Tipologías y arquitecturas de un sistema *big data*

PID_00250686

Jordi Nin

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Índice

Introducción

Es un hecho natural que cada día generamos más y más datos, y que su captura, almacenamiento y procesamiento son piezas fundamentales en una gran variedad de situaciones, ya sean de ámbito empresarial o con la finalidad de realizar algún tipo de investigación científica.

Para conseguir estos objetivos, 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 areas 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 de estado sólido o SSD (*Solid State Disk*), que permiten acceder de forma eficiente a grandes conjuntos de datos— o el desarrollo de redes de computadores más rápidas y eficientes —basadas por ejemplo, en fibra óptica, que permiten compartir gran cantidad de datos entre multiples servidores—, hasta nuevas metodologías de programación que permiten a los desarrolladores e investigadores usar estos nuevos componentes de hardware de una forma relativamente sencilla.

En general, toda arquitectura de *big data*, requiere de una gran cantidad de servidores, generalmente ordenadores de propósito general como los que tenemos en nuestras casas. 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, tanto de datos capturados del mundo real como resultados ya procesados.

El principal objetivo de una arquitectura de *big data* es que todos estos elementos —CPU, memoria y disco— sean accesibles de forma distribuida haciendo su uso transparente para el usuario y proveyendo una falsa sensación de centralidad, es decir, que para el usuario de la infraestructura solo haya un único conjunto de CPU, una única memoria central y un sistema de almacenamiento central. Para permitir esta abstracción es necesario que los datos se encuentren distribuidos y copiados diversas veces en diferentes servidores y que el entorno de programación permita distribuir los cálculos que se han realizar de forma sencilla a la vez que eficiente.

En primer lugar, debemos ser capaces de capturar y almacenar los datos disponibles. Actualmente, los sistemas de gestión de archivos distribuidos más habituales que encontramos en arquitecturas de *big data* son bases de datos relacionales, como Oracle, PostgreSQL o IBM-bd2; bases de datos NoSQL como

Apache Cassandra, Redis o mongoDB; y sistemas de ficheros de texto como HDFS (Hadoop Distributed File System), que permiten almacenar grandes volúmenes de datos. Estos sistemas son los encargados de almacenar y permitir el acceso a los datos de forma eficiente. En paralelo, estos sistemas permiten guardar los resultados parciales generados durante el procesamiento de los datos y los resultados finales en el caso que estos ocupen también mucho espacio.

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

- **Hadoop MapReduce**. Diseñado inicialmente por Google con código propietario y liberado posteriormente por Yahoo! como código abierto o *open source*. 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.

- **GPU Computing**. Esta forma de procesamiento puede definirse como el uso de una unidad de procesamiento gráfico (GPU) disponible en cualquier ordenador en combinación con una CPU para acelerar aplicaciones de análisis de datos e ingeniería. Estos sistemas suelen emplearse en situaciones donde se requiere una gran cantidad de cálculos.

Todos estos entornos se enfrentan a dos grandes retos:

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 módulo extenderemos todas estas ideas y veremos cómo es posible afrontar dichos retos.

Objetivos

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

1. Comprender los diferentes componentes hardware de una arquitectura de *big data*.
2. Conocer el *stack* de software típico de gestión de una arquitectura de *big data*.
3. Entender cómo se almacenan y distribuyen los datos masivos en un sistema de archivos distribuido.
4. Entender las diferentes jerarquías de memoria para poder procesar datos masivos de forma eficiente.
5. Ser capaz de diferenciar los diferentes tipos de procesamiento distribuido: modelo *batch* (por lotes) frente al modelo *streaming* (secuencial).

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

Big data o **datos masivos** 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 de los datos.

Bibliografía complementaria

Viktor Mayer-Schönberger; Kenneth Cukier (2013). *Big Data: A Revolution That Will Transform How We Live, Work and Think*. John Murray Publishers Ltd

Generalmente, en los documentos científico-técnicos en español se usa directamente el término en inglés *big data*, tal y como aparece en el artículo seminal de Viktor Schönberger «Big data: La revolución de los datos masivos».

Datos masivos es un término que hace referencia a un volumen de datos tan grande que supera la capacidad tanto del hardware como 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 el artículo de Viktor Schönberger se estimaba su tamaño de entre una docena de terabytes hasta varios petabytes de datos en un único conjunto de datos. Hoy en día este tamaño se queda pequeño, ya que podemos almacenar varios terabytes de información en los ordenadores convencionales.

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 y procesar. Generalmente diferenciamos tres tipos de fuentes de datos masivos. Nótese que esta clasificación se aplica tanto a datos masivos como a no masivos:

- **Datos estructurados (*Structured Data*)**. Son 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*)**. Son 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 vídeo), correos electrónicos, etc.

- **Datos semiestructurados (*Semistructured Data*)**. Son datos que no se limitan a un conjunto de campos definidos como en el caso de los datos estructurados, sino que 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 cómo 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.

Datos semiestructurados

Encontramos diferentes tipos de codificaciones para los datos semiestructurados como son:
1) HTML. El HyperText Markup Language es un lenguaje de programación que se utiliza para el desarrollo de páginas de Internet.
2) XML. El Extended Markup no es un lenguaje en sí mismo, sino un sistema que permite definir lenguajes de acuerdo a las necesidades.
3) JSON. El JavaScript Object Notation es un formato ligero de intercambio de datos pensado para que los ordenadores les resulte simple interpretarlo y generarlo.

Muchas veces encontramos el concepto de *big data* relacionado con diferentes conceptos diferentes conocidos como *las V del big data*:

- **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**. Aunque los datos no sufren variaciones muy frecuentes, su análisis puede llevar horas e incluso días con técnicas tradicionales sin ser 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.

Las V del *big data*

Las tres primeras V aparecen en la definición original de *big data*, más adelante se incorporó la cuarta. Aún a día de hoy esta cuarta V no está aceptada por todo el mundo.

- **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.

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

La respuesta a esta pregunta es sencilla, en todos los ámbitos de conocimiento, como por ejemplo:

- **Redes sociales**. Su uso, que cada vez está más extendido, hace que los 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.

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

- **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* –que se puede traducir como '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 más de antelación respecto a los sistemas tradicionales.

* http://bit.ly/2CXuECR

- **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 ayudó a la victoria de Barak Obama frente al otro candidato republicano, Mitt Romney, con el 51,06 % de los votos, siendo esta una de las elecciones presidenciales más disputadas.

** http://bit.ly/2yTn9K1

- **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 (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.

- **Finanzas**. Los grandes bancos disponen de sistemas de *trading* algorítmico que analizan una gran cantidad de datos de todo tipo para decidir qué operaciones en bolsa son las más rentables en cada momento.

1.2. Datos, información y conocimiento

***Trading* algorítmico**

El *trading* algorítmico es una modalidad de operación en mercados financieros (*trading*) que se caracteriza por el uso de algoritmos, reglas y procedimientos automatizados en diferentes grados, para ejecutar operaciones de compra o venta de instrumentos financieros.

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.

En general, decimos que los **datos** son la mínima unidad semántica y se corresponden con elementos primarios de información que por sí solos son irre-

levantes como apoyo a la toma de decisiones. El saldo de una cuenta corriente o el número de hijos de una persona, por ejemplo, son datos que, sin un propósito, una utilidad o un contexto, no sirven como base para apoyar la toma de una decisión.

Por el contrario, hablaremos de **información** cuando obtenemos un conjunto de datos procesados y que tienen un significado (relevancia, propósito y contexto) y que, por lo tanto, son de utilidad para quién debe tomar decisiones, al disminuir su incertidumbre.

Finalmente, definiremos **conocimiento** como una mezcla de experiencia, valores e información que sirve como marco para la incorporación de nuevas experiencias e información y es útil para la toma de decisiones.

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 usando dicho conocimiento ayude a optimizar los procesos de negocio.

