apache Hadoop

Los sistemas de cálculo distribuido permiten la integración de los recursos de diferentes máquinas en red, convirtiendo la ubicación de un recurso en algo transparente al usuario. El usuario accede a los recursos del sistema distribuido a través de un gestor de recursos, despreocupándose de dónde se encuentra ese recurso y de cuándo lo podrá usar. En este módulo nos centraremos en describir dos sistemas de cálculo distribuido diferentes y muy extendidos en el ecosistema del *big data*: MapReduce y Spark.

4.1. MapReduce: divide y vencerás. Usos y limitaciones

MapReduce es un modelo de programación introducido por Google en 1995 para dar soporte a la computación paralela sobre grandes volúmenes de datos, con dos características principales:

1) usa clústeres de ordenadores y

2) usa *hardware* no especializado.

El nombre de este sistema está inspirado en los nombres de sus dos métodos o funciones de programación principales: *Map* y *Reduce*, que veremos a continuación. MapReduce ha sido adoptado mundialmente, gracias a que existe una implementación *open source* denominada Hadoop desarrollada por Yahoo que permite usar este paradigma utilizando el lenguaje de programación Java.

Lo primero que hemos de tener en cuenta cuando hablamos de este modelo de cálculo es que no todos los análisis pueden ser calculados con este paradigma. Concretamente, solo son aptos aquellos que pueden calcularse como combinaciones de las operaciones de `map()` y de `reduce()`. Las funciones Map y Reduce están definidas ambas con respecto a datos estructurados en tuplas del tipo (clave, valor). En general este sistema funciona muy bien para calcular agregaciones, filtros, procesos de manipulación de datos, estadísticas, etc., operaciones todas ellas fácil de paralelizar y que no requieren de un procesamiento iterativo y en los que no es necesario compartir los datos entre todos los nodos del clúster.

En la arquitectura MapReduce todos los nodos se consideran *workers* (trabajadores), excepto uno que toma el rol de *master* (maestro). El maestro se encarga de recopilar trabajadores en reposo (es decir sin tarea asignada) y le asignará una tarea específica de `map()` o de `reduce()`. Un *worker* solo puede tener tres estados: reposo, trabajando, completo. El rol de maestro se asigna de manera aleatoria en cada ejecución.

Ahora veamos con un poco más de detalle cómo funcionan las dos operaciones básicas de MapReduce:

- La función `map()` es una función asociativa y se aplicada en paralelo a todos los elementos del conjunto de datos de entrada. En este punto es muy importante el concepto de asociatividad, ya que no podemos establecer un orden concreto en la finalización de las diferentes funciones `map()`. Como resultado devuelve una lista de pares (clave, valor) por cada llamada. Una vez que se ha calculado esta asociación clave-valor, el clúster agrupa los pares con la misma clave de todas las listas, creando un grupo por cada una de las diferentes claves generadas. Desde el punto de vista de la arquitectura, el nodo máster toma los datos de entrada, los divide en pequeñas piezas o problemas de menor complejidad, y los distribuye a los nodos *worker*. A su vez un nodo *worker* puede volver a subdividir los datos que le han llegado, dando lugar a una estructura en forma de árbol. Una vez se ha terminado la división de los datos, los nodos *worker* procesan los subproblemas y pasan las respuestas al nodo maestro.

- La función `reduce()` produce una llamada vacía, un valor o incluso una lista de valores en cada llamada. El retorno de todas esas llamadas, independientemente de si han producido o no un resultado, se recoge como la lista de resultado final deseado.

En consecuencia, una ejecución MapReduce transforma una lista de pares (clave, valor) en una lista de valores. La función `map()` se ejecuta en paralelo de forma distribuida en cada uno de los nodos *worker* del clúster. Como hemos visto en el apartado 3, los datos de entrada que se encuentran almacenados en HDFS, se dividen en un conjunto de *M* particiones de entrada. Estas particiones son procesadas en diversos nodos. Como se describe en la figura 4, en una llamada de MapReduce suelen ocurrir las siguientes operaciones:

- Se dividen los datos de entrada en relación al número de *workers* disponibles en ese momento.

- Se decide cuál de los nodos va a ser el nodo máster, el resto de nodos serán considerados *workers*. El nodo máster se encarga de buscar los nodos *worker* en reposo (sin tarea asignada) y le asignará una tarea específica de `map()` o de `reduce()`. Un *worker* solo puede tener tres estados: *reposo, trabajando o completo*.

