TECNOLOGÍA

JORDI CASAS ROMA
JORDI NIN GUERRERO
FRANCESC JULBE LÓPEZ

BIG DATA

ANÁLISIS DE DATOS EN ENTORNOS MASIVOS

Prólogo de David Carrera

EDITORIAL UOC

13 h

Big data
Análisis de datos en entornos masivos

Jordi Casas Roma
Jordi Nin Guerrero
Francesc Julbe López

Director de la colección Manuales (Tecnología): Antoni Pérez

Diseño de la colección: Editorial UOC
Diseño de la cubierta: Natàlia Serrano

Primera edición en lengua castellana: abril 2019
Primera edición digital (pdf): mayo 2019

Rambla del Poblenou, 156,
08018 Barcelona
http://www.editorialuoc.com

Realización editorial: Reverté-Aguilar
ISBN: 978-84-9180-473-4

Autores

Jordi Casas Roma

Licenciado en Ingeniería Informática por la Universitat Autònoma de Barcelona (UAB), máster en Inteligencia artificial avanzada por la Universidad Nacional de Educación a Distancia (UNED) y doctor en Informática por la UAB. Desde 2009 ejerce como profesor en los estudios de Informática, Multimedia y Telecomunicación de la Universitat Oberta de Catalunya (UOC), y desde 2015 también como profesor asociado en la UAB. Es director del máster universitario en Ciencia de datos de la UOC. Sus actividades docentes se centran en el aprendizaje automático, la minería de datos y el análisis de datos en entornos *big data*. Desde 2010 pertenece al grupo de investigación KISON (K-ryptography and Information Security for Open Networks), donde investiga en temas relacionados con la privacidad de los datos, la minería de grafos (*graph mining*) y el aprendizaje automático (*machine learning*).

Jordi Nin Guerrero

Licenciado en Ingeniería Informática y doctor en Informática por la UAB. Al finalizar sus estudios de doctorado trabajó durante dos años como investigador posdoctoral Marie Curie en el LAAS-CNRS (Toulouse, Francia). Cuando volvió a Barcelona trabajó como profesor en el Departamento de Arquitectura de Computadores de la Universitat Politècnica de Catalunya (UPC). Desde 2015 ejerce como científico de datos sénior en BBVA, como profesor asociado a tiempo parcial en la Universitat de Barcelona (UB) y consultor en la UOC. Sus actividades docentes se centran en la minería de datos y el análisis en entornos *big data*. Sus intereses de investigación incluyen temas relacionados con el aprendizaje automático, la fusión de información y el análisis de redes financieras.

Francesc Julbe López

Licenciado en Ingeniería de Telecomunicaciones por la UPC. Tras unos años trabajando en la consultoría tecnológica, desarrolló su carrera profesional en el campo de la investigación espacial en proyectos de la Agencia Espacial Europea, principalmente para la misión GAIA, perfeccionando áreas de desarrollo de sistemas de procesado de

datos y *big data*, así como en la ESO (European Southern Observatory) en tareas de desarrollo de *software* para sistemas de acceso a los archivos astronómicos. Recientemente ha trabajado en una empresa del sector *fintech*, desarrollando una plataforma de *big data* para gestión de clientes e inversiones. Actualmente trabaja en Deloitte como responsable de desarrollo de proyectos de *data analytics* y *big data* para el sector público.

A mis padres, por haberme guiado, ayudado y apoyado en el camino que me ha traído hasta aquí.
Jordi Casas Roma

Índice

Prólogo

El ámbito de la computación ha sufrido importantes cambios durante la última década. El paradigma del *cloud computing* significó una auténtica revolución que ha transformado la manera en qué las organizaciones contemplan sus gastos en infraestructuras IT, dejando de ser solo una inversión de capital (CAPEX) para pasar a ser en muchos casos un servicio que formaba parte de sus costes operativos (OPEX). Posteriormente, una segunda revolución surgió de la mano de la creciente necesidad de muchas empresas y organizaciones de acumular datos generados de manera masiva, lo que comúnmente se conoce como *big data*. Actualmente nos encontramos ante una tercera revolución, la de la aplicación masiva de la inteligencia artificial, que mediante el uso de una combinación de nuevos e antiguos métodos de análisis de datos ha permitido desarrollar nuevas maneras de extraer información a partir de estos. Queda como ejercicio para el lector recordar como se planeaba un viaje en coche hace unos años, mediante el uso de mapas en papel, para compararlo con el uso generalizado hoy de aplicaciones basadas en plataformas *cloud*, que combinando tecnologías *big data* en tiempo real y modelos de predicción basados en la inteligencia artificial nos ayudan a decidir las rutas de manera dinámica, evitando atascos y optimizando el tiempo necesario para llegar al destino.