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 que se ocupa tanto de distribuir los datos como los resultados obtenidos durante el cómputo.

Las principales características de este modelo son:

- La forma de trabajar de los usuarios en la infraestructura de *big data* 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 desarrolló en 2004 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 propició un ecosistema de herramientas *open source* de procesamiento de grandes volúmenes de datos.

*** http://bit.ly/15V7Pyh**

Open source

Open source (en castellano, '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, que 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 que se han de 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

Un ejemplo de cálculo iterativo es el 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 creo 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.

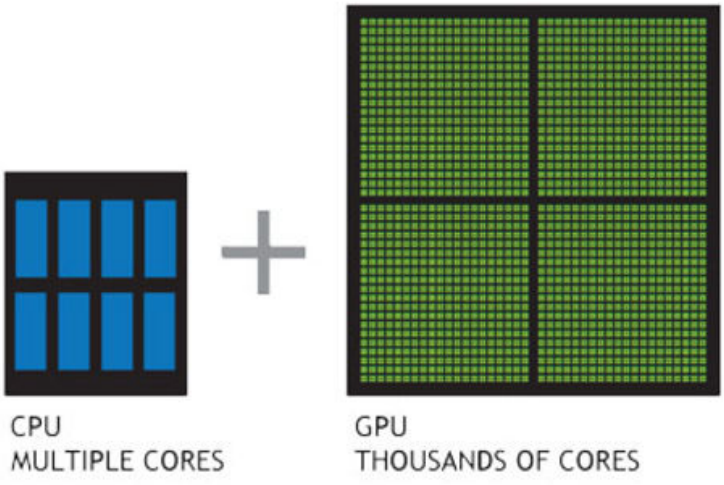
**** http://spark.apache.org**

En paralelo a estas mejoras desde el año 2007, empresas como NVIDIA, han empezado a introducir el uso de GPU (unidad de procesamiento gráfico) como alternativa al cálculo tradicional en CPU (unidad central de procesamiento). Las GPU disponen de procesadores mucho más simples que las CPU. Esto faci-

lita que en una sola tarjeta gráfica instalada en un servidor puedan integrarse un mayor número de procesadores permitiendo realizar un gran número de cálculos numéricos en paralelo.

Una forma sencilla de entender la diferencia entre la GPU y la CPU es comparar la forma en que procesan las tareas. Una CPU está formada por varios núcleos optimizados para el procesamiento en serie, mientras que una GPU consta de millares de núcleos más pequeños y eficientes diseñados para manejar múltiples tareas simultáneamente.

Figura 1. Diferencias entre una arquitectura basada en CPU y otra en GPU



Fuente: http://www.nvidia.com/object/what-is-gpu-computing.html

En general, una CPU y una GPU son lo mismo: circuitos integrados con una gran cantidad de transistores que realizan cálculos matemáticos leyendo números en binario. La diferencia es que la CPU es un procesador de propósito general, con el que podemos hacer cualquier tipo de cálculo, mientras que la GPU es un procesador de propósito específico: está optimizada para trabajar con grandes cantidades de datos y realizar las mismas operaciones, una y otra vez.

Por eso, aunque ambas tecnologías se dediquen a realizar cálculos, tienen un diseño sustancialmente distinto. La CPU está diseñada para el procesamiento en serie: se compone de unos pocos núcleos muy complejos que pueden ejecutar unos pocos programas al mismo tiempo. En cambio, la GPU tiene cientos o miles de núcleos sencillos que pueden ejecutar cientos o miles de programas específicos a la vez. Las tareas de las que se encarga la GPU requieren un alto grado de paralelismo: tradicionalmente, una instrucción y múltiples datos.

1.4. Computación científica

La computación científica es el campo de estudio relacionado con la construcción de modelos matemáticos, algoritmos y técnicas numéricas para resolver problemas científicos, de análisis de datos y problemas de ingeniería. Típicamente se basa en la aplicación de diferentes formas de cálculo a problemas de varias disciplinas científicas.

Este tipo de computación requiere una gran cantidad de cálculos (usualmente de punto flotante) y normalmente se ejecutan en superordenadores o plataformas de computación distribuida como las descritas anteriormente, ya sea utilizando procesadores de uso general o CPU, o bien utilizando procesadores gráficos (GPU) adaptados al computo numérico, como hemos descrito en el subapartado anterior.

Las principales aplicaciones de la computación científica son:

- **Simulaciones numéricas.** Los trabajos de ingeniería que se basan en la simulación buscan mejorar la calidad de los productos –por ejemplo, mejorar la resistencia al viento de un nuevo modelo de coche– y reducir los tiempos y costes de desarrollo, ya que todos los cálculos se realizan en un ordenador y no es necesario fabricar nada. Muchas veces, esto implica el uso de simulaciones con herramientas de ingeniería asistida por ordenador (CAE) para operaciones de análisis de mecánica estructural/elementos finitos, dinámica de fluidos computacional (CFD) y/o electromagnetismo (CEM). En general, estas simulaciones requieren una gran cantidad de cálculos numéricos debido a la dificultad (o imposibilidad) de resolver las ecuaciones de forma analítica.

- **Análisis de datos.** Como hemos visto, el análisis de datos masivos ayuda en gran medida a la toma de decisiones de negocio. Uno de los problemas de los métodos de aprendizaje automático o *Machine Learning* es que los métodos más complejos, como las redes neuronales –también conocidas como *Deep Learning*–, requieren de mucho tiempo de computo para ser entrenados. Esto afecta a su utilidad en escenarios donde el *time to market* es extremadamente bajo y, por tanto, no se dispone de una gran cantidad de tiempo para entrenar las redes neuronales. Por suerte, este tipo de métodos se basan en entrenar una gran cantidad de neuronas/perceptrones en paralelo. Es aquí donde las GPU ayudan a realizar este entrenamiento de forma mucho más rápida gracias a su arquitectura.

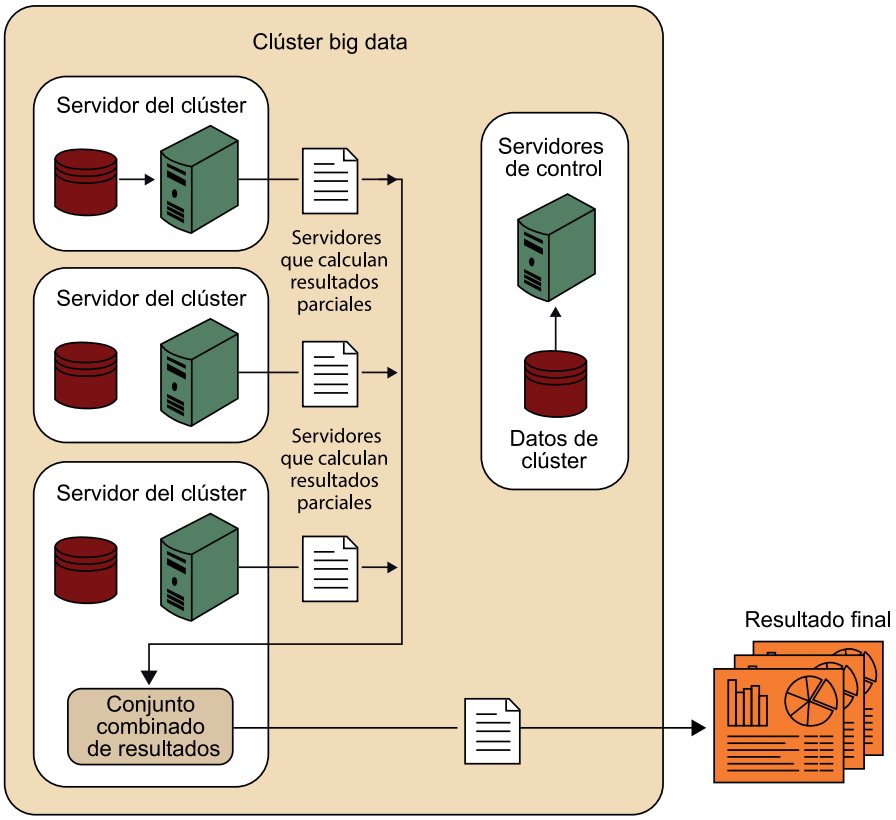
- **Optimización.** Una tarea de optimización implica determinar los valores para una serie de parámetros de tal manera que, bajo ciertas restricciones, se satisfaga alguna condición –por ejemplo, buscar los parámetros de una red neuronal que minimize el error de clasificación, encontrar los precios de renovación de un conjunto de seguros que maximize el beneficio en la

renovación de las pólizas, etc. Hay muchas tipos de optimización, tanto lineales como no lineales. El factor común de todos estos tipos es que requieren usar un conjunto de algoritmos, como el descenso del gradiente que exige de un gran volumen de cálculos repetitivos que han de realizarse sobre un conjunto determinado de datos. Otra vez, nos encontramos con el escenario de cálculos sencillos repetidos en paralelo sobre un conjunto, posiblemente muy grande, de datos.