- Un *worker* que reciba una tarea de `map()` usará como entrada la partición que le corresponda y parseará los pares (clave, valor) para crear una nueva pareja de salida, tal y como este definido dentro de la función `map`. Los pares clave y valor producidos se almacenan como *buffer* en la memoria.

- Cada cierto tiempo, los pares clave-valor almacenados en los *buffers* de los *workers* se escriben en el disco local, distribuidos en *R* regiones. Dichas re-

giones son enviadas al nodo máster, que se encarga de reenviar los nuevos datos a los nodos *worker* que tengas asignadas tareas de `reduce()`.

- Cuando el nodo maestro notifica a un nodo *worker* de tipo reduce la localización de una partición, éste emplea una serie de llamadas remotas para hacer lecturas de la información almacenada en los discos duros de los workers de tipo `map()`. Cuando el worker de tipo `reduce()` lee todos los datos, agrupa los datos usando la información contenida en las claves de modo que se agrupen los datos que poseen la misma clave. El paso es necesario debido a que muchas claves de funciones `map()` diversas, pueden ir a una misma función `reduce()`.

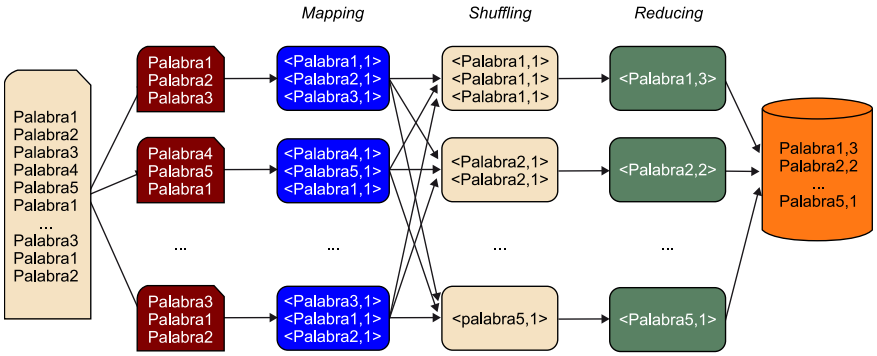
- Los nodos *worker* de tipo `reduce()` iteran sobre el conjunto de valores ordenados intermedios para cada una de las claves únicas encontradas. Para poder realizar este paso, es necesario que la función `reduce()` conozca el conjunto de valores asociados a cada clave. La salida de esta función se añade al fichero de salida de la ejecución.

- Una vez todas las tareas `map()` y `reduce()` se han completado, el nodo maestro finaliza la ejecución y retorna el control al usuario.

Ejemplo de conteo de palabras

Como se ve en la figura 4, la función `map()` se ocupa de convertir cada palabra en una estructura clave-valor, donde la clave será la propia palabra y el valor será el valor 1. Posteriormente, se agrupan estas estructuras clave-valor que comparten la misma clave en un mismo nodo. Finalmente la función `reduce()` se ocupa de sumar todos los valores, en este caso todos serán igual a 1, y a devolver una nueva estructura clave-valor donde se almacena la palabra y su número de apariciones. Esto puede sirve para ejecutar un nuevo cálculo más complejo sobre las palabras o como en el ejemplo para devolver un listado con el resultado al usuario.

Figura 4. Ejemplo de proceso en MapReduce



Uno de los aspectos más importante de esta forma de realizar cálculos es que es **tolerante a fallos**. Cuando en uno de los nodos *workers* produce un fallo, el nodo máster se da cuenta de este fallo ya que periódicamente hace una solicitud de estatus a cada uno de los nodos *worker*. Si la comprobación del *estatus* no es correcta, se cancela el trabaja asignado a ese nodo *worker* y se reasigna a otro nodo para que lo realice.

4.2. Spark: procesamiento distribuido en memoria principal

Como hemos descrito en el subapartado 4.1., MapReduce únicamente es capaz de distribuir el procesamiento a los servidores copiando los datos que hay que procesar a través de su disco duro. Esta limitación reduce el rendimiento de los cálculos iterativos, donde los mismos datos tienen que usarse una y otra vez. Este tipo de procesamiento es básico en la mayoría de algoritmos de aprendizaje automático.

Logo de Apache Spark

El proyecto de Spark se centró desde el comienzo en aportar una solución factible a estos defectos de Hadoop, mejorando el comportamiento de las aplicaciones que hacen uso de MapReduce y aumentando su rendimiento considerablemente.