Cierto es que llamar *revoluciones* a los tres grupos de tecnologías anteriormente citadas puede parecer muy atrevido, especialmente cuando las tres se han producido en un periodo de tiempo tan corto. Pero la realidad es que en tecnología una revolución queda definida por un periodo de tiempo en el que un conjunto de tecnologías queda reemplazado por otras, e indudablemente las tecnologías desarrolladas en los ámbitos del *cloud computing*, el *big data* y la inteligencia artificial han reemplazado un conjunto de tecnologías anteriormente usadas de manera transversal en múltiples sectores relacionados con el ámbito IT.

Aunque el origen del concepto *big data* se inicia a principios del siglo XXI, en especial en ámbitos científicos, no es hasta finales de la primera década del mismo siglo que su uso y aceptación empiezan a crecer en el mundo de las empresas. Buena parte de la popularidad de las plataformas *big data* se inicia con unos artículos publicados por Google en 2003 y 2004 en qué presentaba como afrontaba este gigante tecnológico el paradigma de la computación para grandes volúmenes de datos. En ellos, introduce el modelo de programación MapReduce y su modelo de sistema de ficheros distribuidos Google File System. Posteriormente, es Yahoo! quien lidera un proyecto *open source* llamado Hadoop para implementar la primera plataforma del corriente *big data* tal como lo conocemos hoy en día.

Obviamente, muchas otras empresas se encontraron con retos tecnológicos parecidos a los que afrontó Google, y en especial aquellas grandes organizaciones que ya trabajaban con grandes volúmenes de datos. La particularidad de Hadoop es que posiblemente es la primera plataforma que fue aceptada de forma transversal por la industria, y significó el inicio

de la revolución tecnológica del *big data*. Posteriormente, muchas otras plataformas han aparecido, aunque probablemente Spark sea la que más consenso ha generado dentro de los usuarios de tecnologías *big data*. La adopción de este tipo de plataformas por parte de la industria también ha estado condicionada por las necesidades de cada sector, como por ejemplo la demanda de computación en tiempo real, tolerancia a diferentes niveles de latencia, posibilidad de usar computación aproximada o escalabilidad de los algoritmos a utilizar sobre los datos.

La adopción de la plataformas *big data* por parte de múltiples sectores industriales se inició en sus propios centros de procesado de datos (CPD), configurando manualmente las plataformas. En muchos casos, este proceso requería cambios fundamentales en la estructura de red y almacenamiento de estas infraestructuras, lo que planteaba una compleja decisión para muchas de estas empresas: abandonar las arquitecturas de sistemas habituales usadas para las soluciones de *data warehouse* clásica para dar un salto adelante que podía ser al vacío, o ignorar una corriente tecnológica que aunque podía acabar siendo tan solo una moda transitoria, tenía aspecto de consolidarse en los siguientes años. Una solución alternativa era recurrir a los recursos que ofrecían los proveedores de infraestructura *cloud* para desplegar las plataformas *big data* sin necesidad de hacer arriesgadas y costosas modificaciones a sus infraestructuras. Con este movimiento se combinaban las revoluciones del *big data* y del *cloud computing*.

Una vez el uso de plataformas *big data* en el *cloud* se hubo consolidado, se inició un segundo movimiento: los proveedores de soluciones *cloud* querían ascender por la cadena de valor que ofrecían a las empresas, pasando de ser meros proveedores

de infraestructura a ofrecer servicios de más alto nivel a sus clientes. De esta manera, un proceso de adopción de *big data* en *cloud* que se inició según el paradigma de infraestructura como servicio (o IaaS, por sus siglas en inglés), pasó a desarrollarse en la forma de plataforma como servicio (o PaaS, por sus siglas en inglés) y derivó finalmente en un modelo de *software* como servicio (o SaaS, por sus siglas en inglés), o incluso el modelo de todo como servicio (o XaaS, por sus siglas en inglés).

De esta manera, las empresas pasaron en muchos casos de gestionar su propia plataforma *big data* a cada vez más depender de uno o varios proveedores de soluciones *cloud* para procesar sus datos. Llegados a este punto, la tercera de las revoluciones se unió a las otras dos: los servicios *big data* ofrecidos por los proveedores de *cloud* ya incluyen de manera generalizada la capacidad de utilizar métodos avanzados de inteligencia artificial sobre los datos de una manera sencilla y eficiente, lo que abre numerosas posibilidades a las empresas que no disponen de recursos propios como para explorar todos los ámbitos tecnológicos que se requieren para ser competitivos en el mercado actual, altamente dependiente de la extracción de valor a partir de los datos.

David Carrera
Profesor agregado en UPC - BarcelonaTech
Investigador Asociado en el Centro Nacional de Supercomputación

Introducción

El enfoque del libro es claramente descriptivo, con el objetivo de que el lector entienda los conceptos e ideas básicos detrás de cada método o técnica, sin entrar en excesivos detalles técnicos, tanto matemáticos como algorítmicos. Siempre que es posible, se hace referencia a los trabajos originales que describieron por primera vez el problema a resolver, los cuales pueden ser rastreados para obtener la literatura más actualizada.