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

Aunque todas las infraestructuras de *big data* tienen características especificas que adaptan el sistema a los problemas que tienen que resolver, todas comparten unos pocos componentes comunes, tal y como se describe en la figura 2.

Figura 2. 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 éter.

En la figura 2 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 caso que detecten un mal funcionamiento, fuerzan el reinicio del servidor y avisan al administrador de que hay problemas en ciertos nodos.

El uso combinado de este conjunto de servidores es posible, como veremos en este módulo, gracias al uso de un sistema de ficheros distribuido y al hecho de que los cálculos que se han de realizar se implementan en un entorno de programación distribuida, como por ejemplo *Hadoop* o *Spark*.

2.1. Estructura de un servidor con GPU

En un servidor equipado con GPU la gestión de la memoria es un poco más compleja que en un servidor con CPU, ya que para poder obtener el máximo rendimiento de todos los procesadores de una GPU es necesario garantizar que todos estos procesadores reciben un flujo de datos constante durante la ejecución de las tareas del clúster.

En la figura 3 describimos las diferentes memorias que dispone un servidor con GPU, así como sus flujos de información. Concretamente, el procesamiento en estos equipos implica el intercambio de datos entre HDD (disco duro), DRAM, CPU y GPU.

DRAM

DRAM corresponde a las siglas del inglés *Dynamic Random Access Memory*, que significa memoria dinámica de acceso aleatorio (o RAM dinámica), para denominar a un tipo de tecnología de memoria RAM basada en condensadores, los cuales pierden su carga progresivamente, necesitando de un circuito dinámico de refresco que, cada cierto período, revisa dicha carga y la repone en un ciclo de refresco. En oposición a este concepto surge el de memoria SRAM (RAM estática), con la que se denomina al tipo de tecnología RAM basada en semiconductores que, mientras siga alimentada, no necesita refresco. DRAM es la tecnología estándar para memorias RAM de alta velocidad.

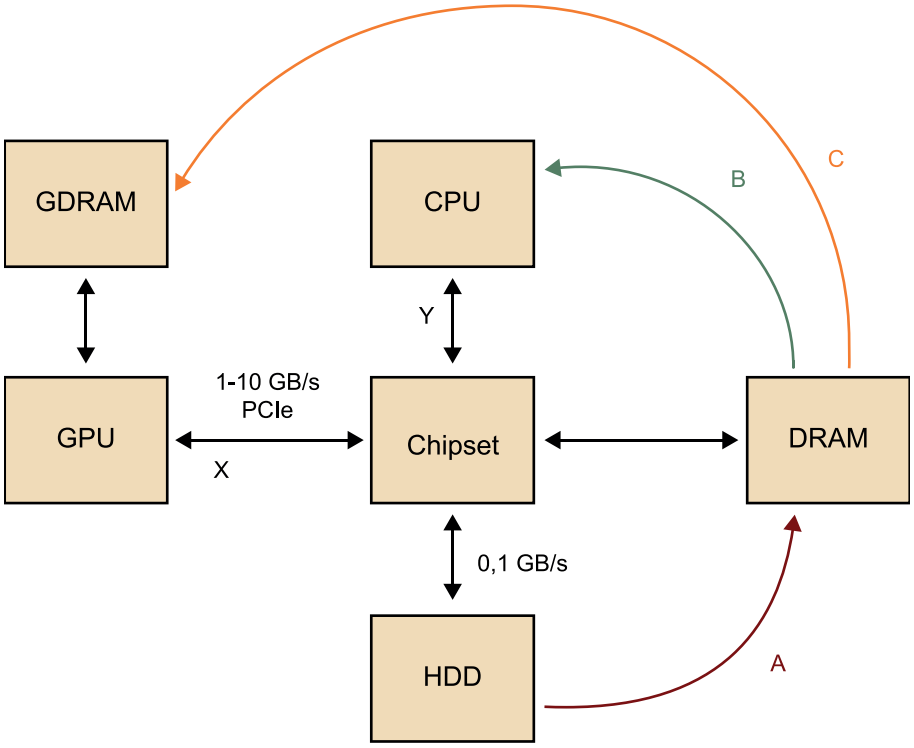
La figura 3 muestra cómo se transfieren los datos cuando un servidor realiza cálculos con una CPU y una GPU. Si observamos los diferentes flujos de información vemos:

- **Flecha A**. Transferencia de datos de un disco duro a memoria principal (paso inicial común tanto para la computación en CPU y GPU)

- **Flecha B**. Procesamiento de datos con una CPU (transferencia de datos: DRAM —→ chipset —→ CPU)

- **Flecha C**. Procesamiento de datos con una GPU (transferencia de datos: DRAM —→ chipset —→ CPU —→ *chipset* —→ GPU —→ GDRAM —→ GPU)

Figura 3. Jerarquía de la memoria en un servidor equipado con GPUs



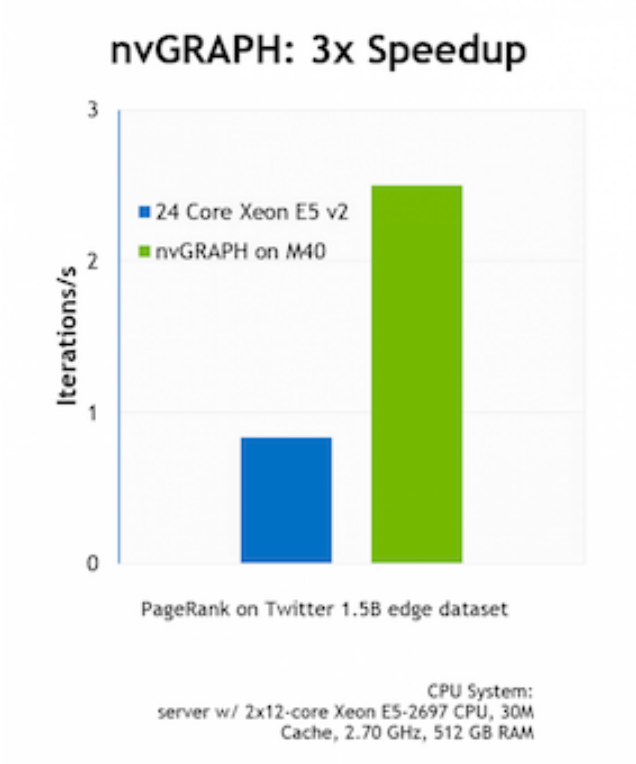
Como resultado de esto, la cantidad total de tiempo que necesitamos para completar cualquier tarea incluye:

- La cantidad de tiempo requerido para que una CPU o una GPU lleven a cabo sus cálculos.