Spark es un sistema de cálculo distribuido para el procesamiento de grandes volúmenes de datos y que gracias a su llamada "interactividad" hace que el paradigma MapReduce ya no se limite a las fases `map()` y `reduce()` y podamos realizar más operaciones como mappers, reducers, joins, groups by, filtros, etc. Spark proporciona API para Java, Scala y Python, aunque es preferible que se programe en Scala, ya que es su lenguaje nativo.

API

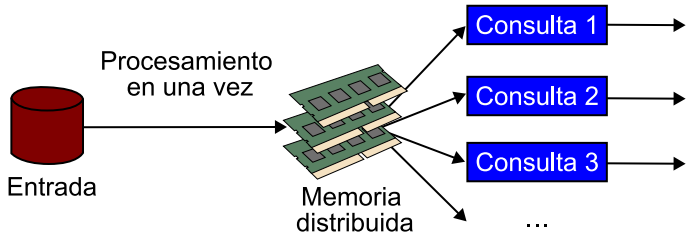
Una Interfaz de Programación de Aplicaciones (API) es el conjunto de rutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece una biblioteca de programación para que pueda ser utilizada por otro software como una capa de abstracción.

La principal ventaja de Spark respecto a Hadoop es que guarda todas las operaciones sobre los datos en memoria. Esta es la clave de su buen rendimiento. La figura 5 muestra algunas de sus principales características:

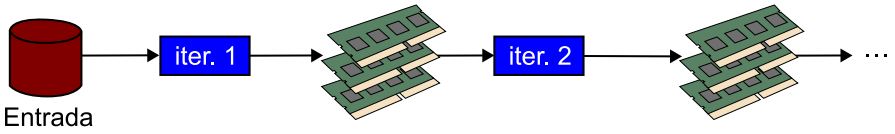
- Baja latencia computacional mediante el cacheo de los datos en memoria (figura 5*a*).

- Los algoritmos iterativos se ejecutan de forma eficiente gracias a que las sucesivas operaciones comparten los datos en memoria (figura 5*b*).

Figura 5. Comparación de una ejecución de Spark y Hadoop



a. Baja latencia computacional mediante el cacheo de los datos en memoria



b. Algoritmos iterativos

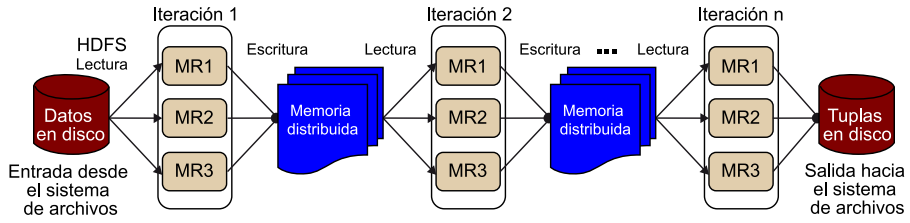
En la figura 6 se ilustra cómo se realiza la ejecución de un programa en Spark. Una ejecución típica de Spark se organiza de la siguiente manera:

1) A partir de una variable de entorno llamada *spark context* que define cómo está organizado el clúster, se crea un objeto RDD leyendo datos del sistema de ficheros (que podría ser perfectamente HDFS), una base de datos o cualquier otra fuente de información como una lista.

2) Una vez creado el RDD inicial se realizan transformaciones para crear más objetos RDD a partir del primero. Dichas transformaciones se expresan en términos de programación funcional y no eliminan el RDD original, sino que crean uno nuevo.

3) Tras realizar las acciones y transformaciones necesarias sobre los datos, los objetos RDD deben converger para crear el RDD final. Este RDD se acaba almacenado en el sistema de archivos del clúster o en cualquier otro lugar que ofrezca persistencia.

Figura 6. Flujo de ejecución de Spark



4.2.1. Spark RDD: Resilient Distributed Dataset

En Spark, a diferencia de Hadoop, no utilizaremos una colección de datos distribuidos en el sistema de archivos, sino que usaremos los RDD (*Resisilient Distributed Datasets*).