Estructura del libro

Este libro se divide en cinco partes principales:

- El primer bloque constituye la introducción al análisis de datos en entornos *big data*, presentando la problemática que originó la aparición del *big data*, así como algunas de las definiciones más ampliamente utilizadas y una clasificación básica de tipos de escenarios de procesamiento de datos masivos. A continuación se presentan algunos conceptos básicos de algorítmica para entender los tipos de algoritmos existentes y su eficiencia en entornos de procesamiento y análisis de datos. Esta parte finaliza con una breve descripción de las tareas de apren-

dizaje automático, que se suelen aplicar en todo tipo de problemas relacionados con el procesamiento de datos masivos.

- En la segunda parte, se describen los conceptos tecnológicos más importantes para entender la arquitectura de un sistema *big data*: los tipos de datos existentes, el funcionamiento básico de *cluster* de computadores y el procesamiento en GPU. A continuación se describen los elementos básicos de una arquitectura *big data*, como son el sistema de archivos, el sistema de cálculo distribuido y el gestor de recursos. Finalmente, se presentan brevemente los diferentes escenarios típicos de procesamiento de datos masivos, que serán ampliados en las partes sucesorias.

- La tercera parte presenta la problemática relacionada con el procesamiento por lotes (*batch*). Se describen los procesos de captura y preprocesemiento típicos empleados, así como los sistemas de almacenamiento habitualmente utilizados en este tipo de procesamiento: sistemas de ficheros distribuidos y bases de datos NoSQL. Finalmente, se introducen las principales herramientas empleadas en el análisis de este tipo de datos, prestando especial atención a los entornos Apache Hadoop y Apache Spark.

- El cuarto bloque de este libro se centra en los métodos de procesamiento de datos en flujo (o datos en *streaming*). Al igual que en la parte anterior, nos centraremos en los procesos de captura y preprocesamiento, el almacenamiento y el análisis de datos dinámicos.

- Finalmente, la quinta parte del libro está dedicada al procesamiento de datos en formato de grafos. En el primer capítulo se presentan las peculiaridades de este tipo de datos, así como sus principales métodos de representación. A continuación se describen los principales modelos de almacenamiento, prestando especial atención a las bases de datos NoSQL en grafo. Y para finalizar, se enumeran las principales técnicas de procesamiento de grafos, así como las problemáticas relacionadas con este tipo de datos y algunas de las principales herramientas que existen en la actualidad para lidiar con grafos.

En los distintos capítulos de este libro se describen algunos ejemplos sencillos con el objetivo de que puedan ser interpretados de forma fácil y ayuden en la comprensión de los detalles teóricos de los métodos descritos. En el libro, sin embargo, no se incluye el código completo asociado a estos ejemplos, dado que este puede sufrir cambios debido a las actualizaciones de librerías y versiones de los lenguajes de programación empleados. El objetivo de estos ejemplos es, simplemente, ilustrar algunos conceptos claves, pero en ningún caso se pretende mostrar ejemplos completos y funcionales, que serían más propios de una guía de referencia.

Notación empleada

Marcamos el texto resaltado, generalmente en definiciones o partes especialmente importantes, como:

Definición o texto especialmente importante o relevante.

Marcamos los ejemplos cortos incrustados en el texto, como:

Texto de un ejemplo corto incrustado en el texto principal.

Parte I

Introducción

Capítulo 1

Introducción al *big data*

Iniciaremos este capítulo con una introducción al concepto de *big data*, centrándonos en el cambio de paradigma que supone la llegada de los datos masivos. A continuación, veremos una de las primeras definiciones de *big data*, relacionada con las magnitudes del dato. A partir de esta primera definición han surgido muchas más, que en general, amplían la definición inicial. A partir de esta definición inicial, presentaremos la definición utilizada en este texto y veremos algunos estándares importantes en relación a los datos y la interconexión de sistemas. Finalmente, presentaremos un pequeño ejemplo que sirva para ilustrar un escenario de uso de tecnologías *big data*, en este caso concreto, en una *smart city*.

1.1. Antecedentes y contextualización

El término *big data*, terminología anglosajona ámpliamente utilizada que se suele traducir por *datos masivos*, apareció a principios del siglo XXI en el entorno de la ciencias, en particular, de la astronomía y de la genética, debido a que ambos

campos experimentaron una gran explosión en la disponibilidad de datos. Por ejemplo, en el campo de la astronomía el proyecto de exploración digital del espacio llamado Sloan Digital Sky Survey[1] generó más volumen de datos en sus primeros meses que el total de los datos acumulados en la historia de la astronomía hasta ese momento. En el campo de la genética, un ejemplo relevante sería el proyecto del genoma humano. Este proyecto tenía como objetivo encontrar, secuenciar y elaborar mapas genéticos y físicos de gran resolución del ADN humano, que genera una cantidad de datos del orden de 100 gigabytes por persona.