- Más la cantidad de tiempo dedicado a la transferencia de datos entre todos los componentes. Este punto es crucial en el caso de las GPU, tanto por la cantidad de datos que han de transferirse en paralelo como por el aumento de los componentes intermedios necesarios.

Es fácil encontrar en Internet comparaciones sobre los tiempos de ejecución de una tarea en GPU y en CPU. Por ejemplo, en la figura 4 podemos observar los tiempos de ejecución del algoritmo de *PageRank* sobre una red social. Como estas comparaciones varían cada poco tiempo, aquí nos centraremos en describir cuál es el impacto que tiene la transferencia de datos a una GPU y así poder valorar si el uso de GPU es viable y/o adecuado.

Figura 4. Jerarquía de la memoria en un servidor equipado con GPUs



Fuente: NVIDIA

Aunque es muy probable que cuando usamos un supercomputador esté optimizado para trabajar con GPU, un servidor estándar puede ser mucho más lento cuando intercambia datos de un tipo de almacenamiento a otro. Mientras que la velocidad de transferencia de datos entre una CPU estándar y el *chipset* de cómputo es de 10-20 GBps* (punto Y en la figura 3), una GPU intercambia datos con DRAM a la velocidad de 1-10 GBps (véase el punto X de la misma figura). Aunque algunos sistemas pueden alcanzar hasta unos 10 GBps (PCIe v3)**, en la mayoría de las configuraciones estándar los flujos de datos entre una GPU (GDRAM) y la DRAM del servidor dispone de una velocidad aproximada de 1 GBps.

*** GBps es el símbolo de GigaBytes per second.**

**** PCIe v3 es la abreviatura de Peripheral Component Interconnect Express de tercera generación.**

Por tanto, aunque una GPU proporciona una computación más rápida, el principal cuello de botella es la velocidad de transferencia de datos entre la memoria de la GPU y la memoria de la CPU (punto X). Por este motivo, para cada proyecto en particular, es necesario medir el tiempo dedicado a la transferencia de datos desde/hacia una GPU con el tiempo ahorrado debido a la aceleración de la GPU. Por este motivo, lo mejor es evaluar el rendimiento real en un pequeño conjunto de datos para luego poder estimar cómo se comportará el sistema en una escala mayor.

A partir del punto anterior podemos concluir que dado que la velocidad de transferencia de datos puede ser bastante lenta, el caso de uso ideal es cuando la cantidad de datos de entrada/salida para cada GPU es relativamente pequeña en comparación con la cantidad de cálculos que se deben realizar. Es

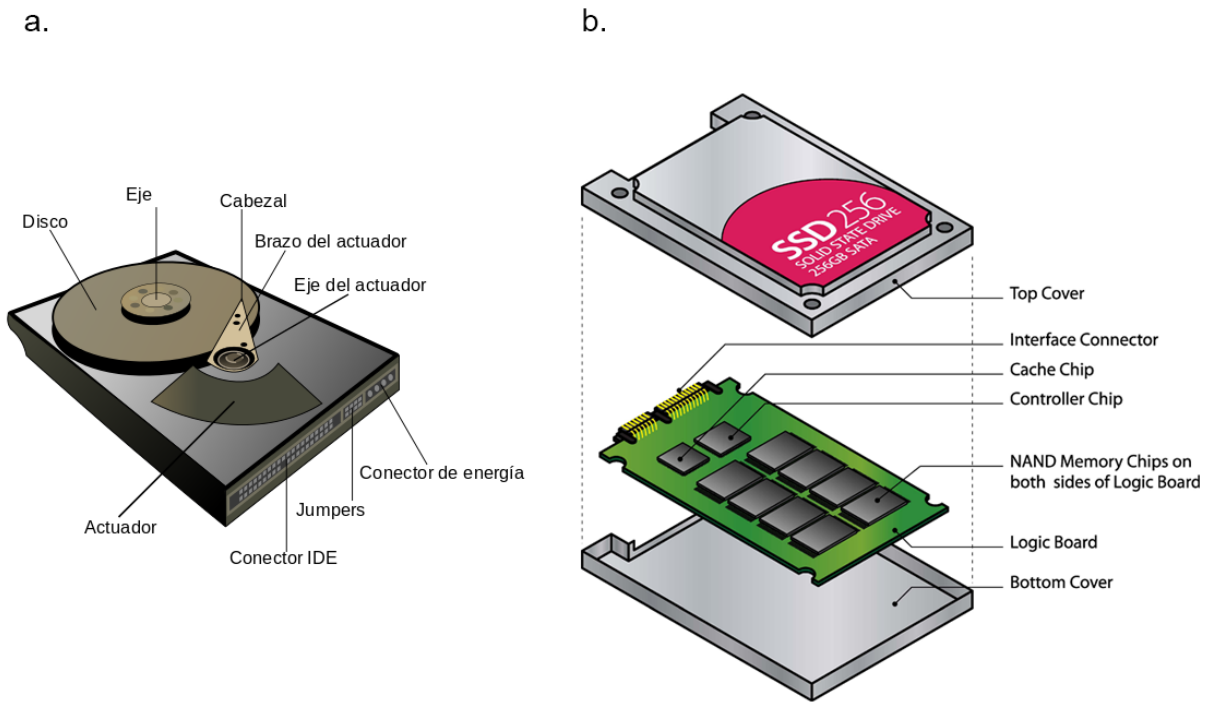
importante tener en cuenta que, primero, el tipo de tarea debe coincidir con las capacidades de la GPU; y, segundo, la tarea se puede dividir en subprocesos independientes paralelos como en MapReduce. Este segundo punto favorece la idea de poder realizar una gran cantidad de cálculos en paralelo como hemos descrito en el apartado anterior.

3. Sistema de archivos

Una vez que ya hemos visto la arquitectura general de un sistema de *big data* y que hemos hablado de la diferencia entre cómputo en CPU y GPU, y de qué características deben tener las diferentes jerarquías de memoria en un servidor, es el momento de centrarnos en cómo se han de almacenar los datos para poder aplicar cálculos sobre estos.

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 –en inglés, *Hard Disk Drive* (HDD)– o en memorias de estado sólido –en inglés, *Solid State Disk* (SDD). En general, es extraño en sistemas de *big data* usar almacenamiento terciario como DVD o CD-ROM, ya que estos no permiten el acceso a la información de forma distribuida. En la figura 5 podemos observar una comparación entre ambas tecnologías.

Figura 5. Comparación entre un disco duro magnético y una memoria de estado sólido



De forma breve, podemos decir que un disco duro tradicional (figura 5a) 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 dónde realizar la operación del 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 5b) sí que disponen de acceso aleatorio a la información almacenada y, en general, son bastante más rápidas, aunque 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 son 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 el **nombre** de archivo seguido de una **extensión** (por ejemplo, *.txt*) que indica el contenido del archivo.

Ejemplo de ruta

home/user/mydata/data.csv es un ejemplo de 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, tal y como la acabamos de describir, 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 no todos los sistemas de archivos son una opción válida. 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, aunque en la actualidad se utiliza en casi todos los sistemas *big data* (por ejemplo, Hadoop, Spark, Flink, Kafka, Flume, etc). En el siguiente subapartado introduciremos su funcionamiento y sus principales componentes.

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*** (que se puede traducir por '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.

Tamaño de información en HDFS

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.

Proceso *batch*

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

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*.