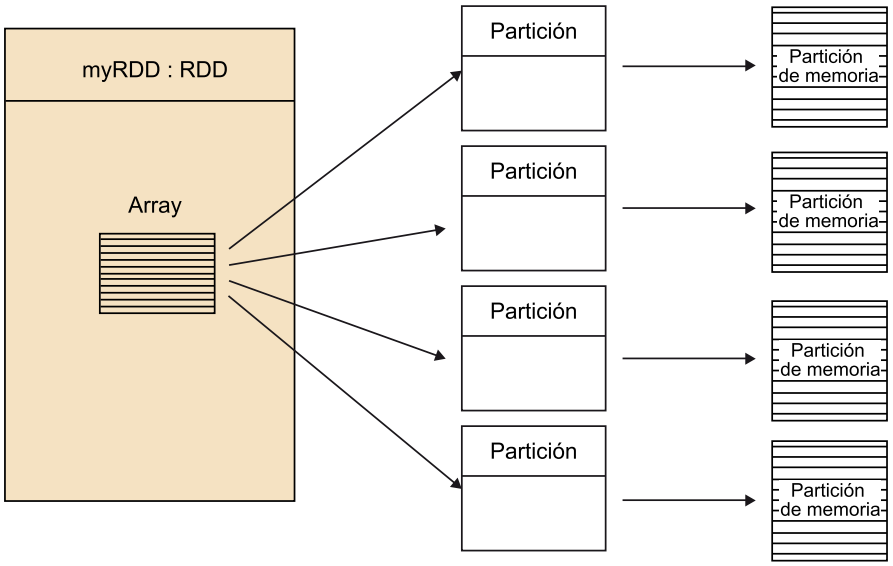
Lectura recomendada

Para una información más detallada sobre RDD consultad http://www.cs.berkeley.edu/matei/papers/2012/nsdi_spark.pdf.

Los RDD son colecciones lógicas, inmutables y particionadas de registros de datos distribuidos en la memoria principal de los nodos del clúster y que pueden ser reconstruidas si alguna partición se pierde (no necesitan ser materializadas pero si reconstruidas para mantener el almacenamiento estable). Se crean mediante la transformación de los datos utilizando para ello transformaciones (*filters, joins, Group by*, etc). Por otra parte permite cachear (guardar) los datos mediante acciones como *Reduce*, *Collect*, *take*, *cache*, *persist*, etc.

Los RDD, descritos en la figura 7, son tolerantes a fallos. Para ello mantiene un registro de las transformaciones realizadas llamado el linaje (*lineage*, en inglés) del RDD. Este *lineage* permite que los RDDs se reconstruyan en caso de que una porción de datos se pierda debido a un fallo en un nodo del clúster.

Figura 7. Representación en memoria principal de un RDD de Spark



Gracias a esto, los RDDs nos proporcionan los siguientes beneficios:

- La consistencia se vuelve más sencilla gracias a la propiedad de inmutabilidad.

- Obtenemos la tolerancia a fallos con un bajo coste, gracias a la idea del linaje y al hecho de poder generar puntos de control (*checkpoints*) gracias a generar acciones, `cache()` y `persist()` concretamente, sobre los RDD que permiten al programador guardar los resultados en la memoria RAM o en el disco duro respectivamente.

- A pesar de ser un modelo restringido a una serie de casos de uso por defecto, gracias a flexibilidad de los RDDs se puede utilizar Spark para una cantidad de aplicaciones muy variadas y donde MapReduce obtiene un rendimiento muy bajo.

4.3. Hadoop y Spark. Compartiendo un mismo origen

Tanto Hadoop como Spark están escritos en Java. Este factor común no es una simple casualidad, sino que tiene una explicación muy sencilla: ambos ecosistemas usan los RMI (Remote Method Invocation) de Java para poder comunicarse de forma eficiente entre los diferentes nodos del clúster.

RMI es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y proporciona un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java.

RMI se caracteriza por su facilidad de su uso en la programación, ya que está específicamente diseñado para Java; proporciona paso de objetos por referencia, recolección de basura distribuida (*Garbage Collector* distribuido) y paso de tipos arbitrarios. A través de RMI, un programa Java puede exportar un objeto, con lo que dicho objeto estará accesible a través de la red y el programa permanecerá a la espera de peticiones en un puerto TCP. A partir de ese momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto. Esto es justo lo que hace el nodo máster en MapReduce para enviar funciones `map()` o `reduce()` a los nodos *worker*. Spark utiliza el mismo sistema para enviar las transformaciones y acciones que se tienen que aplicar a los RDD.

5. Gestor de recursos

Un gestor de recursos distribuidos es un sistema de gestión de colas de trabajos. Permite que varios usuarios, grupos y proyectos puedan trabajar juntos usando una infraestructura compartida como, por ejemplo, un clúster de computación.