En estos últimos años la explosión de datos se ha generalizado en muchos de los campos que rodean nuestra vida cotidiana. Entre otros, el incremento del número de dispositivos con conexión a internet, el auge de las redes sociales y el internet de la cosas (IoT)[2] han provocado una explosión en el volumen de datos disponibles. Además de la gran cantidad de datos, es importante destacar que muchos de ellos son abiertos y accesibles, lo que permite que puedan ser explotados por usuarios o instituciones de cualquier parte del mundo. Pero el mero hecho de disponer de una gran cantidad de datos no aporta valor. El verdadero valor de los datos está en su análisis e interpretación.

La aparición de nuevas técnicas y tecnologías de procesamiento de datos surgió a causa de la imposibilidad de procesar la enorme cantidad de datos que se generaban con las

[1]El proyecto Sloan Digital Sky Survey (http://www.sdss.org) tiene como objetivo identificar y documentar los objetos observados en el espacio.

[2]Internet de las cosas o IoT (*internet of things*, en inglés) es un concepto que se refiere a la interconexión digital de objetos cotidianos con internet.

técnicas tradicionales. Aunque la mejoras y el abaratamiento en el *hardware* de los ordenadores permite cargar y procesar más datos con las técnicas tradicionales, el aumento en la cantidad de datos es de varios órdenes de magnitud superiores. Por lo tanto, aunque podamos adquirir más y mejor *hardware*, es absolutamente insuficiente para afrontar el aumento masivo de datos. Por ejemplo, imaginemos en cómo debería ser el ordenador de Google para indexar todos los contenidos de la web. Por consiguiente, también fue necesaria la evolución de la tecnología basada en el *software*.

Las grandes empresas de internet, como Google, Amazon y Yahoo!, se encontraron con varios problemas importantes para continuar desempeñando sus tareas cotidianas. En primer lugar, la gran cantidad de datos que estaban acumulando hacía inviable su procesamiento en un único ordenador. Por tanto, se debía usar procesamiento distribuido para involucrar distintos ordenadores que trabajasen con los datos de manera paralela, y así poder procesar más en menos tiempo. En segundo lugar, la heterogeneidad de los datos requirió nuevos modelos para facilitar la inserción, la consulta y el procesamiento de datos de cualquier tipo y estructura. Y en tercer lugar, los datos debían de procesarse de forma rápida, aunque hubiera muchos para procesar. Por ejemplo, un buscador web no sería útil si devolviera los resultados de nuestra búsqueda después de un tiempo demasiado largo, por ejemplo, más de una hora. En consecuencia, estas grandes empresas que gestionaban grandes volúmenes de datos se dieron cuenta de que las técnicas de procesamiento de datos tradicionales no permitían tratar todos los datos que utilizaban de manera eficiente, y tuvieron que crear sus propias tecnologías para

poder continuar con el modelo de negocio que ellos mismos habían creado.

Muchas de las técnicas que se desarrollaron para dar respuesta a los problemas planteados por los datos masivos utilizan un planteamiento basado en el procesamiento paralelo. Este paradigma se basa en dos pasos principales:

1. En primer lugar, se divide el problema en subproblemas de menor tamaño y complejidad. De esta forma, se pueden distribuir los distintos subproblemas a distintas computadoras de forma que cada una se encargue de uno de ellos de forma independiente.

2. En segundo lugar, la solución final del problema se compone a partir de las soluciones parciales de los subproblemas. Una vez cada subproblema es resuelto de forma independiente, se ensamblan todas las pequeñas soluciones resultantes para crear la solución global del problema inicial.

1.2. El nuevo paradigma de *big data*

Hasta ahora, si un agente o institución deseaba evaluar un fenómeno no podía, generalmente, recoger todos los datos relacionados con él. El motivo era que los los métodos de recogida y procesamiento de datos eran muy costosos en tiempo y en dinero. En estos casos, se escogía una pequeña muestra aleatoria del fenómeno, se definía un conjunto de hipótesis que comprobar y se estimaba con una cierta probabilidad que, para la muestra elegida, dichas hipótesis eran válidas. Este es el paradigma de la **causalidad**, donde se intenta establecer una

relación de causa-efecto entre el fenómeno que se analiza y los datos relacionados con él (Mayer-Schönberger, 2013).

Hoy en día, en cambio, la recogida de datos masivos ha permitido obtener información sobre la muestra completa (o casi) de datos relacionada con el fenómeno que hay que evaluar, es decir, toda la población. Por ejemplo, si una institución desea analizar los *tweets* que tratan sobre un tema de interés público, es perfectamente factible que pueda recoger todos aquellos que hablen del tema y analizarlos. En este caso, el análisis no pretende confirmar o invalidar una hipótesis, sino establecer **correlaciones** entre distintas variables de la muestra. Por ejemplo, supongamos que existe una fuerte correlación entre el lugar de residencia de los vecinos de una ciudad y su opinión ante una determinada problemática de esta. En este caso, podemos explotar la relación que existe entre ambas variables aunque no sepamos la causa que induce de la una a la otra.