Logo del Hadoop Distributed File System (HDFS)

Las arquitecturas HDFS tienen 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:

- ***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

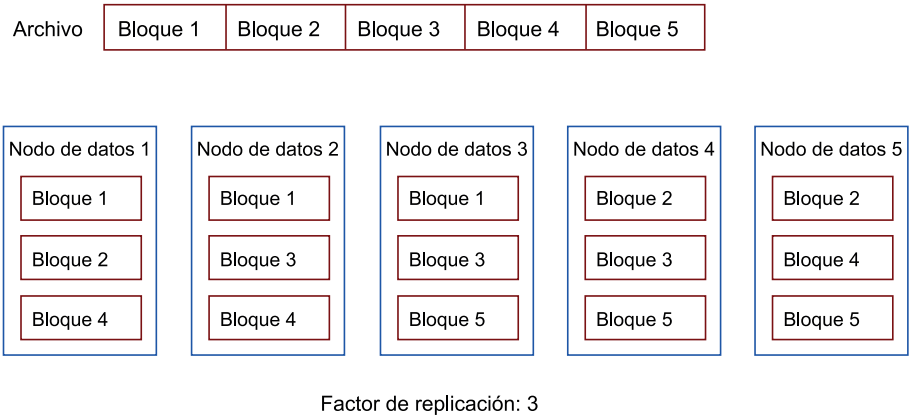
dicho. En este caso, almacenan los bloques de información y los recuperan bajo demanda.

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 cuántas 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 6, 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*. De esta forma permite que podamos terminar los procesos en curso, incluso si fallan dos nodos de los cinco que dispone el clúster.

Figura 6. Ejemplo de almacenamiento en HDFS



3.2. Bases de datos NoSQL

Hasta hace unos años, las bases de datos relacionales han 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.

* http://bit.ly/2DPRlsv

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 en ellas diversos tipos de operaciones.

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.

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 concurrente tanto para consultar como para generar nuevos datos.

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*.

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 funcionamiento en almacenar la información en columnas, en lugar de filas, 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 pretenden sustituir a las bases de datos relacionales, 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 de 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 son 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 es 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

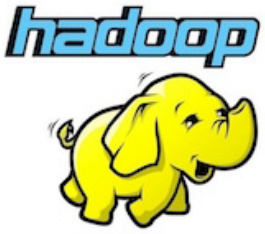
Los sistemas de cálculo distribuido permiten la integración de los recursos de diferentes máquinas en red, convirtiendo la ubicación de un recurso en algo transparente al usuario. El usuario accede a los recursos del sistema distribuido a través de un gestor de recursos, despreocupándose de dónde se 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. Además también introduciremos cómo se pueden aplicar las ideas del cálculo distribuido sobre arquitecturas basadas en GPU.

4.1. Paradigma MapReduce

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) la utilización de clústers de ordenadores y

2) la utilización de hardware no especializado.



Logo de Apache Hadoop

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 fáciles 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* (en castellano, 'trabajadores'), excepto uno que toma el rol de *master* (en castellano, '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 y completo. El rol de maestro se asigna de manera aleatoria en cada ejecución.

Ahora veamos con un poco más en 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 maestro 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 7, 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 maestro, el resto de nodos serán considerados *workers*. El nodo maestro 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* 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 esté 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 regiones son enviadas al nodo maestro, que se encarga de reenviar los nuevos datos a los nodos *worker* que tengan asignadas tareas de `Reduce()`.

- Cuando el nodo maestro notifica a un nodo *worker* de tipo *Reduce* la localización de una partición, este 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()`.

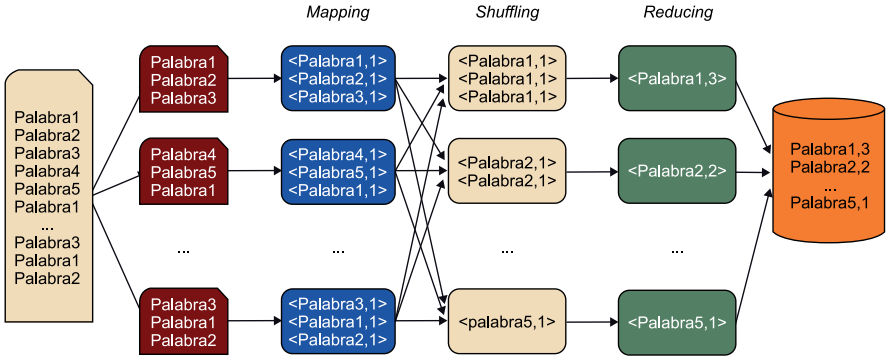
- Los nodos *worker* de tipo `Reduce()` iteran sobre el conjunto de valores ordenados intermedios para cada una de las claves únicas encontradas. Para poder realizar este paso, es necesario que la función `Reduce()` conozca el conjunto de valores asociados a cada clave. La salida de esta función se añade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

Observemos la figura 7. En este caso 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 igual a 1. Posteriormente, se agrupan estas estructuras clave-valor que comparten la misma clave en un mismo nodo. Finalmente, la función `Reduce()` se ocupa en 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 7. Ejemplo de proceso en MapReduce



Uno de los aspectos más importantes de esta forma de realizar cálculos es que es **tolerante a fallos**. Cuando en uno de los nodos *workers* se produce un fallo, el nodo maestro 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 trabajo asignado a ese nodo *worker* y se reasigna a otro nodo para que lo realice.

4.2. Apache 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 se han de 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

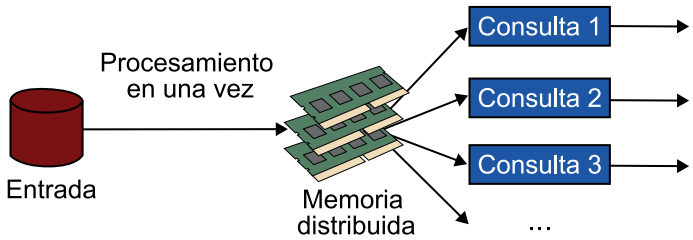
El proyecto de Spark se centró desde el comienzo en aportar una solución factible a estos defectos de Hadoop, mejorando el comportamiento de las aplicaciones que hacen uso de MapReduce y aumentando su rendimiento considerablemente.

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.

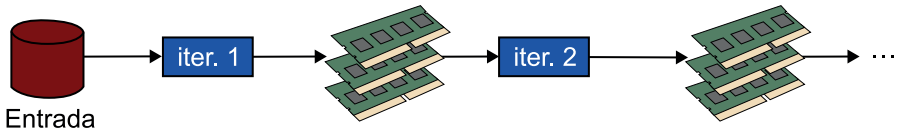
API

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.

Figura 8. Comparación de una ejecución de Spark y Hadoop



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



b. Algoritmos iterativos

La principal ventaja de Spark respecto a Hadoop es que guarda en memoria todas las operaciones sobre los datos. Esta es la clave de su buen rendimiento. La figura 8 muestra algunas de sus principales características:

- Baja latencia computacional mediante el cacheo de los datos en memoria (figura 8a).
- Los algoritmos iterativos se ejecutan de forma eficiente gracias a que las sucesivas operaciones comparten los datos en memoria (figura 8b).

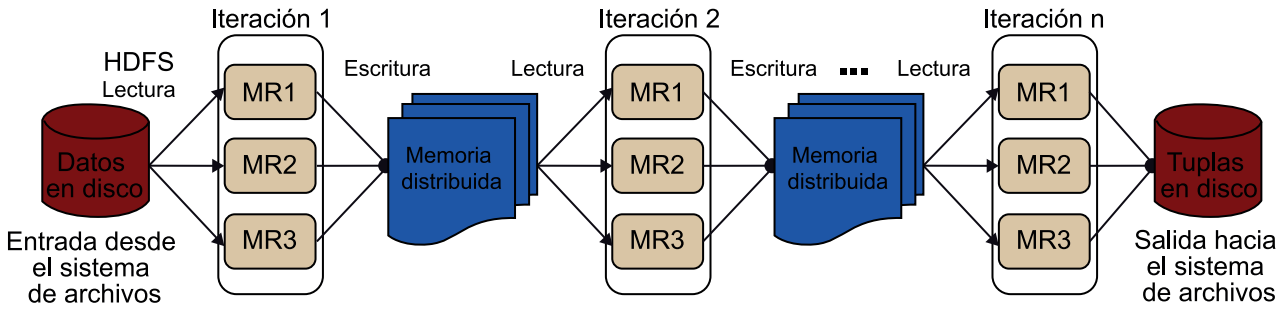
En la figura 9 se ilustra cómo se realiza la ejecución de un programa en Spark. Una ejecución típica de Spark se organiza de la siguiente manera:

1) A partir de una variable de entorno llamada *spark context* que define cómo está organizado el clúster, se crea un objeto RDD —que almacena un conjunto de datos en memoria principal de forma distribuida— leyendo datos del sistema de ficheros (que podría ser perfectamente HDFS), una base de datos o cualquier otra fuente de información en forma de 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 un nuevo RDD en cada operación.