5.1. Apache Mesos

Apache Mesos es un gestor de recursos que simplifica la complejidad de la ejecución de aplicaciones en un conjunto compartido de servidores. En su origen, Mesos fue construido como un sistema global de gestión de recursos, siendo completamente agnóstico sobre los programas y servicios que se ejecutan en el clúster.

Colas de trabajos

Una cola no es más que un sistema para ejecutar los trabajos en un cierto orden aplicando una serie de políticas de priorización.

Bajo esta idea, Mesos ofrece una capa de abstracción entre los servidores y los recursos. Es decir, que básicamente lo que nos ofrece Mesos es un lugar donde ejecutar aplicaciones sin preocuparnos de los servidores que tenemos por debajo. Siguiendo la forma de funcionar de MapReduce, dentro de un clúster de Mesos, tendremos un único nodo máster (del que podemos tener réplicas inactivas), que se ocupará de gestionar todas las peticiones de recursos que reciba el clúster. El resto de nodos serán nodos *slaves*, que son los encargados de ejecutar los trabajos de los entornos de ejecución (por ejemplo Spark, Hadoop,...). Estos nodos reportan su estado directamente al nodo máster activo. Es decir, el nodo máster también se encarga del seguimiento y control de los trabajos en ejecución.

5.2. YARN

YARN (Yet Another Resource Negotiator) nace para dar solución a una idea fundamental: dividir las dos funciones principales del *JobTracker*. Es decir, tener en servicios o demonios totalmente separados e independientes la gestión de recursos por un lado y, por otro, la planificación y monitorización de las tareas o ejecuciones.

Un algoritmo de MapReduce, por sí solo, no es suficiente para la mayoría de análisis que Hadoop puede llegar a resolver. Con YARN, Hadoop dispone de un entorno de gestión de recursos y aplicaciones distribuidas donde se pueden implementar múltiples aplicaciones de procesamiento de datos totalmente personalizadas y específicas para realizar una gran cantidad de análisis de forma concurrente.

Daemon

Un *daemon* (nomenclatura usada en sistemas UNIX y UNIX-like) o servicio (nomenclatura usada en Windows) es un tipo especial de proceso informático no interactivo, que se ejecuta en segundo plano en vez de ser controlado directamente por el usuario.

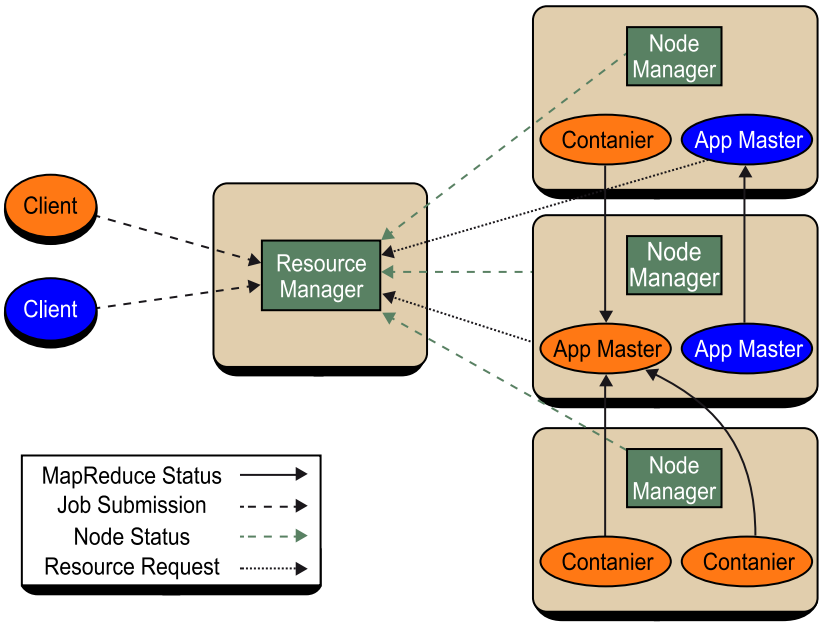
De esta separación surgen dos elementos:

- **Resource Manager (RM)**. Este elemento es global y se encarga de toda la gestión de los recursos.

- **Application Master (AM)**. Este elemento es específico de cada aplicación y se encarga de la planificación y monitorización de las tareas.

Esto deriva en que ahora, una aplicación, es simplemente un Job (en el sentido tradicional de MapReduce).

En la figura 8 se ilustra la arquitectura de YARN para que se pueda entender de forma más clara.