Los datos masivos imponen un nuevo paradigma donde la correlación «sustituye» a la causalidad. Determinar la causalidad de un fenómeno pierde importancia, y en contraposición, «descubrir» las correlaciones entre las variables se convierte en uno de los objetivos principales del análisis.

Este cambio de paradigma provoca que los sistemas de *big data* se centren en encontrar «qué» aspectos están relacionados entre sí, y no en «por qué» están relacionados. Estos sistemas pretenden responder cuestiones del tipo: ¿qué pasó?, ¿qué está pasando? y ¿qué pasaría si?, pero desde un punto de vista basado en las correlaciones, donde no se busca la explicación del fenómeno, sino solo el descubrimiento del fenómeno en sí. En consecuencia, la causalidad pierde terreno a favor de asociación entre hechos.

1.2.1. Primera definición de *big data*

En el 2001, el analista Doug Laney (2001) de META Group (ahora Gartner) utilizaba y definía el término *big data* como el conjunto de técnicas y tecnologías para el tratamiento de datos, en entornos de gran volumen, variedad de orígenes y en los que la velocidad de respuesta es crítica.

Esta definición se conoce como las 3 V del *big data*: volumen, velocidad y variedad. Hoy en día está comúnmente aceptado que la definición de las 3 V haya sido ampliada con una cuarta V, la veracidad.

La figura 1 muestra la interacción de las 4 V de *big data* según IBM: existen grandes volúmenes de datos (*volume*), procedentes de una gran variedad de fuentes (*variety*), de un cierto grado de incertidumbre (*veracity*) y que puede ser necesario procesar para obtener respuestas rápidas (*velocity*).

A continuación veremos en más detalle las 4 V de la definición de *big data*.

Volumen

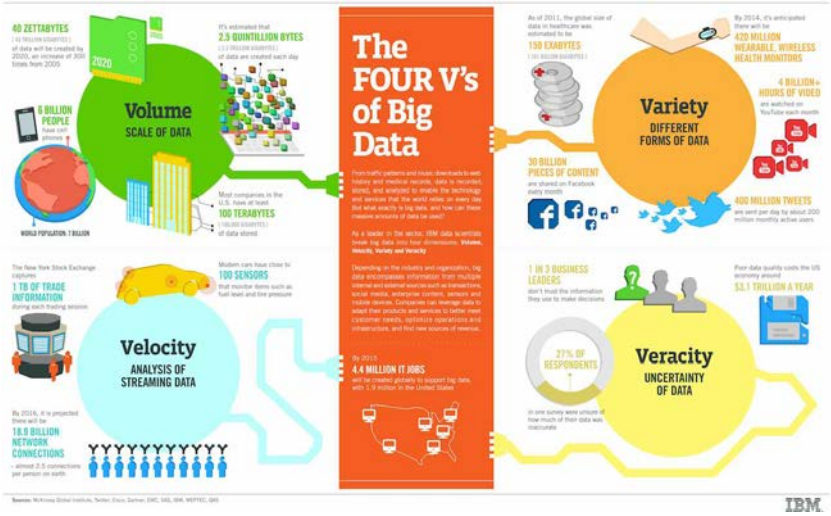
Se estima que el volumen de datos existente en la actualidad está por encima del zettabyte[3] y que crecerá de forma exponencial en el futuro.

Los almacenes de datos tradicionales, basados en bases de datos relacionales,[4] tienen unos requisitos de almacenamiento muy controlados y suelen estar acotados en máximos de crecimiento de unos pocos gigabytes diarios. Si multiplicamos el

[3]Zettabyte (ZB) = 10^{21} bytes = 1.000.000.000.000.000.000.000 bytes.

[4]Las bases de datos relacionales almacenan los datos en tablas y permiten las interconexiones o relaciones entre los datos de distintas tablas. El lenguaje utilizado para consultas y mantenimiento se llama SQL (Structured Query Language).

Figura 1. Interacción de las 4 V de *big data* según IBM



Fuente: http://www.ibmbigdatahub.com/infographic/four-vs-big-data

volumen de información y sobrepasamos este límite de confort, el rendimiento del sistema podría verse gravemente afectado y, por tanto, habría que replantearse reestructurar el sistema de almacenamiento considerando un entorno de *big data*.

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.

Velocidad

En un entorno dinámico, el tiempo que se tarda en obtener la información o el conocimiento frente a determinados sucesos es un factor tan crítico como la información en sí misma.