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 9. Flujo de ejecución de Spark



4.2.1. Resilient Distributed Dataset (RDD)

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

Enlace de interés

Para una información más detallada sobre RDD consultad la siguiente dirección web: http://bit.ly/1ajlZop.

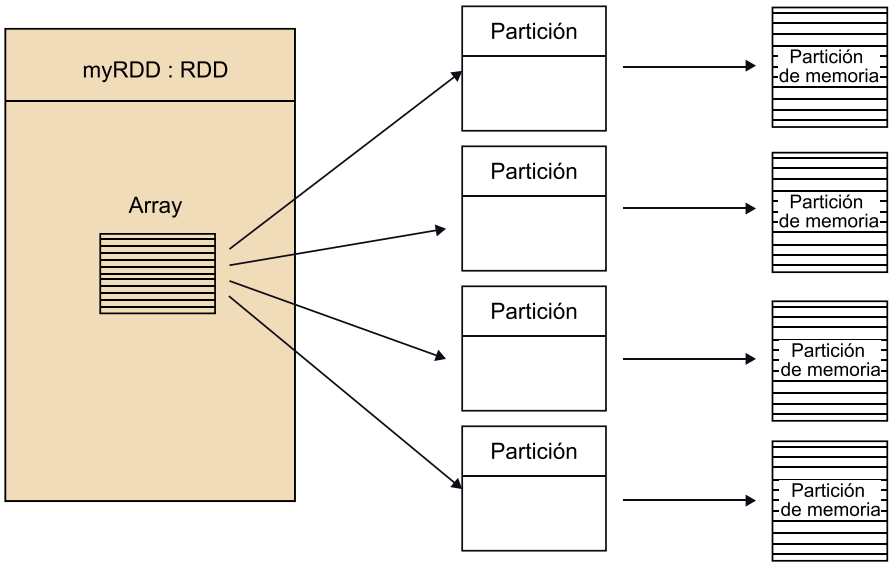
Los RDD son colecciones lógicas, inmutables y particionadas de registros de datos distribuidos en la memoria principal de los nodos del clúster y que

pueden ser reconstruidas si alguna partición se pierde*. Se crean mediante la transformación de los datos utilizando para ello transformaciones (*filters, joins, group by*, etc). Por otra parte permite cachear –es decir, guardar– los datos mediante acciones como *reduce, collect, take, cache, persist,* etc.

*** No necesitan ser materializadas pero si reconstruidas para mantener el almacenamiento estable.**

Los RDD, descritos en la figura 10, son tolerantes a fallos. Para ello mantienen un registro de las transformaciones realizadas, llamado el linaje (*lineage*, en inglés) del RDD. Este linaje permite que los RDD se reconstruyan en caso de que una porción de datos se pierda por un fallo en un nodo del clúster.

Figura 10. Representación en memoria principal de un RDD de Spark



Gracias a esto, los RDD 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 la flexibilidad de los RDD 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 un explicación muy sencilla: ambos ecosistemas usan los Remote Method Invocation (RMI) 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 la 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 será 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 maestro 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.

4.4. GPU: simplicidad y alta paralelización

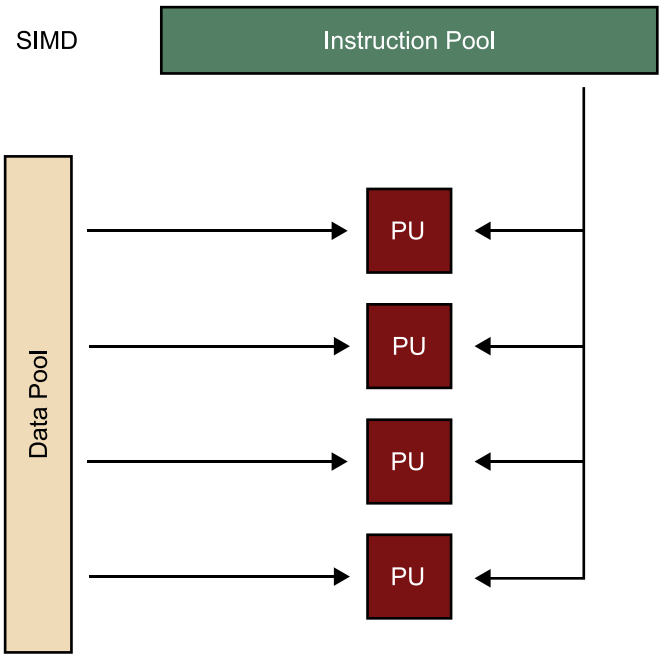
Como hemos comentado en apartados anteriores, los servidores con GPU se basan en arquitecturas **many-core**. Estas arquitecturas disponen de una cantidad ingente de núcleos (procesadores) que realizan una misma operación sobre múltiples datos.

En computación esto se conoce como SIMD (*Single Instruction, Multiple Data*). SIMD es una técnica empleada para conseguir un gran nivel de paralelismo a nivel de datos. Por este motivo en subapartados anteriores nos hemos ocupado de describir cómo se organizan los flujos de datos dentro de un servidor con GPU, ya que la alta disposición de datos en los registros de los múltiples procesadores es un elemento esencial para poder obtener mejoras en el rendimiento de aplicaciones basadas en GPU.

Los repertorios SIMD consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos. Es una organización en donde una única unidad de control común despacha las instrucciones a diferentes unidades de procesamiento. Todas estas reciben la misma instrucción, pero operan sobre diferentes conjuntos de datos. Es decir, la misma instruc-

ción es ejecutada de manera sincronizada por todas las unidades de procesamiento. La figura 11 describe de forma gráfica esta idea.

Figura 11. Estructura funcional del concepto «Una instrucción, múltiples datos»



Fuente: https://es.wikipedia.org/wiki/SIMD

El uso de estas arquitecturas hace posible definir la idea de **GPGPU** o computación de propósito general sobre procesadores gráficos. Esto nos permite realizar el cómputo de aplicaciones que tradicionalmente se ejecutaban sobre CPU en GPU de forma mucho más rápida.

Lenguajes de programación como CUDA, desarrollado por NVDIA, permiten utilizar una plataforma de computación paralela con GPU para realizar computación general a una gran velocidad. Con lenguajes como CUDA, ingenieros y desarrolladores pueden acelerar las aplicaciones informáticas mediante el aprovechamiento de la potencia de las GPU de una forma más o menos transparente, ya que aunque CUDA se basa en el lenguaje de programación C, dispone de API y librerías para una gran cantidad de lenguajes como por ejemplo python o java.

Logo de NVIDIA

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.

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.

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.

* http://mesos.apache.org

Bajo esta idea, Mesos ofrece una capa de abstracción entre los servidores y los recursos. Es decir, 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 maestro (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 maestro activo. Es decir, el nodo maestro también se encarga del seguimiento y control de los trabajos en ejecución.

5.2. YARN (Yet Another Resource Negotiator)

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.

** http://bit.ly/2btcahe

***Deamon* o servicio**

Un *daemon*, o demonio, (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.

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.