De este modo, el Resource Manager y el Node Manager (NM) esclavo de cada nodo forman el entorno de trabajo, encargándose el Resource Manager de repartir y gestionar los recursos entre todas las aplicaciones del sistema mientras que el Application Master se encarga de la negociación de recursos con el Resource Manager y los Node Manager para poder ejecutar y controlar las tareas, es decir, les solicita recursos para poder trabajar.

6. Escenarios de procesamiento distribuido

Ahora describiremos los dos escenarios de procesamiento de datos distribuidos más comunes. El procesamiento de grandes volúmenes de información en lotes (*batch*) y el procesamiento de datos en flujo (*stream*). Aunque ambos escenarios se incluyen en los entornos de *big data*, poseen importantes diferencias que provocan que no puedan resolverse de la misma forma.

6.1. Procesamiento en *batch*

Un sistema por lotes (en inglés, *batch processing*) se refiere a la ejecución de un programa sin el control o supervisión directa del usuario. Este tipo de programas se caracterizan porque su ejecución no precisa ningún tipo de interacción con el usuario.

Por norma general, este tipo de ejecuciones se utilizan en tareas repetitivas sobre grandes conjuntos de información. Un ejemplo sería el procesamiento de *logs* de un servidor web. En este caso, cada noche se procesarían los ficheros de logs generados por los servidores web durante ese día. Este procesamiento en *bach* se puede realizar de forma sencilla utilizando MapReduce, ya que el conocimiento que nos interesa obtener de los *logs* son valores acumulados, estadísticas y/o cálculos que se combinan con los resultados obtenidos en los días anteriores. En este caso no hay ningún tipo de procesamiento iterativo de los datos.

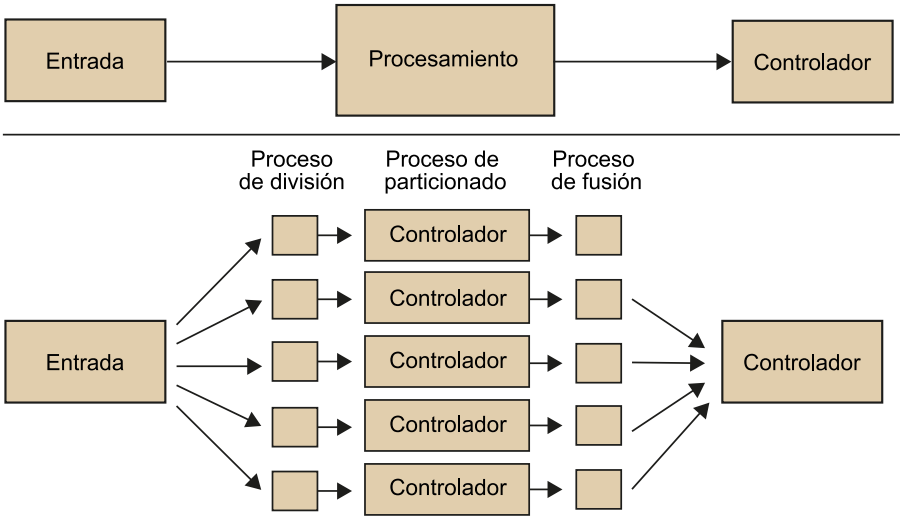
El procesamiento en *batch* requiere el uso de distintas tecnologías para la entrada de los datos, el procesamiento y la salida. En el procesamiento en *batch* se puede decir que Hadoop ha sido la herramienta estrella que ha permitido almacenar cantidades gigantes de datos y escalarlos horizontalmente, añadiendo más nodos de procesamiento en el clúster.

Fichero de logs

Un fichero de *logs*, o "bitácora" en español, es un fichero secuencial que almacena todos los eventos que ha procesado un servidor. Normalmente se guardan en un formato estándar para poder ser procesados de forma sencilla.

En la figura 9 observamos la estructura típica del procesamiento por lotes. En la parte de arriba de la figura observamos cómo se realizaría el procesado sin un sistema de cálculo distribuido, mientras que en la parte de abajo podemos observar cómo se realizaría usando el paradigma de programación basado en MapReduce.

Figura 9. Ejemplo de aplicación de proceso en *batch*



6.2. Procesamiento en *Stream*

Este tipo de técnicas de procesamiento y análisis de datos se basan en la implementación de un modelo de flujo de datos en el que los datos asociados a series de tiempo (hechos) fluyen continuamente a través de una red de entidades de transformación que componen el sistema. En general, y a no ser que necesitemos hacer el procesamiento y análisis de datos en tiempo real, se asume que no hay limitaciones de tiempo obligatorias en el procesamiento en *stream*.