En algunos casos, la información extraída de los datos es útil mientas los datos son «frescos», pero pierde valor cuando los datos dejan de reflejar la realidad. La gestión del tráfico es un ejemplo que requiere decisiones que deben tomarse en un espacio de tiempo breve; esto es, prácticamente en tiempo real. Por tanto, en algunos casos el *big data* debe tratar de proporcionar la información necesaria en el menor tiempo posible. Aunque se trata de un objetivo y una característica deseable en *big data*, técnicamente no siempre es posible trabajar en tiempo real.

Existen dos tipos de velocidades que juntas condicionarán la velocidad final de respuesta ante nuevos datos. Estas son:

- **Velocidad de carga**. Los nuevos datos deben ser preparados, interpretados e integrados con el resto de datos antes de poder ser procesados. Estos procesos incluyen los procesos de extracción, transformación y carga (ETL),[5] que son costosos en tiempo y en recursos *hardware* y *software*.

- **Velocidad de procesamiento**. Una vez los nuevos datos están integrados y listos para su análisis, debemos considerar otros tipos de procesamiento, como la aplicación de funciones estadísticas avanzadas o técnicas de inteligencia artificial. Este procesamiento suele implicar consultas para la extracción del conjunto de los datos de interés, almacenamiento intermedio de dichos datos o aplicación de los cálculos sobre el conjunto de los datos extraídos, por ejemplo.

[5]Extraer, transformar y cargar (ETL en inglés *extract, transform and load*) es el proceso que permite mover datos desde múltiples fuentes, reformatearlos, limpiarlos, normalizarlos, y almacenarlos en el sistema necesario para su análisis.

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.

Variedad

La estructura de datos se define como la forma en que se encuentran organizados un conjunto de datos. La variedad se refiere a los diferentes formatos y estructuras en que se representan los datos. Podemos clasificar los orígenes de datos según su nivel de estructuración en:

- **Orígenes de datos estructurados**. La información viene representada por un conjunto o agrupación de datos atómicos elementales, es decir, datos simples que no están compuestos de otras estructuras. Se conoce de antemano la organización de los datos, la estructura y el tipo de cada dato elemental, su posición y las posibles relaciones entre los datos. Los datos estructurados son de fácil interpretación y manipulación.

  Los ficheros con una estructura fija en forma de tabla, como los ficheros CSV o las hojas de cálculo, son claros ejemplos de orígenes de datos estructurados.

- **Orígenes de datos semiestructurados**. La información viene representada por un conjunto de datos elementales, pero a diferencia de los datos estructurados

no tienen una estructura fija, aunque tienen algún tipo de estructura implícita o autodefinida.

Ejemplos de este tipo de datos son, por ejemplo, los documentos XML o las páginas web. En ambos casos los documentos siguen ciertas pautas comunes, pero sin llegar a un nivel de estructuración fija.

- **Orígenes de datos no estructurados**. La información no aparece representada por datos elementales, sino por una composición cohesionada de unidades estructurales de nivel superior. La interpretación y manipulación de estos orígenes de datos resulta mucho más compleja que el de los estructurados o semiestructurados.

  Ejemplos de orígenes de datos no estructurados son textos, audios, imágenes o vídeos.

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.

Veracidad

La gran cantidad de datos y sus orígenes en *big data* provoca que la veracidad del dato deba ser especialmente considerada y se deba aceptar cierto grado de incertidumbre. Este grado tolerado de incertidumbre puede tener origen en la exactitud

del dato y en la fiabilidad de su procesamiento (exactitud del cálculo).

- **Exactitud del dato**. Muchos de los datos analizados mediante *big data* son intrínsecamente dudosos, relativos o con un cierto grado de error inherente. Por ejemplo, los datos procedentes de redes de sensores utilizados para medir la temperatura ambiental pueden incluir cierto grado de incertidumbre, dado que generalmente unas pocas mediciones se hacen extensibles a zonas y periodos más grandes.

- **Exactitud del cálculo**. Una parte muy importante de los cálculos en *big data* están basados en métodos analíticos que permiten cierto grado de incertidumbre. La minería de datos, el procesamiento del lenguaje natural, la inteligencia artificial o la propia estadística permiten calcular el grado de fiabilidad. Se trata de indicadores de la fiabilidad o exactitud de la predicción, que puede ser inferior a 100 % aunque los datos originales se consideren veraces. Por ejemplo, aunque los comentarios de los usuarios de Facebook sobre una empresa son veraces, el resultado de su análisis mediante técnicas automáticas de procesamiento del lenguaje natural puede tener una fiabilidad por debajo del 100 %.

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.

1.2.2. Nuestra definición de *big data*

A partir de la definición anterior, han aparecido otras definiciones alternativas que iban añadiendo, progresivamente, más V a las definiciones anteriores. Conceptos como por ejemplo la variabilidad, la validez o la volatilidad se han incorporado en esta definición de *big data* según la propuesta de algunos autores.