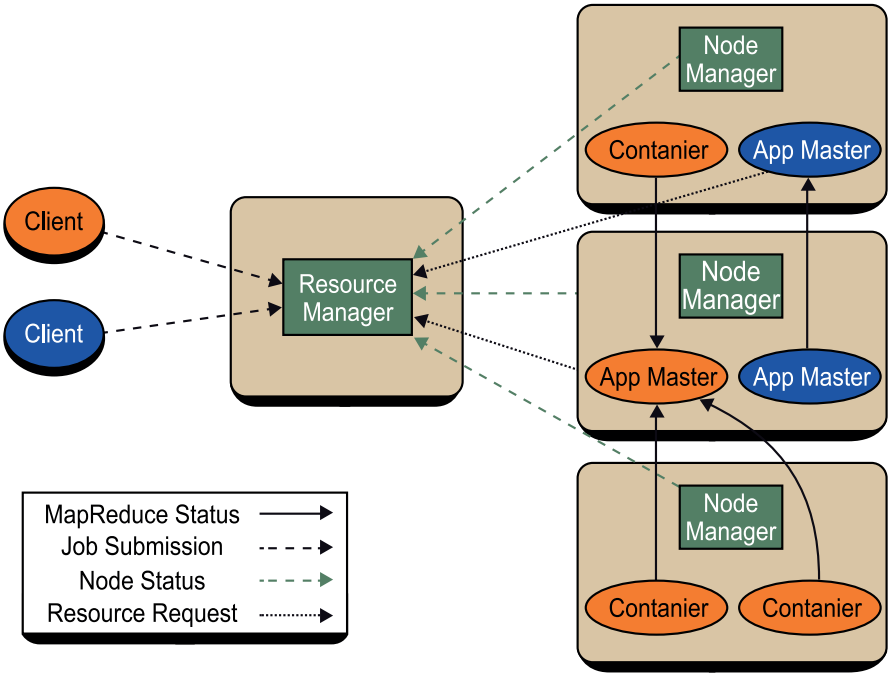
De esta separación surgen dos elementos:

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

- **ApplicationMaster (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 12 se ilustra la arquitectura de YARN para que se pueda entender de forma más clara.

Figura 12. Ejemplo de arquitectura en YARN



Fuente: https://hadoop.apache.org/docs/

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

6. Escenarios de procesamiento distribuido

Ahora describiremos los tres escenarios de procesamiento de datos distribuidos más comunes. El procesamiento de grandes volúmenes de información en lotes (*batch*), el procesamiento de datos en flujo (*stream*) y el procesamiento de datos usando tarjetas gráficas (*GPU*). Aunque todos estos 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.

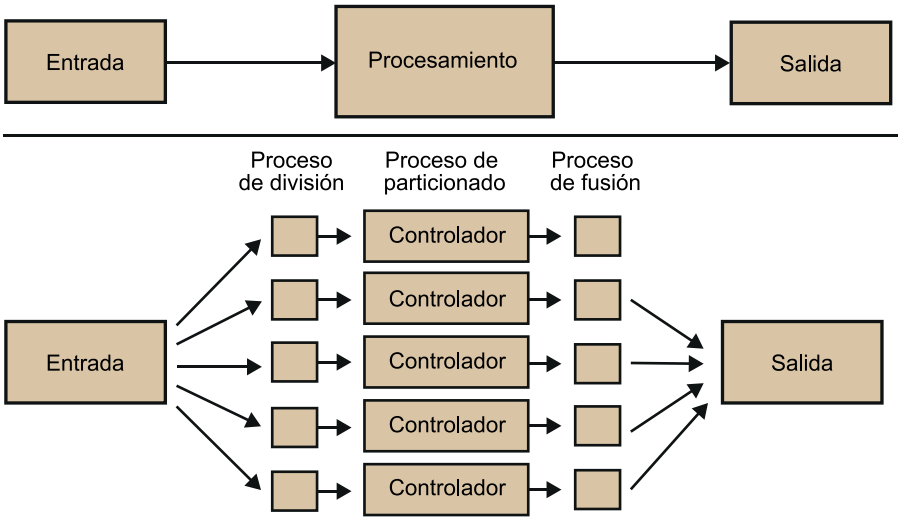
Fichero de logs

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.

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.

En la figura 13 observamos la estructura típica del procesamiento por lotes. En la parte superior de la figura observamos cómo se realizaría el procesado sin un sistema de cálculo distribuido, mientras que en la parte inferior podemos observar cómo se realizaría usando el paradigma de programación basado en MapReduce.

Figura 13. Ejemplo de aplicación de proceso en *batch*



6.2. Procesamiento en *stream*

El procesamiento en *stream* es un tipo de técnica de procesamiento y análisis de datos que se basa 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

Intentar determinar que clientes de los que están accediendo a una tienda online en un momento determinado tienen más probabilidad de comprar, una vez se ha determinado cuales 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 aun más su probabilidad de compra.

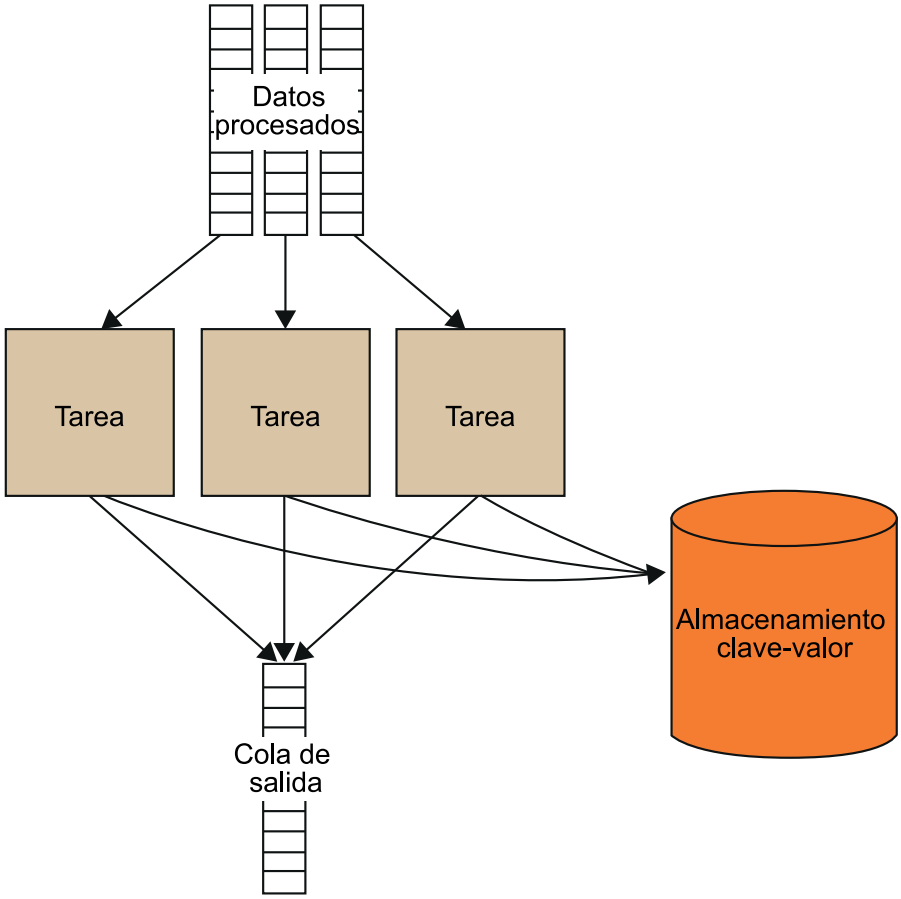
Generalmente, las únicas limitaciones de estos sistemas son las siguientes:

- 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 14 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 14. Ejemplo de aplicación de proceso en *stream*



6.3. Procesamiento en GPU

Un aspecto fundamental de las actuales GPU es el uso de pequeños procesadores (conocidos también como *Unified Shaders*). Estos son muy sencillos y ejecutan un conjunto de instrucciones muy concreto que realizan operaciones aritméticas básicas, pero su crecimiento en la integración (número de procesadores) en cada nueva generación de GPU es significativo. Actualmente en las tarjetas gráficas modernas nos encontrarnos con un número entre 1.000 y 4.000 procesadores.