Ejemplo

Un ejemplo es intentar determinar qué clientes de los que están accediendo a una tienda online en un momento determinado tienen más probabilidad de comprar. Una vez se ha determinado cuáles son, se añade una marca en su sesión web y se les asigna una prioridad más alta para que así disfruten de más recursos del servidor web. Esto provocará que su sesión sea más rápida y por tanto su experiencia de compra en la tienda online sea mejor y aumente aún más su probabilidad de compra.

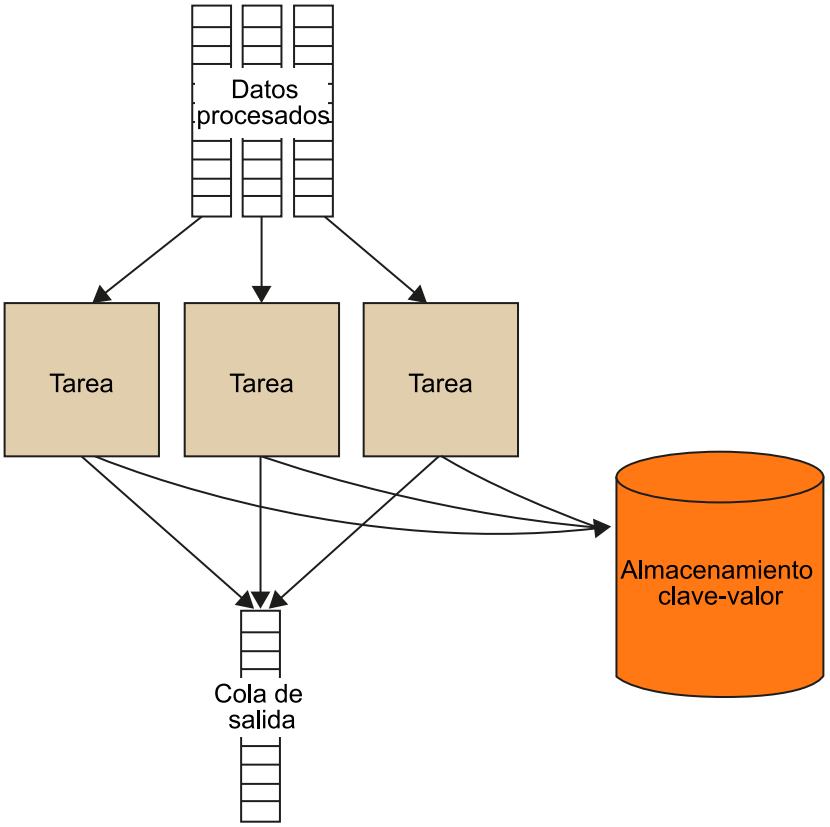
Generalmente, las únicas limitaciones de estos sistemas son:

- Se debe disponer de suficiente memoria para almacenar los datos de entrada en cola.

- La tasa de productividad del sistema a largo plazo debería ser más rápida, o por lo menos igual a la tasa de entrada de datos en ese mismo periodo. Si esto no fuese así, los requisitos de almacenamiento del sistema crecerían sin límite.

Este tipo de técnicas de procesamiento y análisis de datos no está destinado a analizar un conjunto completo de grandes datos, sino a una parte de estos. En la figura 10 observamos un esquema sencillo de procesamiento en *stream* donde los datos procesados se guardan tanto en una cola de salida como en un formato clave-valor para ser procesados posteriormente en *batch*.

Figura 10. Ejemplo de aplicación de proceso en *stream*



Resumen

Todos los expertos apuntan que el *big data* ha aparecido en la escena de las tecnologías de la información para quedarse. Muchos estudios apuntan que la revolución que está generando esta nueva cultura de los datos ha impactado e impactará en todos y cada uno de los negocios actuales y del futuro. Para darse cuenta de cómo de cierta es esta afirmación solo es necesario ver la gran cantidad de *start-ups*, o nuevos negocios, que están apareciendo alrededor de esta nueva cultura de *big data* o *data science*.