El hecho de que existan múltiples definiciones complica la comprensión e identificación de escenarios. La gran mayoría de ellas incluyen lo que se conoce como las 3 V del *big data* que hemos comentado anteriormente y que son magnitudes físicas del dato.

Por tanto, en aras de tener un enfoque pragmático, en este texto vamos a usar la siguiente definición de *big data*:

Entendemos por ***big data*** el conjunto de estrategias, tecnologías y sistemas para el almacenamiento, procesamiento, análisis y visualización de conjuntos de datos complejos.

Y entenderemos por conjuntos de datos complejos aquellos que dado su volumen, velocidad o variedad no pueden ser tratados de forma eficiente en un sistema tradicional de análisis de datos.

1.2.3. Clasificación de NIST

De acuerdo con el NIST,[6] y en particular dentro de su grupo de trabajo de *big data*, existen tres tipologías de escenarios que requieren el uso de *big data*.

Los tipos disponibles se resumen a continuación:

- Tipo 1: donde una estructura de datos no relacional es necesaria para el análisis de datos.
- Tipo 2: donde es necesario aplicar estrategias de escalabilidad horizontal para procesar y analizar de manera eficiente los datos.
- Tipo 3: donde es necesario procesar una estructura de datos no relacional mediante estrategias de escalabilidad horizontal para procesar y analizar de manera eficiente los datos.

Por tanto, para una determinada necesidad analítica, es posible identificar si estamos en un escenario de *big data* o no, y si es necesario este tipo de tecnologías, hecho que cada vez más se erige como un punto relevante y de partida para la implementación de este tipo de proyectos.

1.2.4. Estándares en *big data*

A medida que *big data* ha adquirido mayor importancia para las organizaciones y estas se han empezado a preocupar por cómo llevar a cabo un proyecto de este tipo, ha quedado patente que se necesita interconectar múltiples sistemas y

[6]NIST (https://www.nist.gov/) es el acrónimo de National Institute of Standards and Technology, institución americana que estudia, define y promueve estándares tecnológicos.

tecnologías. Esta integración e interoperabilidad de sistemas requiere estándares de mercado.

Por ejemplo, dentro del contexto de la inteligencia de negocio y la analítica ya existen estándares como UIMA[7] (Unstructured Information Management Architecture), OWL[8] (Web Ontology Language), PMML[9] (Predictive Model Markup Language), RIF[10] (Rule Interchange Format) y XBRL[11] (eXtensible Business Reporting Language), que permiten la interoperabilidad de analítica de datos en información no estructurada, ontologías de modelos de datos, modelos predictivos, el intercambio de datos entre organizaciones y reglas e informes financieros respectivamente.

Desde 2012, varios grupos de trabajo de la comunidad internacional han empezado a trabajar en la creación de estándares, por ejemplo NIST, TMForum, Cloud Security Alliance, ITU, Open Data Platform initiative (ODPi) y Common Criteria Portal.

En el caso de ODPi, su búsqueda de estándares se fundamenta en proponer una configuración mínima de Apache Hadoop que según su criterio incluye solo cuatro componentes: HDFS, YARN, MapReduce y Ambari.

La gran mayoría de estos grupos soporta la adopción efectiva de tecnología *big data* a través del consenso en definiciones, taxonomías, arquitecturas de referencia, casos de uso y *roadmap* tecnológicos.

[7]https://uima.apache.org/

[8]https://en.wikipedia.org/wiki/Web_Ontology_Language

[9]http://dmg.org/

[10]https://en.wikipedia.org/wiki/Rule_Interchange_Format

[11]https://www.xbrl.org/

1.3. Utilidad: ¿donde 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, 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.

- **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*[12] 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 más de antelación respecto a los sistemas tradicionales.

[12]https://bit.ly/2Ml16mU

- **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*.[13] 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.

- **Telefonía**. Las compañías de telefonía utilizan la información generada por los teléfonos móviles (posición GPS y los CDR)[14] para estudios demográficos, planificación urbana, etc.

- **Finanzas**. Los grandes bancos disponen de sistemas de *trading* algorítmico[15] que analizan una gran cantidad de datos de todo tipo para decidir que operaciones en bolsa son las más rentables en cada momento.

1.4. Datos, información y conocimiento

La última de las 4 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

[13] «How Obama Used big data to Rally Voters», https://bit.ly/2yTn9K1

[14] *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. Contiene varios atributos como la hora, duración, estado de finalización, número de la fuente o número de destino.

[15] El *trading* algorítmico es una modalidad de operación en mercados financieros 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.

conocimiento», y 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 irrelevantes 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 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.

1.5. Ejemplo de escenario *big data*

A continuación se muestra el contexto de una *smart city* como una situación en la que las técnicas y tecnologías de *big*

data pueden ser necesarias para un correcto procesamiento y análisis de los datos.