El problema de esta integración es precisamente que no todas las tareas tienen que ser más eficientes en una GPU. Estas están especializadas en tareas altamente paralelizables cuyos algoritmos puedan subdividirse, procesarse por separado para luego unir los subresultados y tener el resultado final. Típicamente son problemas científicos, matemáticos o simulaciones, aunque un ejemplo mucho más popular es la problemática de minar Bitcoins.

Ejemplo

Los bitcoins se han ganado la confianza de mucha gente al ser dinero relativamente fácil de conseguir. Para un usuario final, la tarea que ha de realizar para obtener un bitcoin es tan sencilla como ejecutar un programa y esperar. Dependiendo de la capacidad del servidor tardará más o menos y podremos obtener beneficios dependiendo del consumo energético. Al fin y al cabo es un problema con base económica, donde los beneficios finales dependerán de los ingresos (los bitcoins ganados) menos los gastos (energéticos y de tiempo invertido).

Para obtener un bitcoin se ha de resolver un problema criptográfico que hace uso de un algoritmo de hashing, concretamente el SHA-256. Una persona que quiera generar un bitcoin tiene que ejecutar este algoritmo sobre múltiples cadenas alfanuméricas de forma repetida hasta que el resultado sobre una de ellas sea válido y entonces ganará una fracción de bitcoin.

A la vista de que minar bitcoins es una tarea altamente paralelizable, en la que se han de ejecutar un conjunto de operaciones matemáticas sencillas sobre cadenas alfanuméricas aleatorias, los sistemas con GPU son la mejor opción, ya que permiten repetir una misma operación matemática en paralelo de forma eficiente.

Secure Hash Algorithm

SHA versión 2 es un conjunto de funciones hash criptográficas diseñadas por la Agencia de Seguridad Nacional (NSA) y publicada en 2001 por el Instituto Nacional de Estándares y Tecnología (NIST) como un Estándar Federal de Procesamiento de la Información (FIPS) en USA. Una función hash es un algoritmo que transforma un conjunto arbitrario de elementos de datos, como puede ser una cadena alfanumérica, en un único valor de longitud fija (el *hash*). El valor hash calculado puede ser utilizado para la verificación de la integridad de copias de un dato original sin la necesidad de proveer el dato original. Esta irreversibilidad significa que un valor hash puede ser libremente distribuido o almacenado, ya que solo se utiliza para fines de comparación.

7. Stacks de software para sistemas de *big data*

En el mercado actual del *big data* es posible encontrar una gran cantidad de plataformas, software, librerías, proyectos *open source*, etc. que al combinarse permiten generar todo el *stack* necesario para la captura, almacenamiento y procesamiento de grandes volúmenes de datos.

Como el ecosistema de *big data* aún se encuentra en evolución, es difícil describir un *stack* de software completo y universal que funcione correctamente en todos los proyectos relacionados con el *big data*. En la figura 15 se representan las cuatro capas principales de un sistema *big data*, además se incluye algunos de los software y/o tecnologías más comunes que podemos encontrar para resolver los problemas de esa capa.

Figura 15. Descripción de las principales capas de un sistema de *big data*

Concretamente, observamos las siguientes capas (en inglés, *layers*):

- **Ingest Layer.** Es la capa encargada de capturar datos. Encontramos herramientas como Apache flume*, un sistema de gestión de *streams* que permite transformar datos en crudo antes de ser almacenados en el sistema.

  * https://flume.apache.org

- **Store Layer.** En la capa de almacenamiento encontramos todo tipo de sistemas para guardar y recuperar grandes volúmenes de datos de forma efi-

ciente. Aquí se incluyen todo tipo de bases de datos, tanto relacionales como NoSQL, sistemas de ficheros distribuidos, como HDFS, u otras tecnologías de almacenamiento en la nube, como Amazon S3.

- **Process/Analyze Layer.** Sin duda la capa de procesamiento es la más compleja y más heterogénea de todas. Es donde ubicamos tecnologías como Spark o Hadoop de las que tanto hemos hablado en este módulo.

- **Visualize Layer.** Finalmente, en la capa de visualización, herramientas como Tableau* permiten una visualización de datos interactiva de forma relativamente sencilla.

* https://www.tableau.com

Como ya hemos comentado, este *stack* es muy variable. Depende, en primer lugar, del proyecto concreto en el que trabajamos. En este sentido, el tipo de datos que empleamos, así como el tipo de análisis requerido serán determinantes para configurar un *stack* determinado con los elementos imprescindibles o más adecuados para poder implementar de forma satisfactoria el proyecto en cuestión.

Por otro lado, es importante destacar que el conjunto de herramientas disponibles puede variar de forma significativa en poco tiempo.

Por todo esto, no es posible definir un *stack* único y perdurable que pueda ser utilizado en cualquier proyecto a corto o medio plazo y será necesario definir un *stack* concreto en cada situación y en cada momento.

Resumen

Todos los expertos confirman 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 del *big data* o Data Science.

En este módulo hemos introducido los principales componentes de una arquitectura *big data*. Hemos empezado describiendo la estructura general de un sistema de *big data*, luego hemos hablado sobre 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 y científica. 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*), el procesamiento en flujo (*stream*) y el procesamiento en GPU, explicando sus diferencias y cuáles son sus principales características.

Glosario

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

call detail record 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**

chipset *m* Conjunto de circuitos integrados diseñados con base en la arquitectura de un procesador permitiendo que este funcionen en una placa base. Sirve de puente de comunicación del procesador con el resto de componentes de la placa, como son la memoria, las tarjetas de expansión, los puertos USB, ratón, teclado, etc.

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

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

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 contienen marcadores para separar sus diferentes elementos. *en* Semistructured data

deamon *f* 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.

DRAM *f* DRAM son las siglas de la voz inglesa Dynamic Random Access Memory, que significa memoria dinámica de acceso aleatorio (o RAM dinámica), para denominar a un tipo de tecnología de memoria RAM basada en condensadores.

fichero de logs *m* 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.

GPU *f* Una unidad de procesamiento gráfico o GPU (Graphics Processor Unit) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos, aplicaciones 3D interactivas o de cálculo científico.

HashMap *m* 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.

HDD *m* Acrónimo de Hard Disk Drive o disco magnético rotacional

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**

Lenguaje SQL *m* 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.

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

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

Open source El 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.

Page Rank *m* Familia de algoritmos utilizados para asignar de forma numérica la relevancia de los documentos (o páginas web) indexados por un motor de búsqueda.

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

proceso batch *m* Se conoce como sistema por lotes (*batch*) 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.

reglas ACID *f pl* 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 se puede consultar siguiente entrada de la Wikipedia: http://es.wikipedia.org/wiki/ACID.

RMI Remote Method Invocation, tecnología Java para ejecutar código arbitrario de forma remota en un ordenador.

SSD Solid State Disk, sistema de almacenamiento de datos permanente con acceso directo a los datos.

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

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

Bibliografía

Barlas, Gerassimos (2014). *Multicore and GPU Programming. An Integrated Approach*. Boston: Elsevier.

Biery, Roger (2017). *Introduction to GPUs for Data Analytics. Advances and Applications for Accelerated Computing*. San Francisco: O'Reilly Media.

Kumar, Manish; Singh, Chanchal (2017). *Building Data Streaming Applications with Apache Kafka*. Birmingham: Packt Publishing.

Sarkar, Aurobindo (2017). *Learning Spark SQL*. Birmingham: Packt Publishing.

White, Tom (2015). *Hadoop: The Definitive Guide, 4th Edition Storage and Analysis at Internet Scale*. Boston: O'Reilly Media.

Zaharia, Matei; Chambers, Bill (2017). *Spark: The Definitive Guide, Big Data Processing Made Simple*. Boston: O'Reilly Media.