En este módulo hemos introducido los principales componentes de una arquitectura *big data*. Hemos empezado hablando de cómo almacenar los datos en sistemas de archivos distribuidos o en bases de datos NoSQL. Seguidamente, hemos pasado a describir dos tecnologías diferentes para poder procesar grandes volúmenes de información de forma eficiente haciendo uso de la computación distribuida. A continuación, hemos explicado cómo se gestionan todos los componentes de un clúster usando un gestor de recursos, como por ejemplo YARN.

Finalmente, hemos mencionado los dos principales paradigmas o escenarios del procesamiento distribuido, el procesamiento en lotes (*batch*) y el procesamiento en flujo (*stream*), explicando sus diferencias y cuáles son sus principales características.

Glosario

API *f* Véase interfaz de programación de aplicaciones

bitácora *f* Véase **fichero de *logs***.

bloque de datos *m* Unidad mínima en la que un fichero almacena información.

call detail record *m* Registro de datos generado en la comunicación entre dos teléfonos fijos o móviles que documenta los detalles de la comunicación (por ejemplo, llamada telefónica, mensajes de texto, etc.). El registro contiene varios atributos como la hora, duración, estado de finalización, número de la fuente o número de destino.
sigla **CDR**

clúster *m* Conjunto de servidores

cola de trabajo *f* Sistema para ejecutar los trabajos en un cierto orden aplicando una serie de políticas de priorización.

código abierto *m* Término con el que se conoce al software distribuido y desarrollado libremente. El código abierto tiene un punto de vista más orientado a los beneficios prácticos de compartir el código que a las cuestiones éticas y morales que resalta el llamado "software libre".
en open source

datos estructurados *m pl* Datos que tienen bien definidos su longitud y su formato, como las fechas, los números o las cadenas de caracteres.
en Structured Data

datos no estructurados *m pl* Datos que en su formato original, carecen de un formato específico
en Unstructured Data

datos semiestructurados *m pl* Datos que no se limitan a un conjunto de campos definidos, como en el caso de los datos estructurados, sino que a su vez contienen marcadores para separar sus diferentes elementos.
en Semistructured Data

daemon *m* Véase **servicio**.

fichero de *logs* *m* Fichero secuencial que almacena todos los eventos que ha procesado un servidor. Normalmente se guardan en un formato estándar para poder ser procesados de forma sencilla.

HashMap *m* Colección de objetos, como un vector o *arrays*, pero sin orden. Cada objeto se identifica mediante algún identificador apropiado. El nombre *hash*, hace referencia a una técnica de organización de archivos llamada *hashing* o dispersión con la cual se almacenan los registros en una dirección que es generada por una función que se aplica sobre la clave del registro.

interfaz de programación de aplicaciones *f* Conjunto de rutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece una biblioteca de programación para que pueda ser utilizada por otro software como una capa de abstracción.
sigla **API**

hard disk drive *m* Sistema de almacenamiento de datos en un disco magnético rotacional.
sigla **HDD**

lenguaje SQL *m* Lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en ellas. *SQL* es el acrónimo en inglés de *structured query language*.

MapReduce *m* Técnica de procesamiento distribuido basada en el concepto de divide y vencerás.

múltiples servidores *m pl* Véase **clúster**.

open source *m* Véase **código abierto**.

procesamiento iterativo *m* Tipo especial de procesamiento que requiere acceder muchas veces a los datos de entrada.

proceso batch *m* Ejecución de un programa sin el control o supervisión directa del usuario. Este tipo de programas se caracterizan porque su ejecución no precisa ningún tipo de interacción con el usuario.

regla ACID *f* En el contexto de bases de datos ACID (acrónimo inglés de *atomicity, consistency, isolation, durability*) es una serie de propiedades que tiene que cumplir todo sistema de gestión de bases de datos para garantizar que las transacciones sean fiables. Para una explicación más detallada podéis consultar la siguiente entrada de la Wikipedia: http://es.wikipedia.org/wiki/ACID.

remote method invocation *m* Tecnología Java para ejecutar código arbitrario de forma remota en un ordenador
sigla **RMI**

servicio *m* Tipo especial de proceso informático no interactivo que se ejecuta en segundo plano en vez de ser controlado directamente por el usuario. Es un término que se utiliza principalmente en Windows. En sistemas UNIX y UNIX-like se utiliza habitualmente *daemon*.
en daemon

shuffling *m* Técnica que permite agrupar un conjunto de datos que comparten ciertas características.

solid state disk *m* Sistema de almacenamiento de datos permanente con acceso directo a los datos.
sigla **SSD**

tolerancia a fallos *f* Propiedad que permite a un sistema computacional recuperarse de un fallo de hardware