Supongamos que la ciudad en cuestión recoge los siguientes datos:

- Las cámaras de tráfico recogen imágenes de forma continua, ya sea en formato de vídeo, pero también identificando los vehículos que circulan en cada vía a través de su matrícula.
- Los sensores de las zonas de aparcamiento exteriores proporcionan información continua sobre su ocupación, indicando en cada momento si cada una de las plazas de la ciudad está vacía o ocupada.
- Una gran cantidad de sensores repartidos por toda la ciudad analizan la calidad del aire en periodos cortos de tiempo, produciendo un análisis continuo en las distintas zonas de la ciudad. En cada análisis se incluyen muchos factores, relacionados con la contaminación y los agentes tóxicos del aire.
- Los puntos de transporte urbano basado en el uso compartido de bicicletas informa en cada momento sobre la disponibilidad de bicicletas en cada punto de recogida y devolución de la ciudad.

Pero además, el ayuntamiento ha decidido complementar la información que recoge mediante los distintos sensores de la ciudad con información obtenida a través de internet y de las redes sociales. Entre otros, el ayuntamiento se plantea:

- Monitorizar las acciones de los usuarios que visitan alguna de las páginas web municipales, registrando información como por ejemplo las páginas accedidas, el

dispositivo con el que se accede o la ubicación cuando está disponible. Adicionalmente, también se quiere analizar y registrar los comentarios de los usuarios en los foros municipales, donde se discute cualquier tema de interés para la ciudad, y de las encuestas en línea que el ayuntamiento realiza periódicamente.

- Recopilar información de la red social Twitter referente al estado del tráfico en cualquier momento y punto de la ciudad. Para ello, obtienen y almacenan todos los *tweets* con información referente a alguna de las principales calles, vías o paseos de la ciudad.

- Almacenar la información de las interacciones de los usuarios en la página municipal de Facebook, como por ejemplo el número de «me gusta» o los comentarios de los usuarios.

Este escenario presenta los siguientes problemas relacionados con las 4 V y que lo hace un buen candidato para aplicar técnicas de *big data*:

1. **Volumen**. El volumen de datos generado diariamente en una gran ciudad puede superar los límites físicos de las bases de datos y herramientas de análisis tradicionales.

2. **Velocidad**. Los análisis de datos relacionados con el tránsito deben tener respuestas rápidas que permitan detectar y corregir problemas, en la medida de lo posible, de forma casi inmediata. Una reacción tardía a los problemas de tráfico es sinónimo de no reaccionar.

3. **Variedad**. Existen distintos orígenes de datos, algunos de ellos no estructurados o semiestructurados, como por ejemplo, los comentarios en Facebook o los foros municipales, donde encontramos algunos campos de texto libre para recoger opiniones e impresiones.

4. **Veracidad**. Existen datos provenientes de redes sociales, de encuestas en línea y de foros que pueden contener faltas de ortografía, abreviaturas e interpretaciones ambiguas. El hecho de tratar con estos datos provoca que el grado de incertidumbre sea elevado.

Las 4 V son los síntomas que indican la conveniencia de utilizar un sistema de *big data* para realizar un determinado análisis. El análisis de *big data* difiere ligeramente de los análisis tradicionales, debido a que se analizan todos los datos de las distintas fuentes de datos de manera integrada. Al contar con los datos combinados de raíz, se minimiza la pérdida de información y se incrementan las posibilidades de encontrar nuevas correlaciones no previstas.

Capítulo 2

Algoritmos, paralelización y *big data*

En este capítulo discutiremos la problemática relacionada con el procesamiento de grandes volúmenes de datos desde un punto de vista algorítmico. En este sentido, veremos los distintos tipos de algoritmos, como medir su complexidad y cuáles de ellos son más eficientes para el procesamiento en paralelo, que es la base del procesamiento de datos masivos.

2.1. La problemática del procesado secuencial y el volumen de datos

Las aproximaciones habituales a la computación y a la realización de cálculos complejos en modo secuencial (no paralelizables) o lo que en lenguaje de computación se traduce en el uso de un solo procesador, está convirtiéndose en un mecanismo arcaico para la resolución de problemas complejos por varias razones:

- El volumen de datos a analizar crece a un ritmo muy elevado en la que llamamos la «era del *big data*». Así, los problemas que tradicionalmente podían solucionarse con un código secuencial bien implementado y datos almacenados localmente en un equipo, ahora son computacionalmente muy costosos e ineficientes, lo que se traduce en problemas irresolubles.

- Actualmente los equipos poseen procesadores multinúcleo[1] capaces de realizar cálculos en paralelo. Deberíamos ser capaces de aprovechar su potencial mejorando las herramientas de procesado para que sean capaces de utilizar toda la capacidad de cálculo disponible.

- Además, la capacidad de cálculo puede ser incrementada más allá de la que proveen los equipos individuales. Las nuevas tecnologías permiten combinar redes de computadores, lo que se conoce como escalabilidad horizontal.[2] La escalabilidad horizontal incrementa la capacidad de cálculo de forma exponencial así como la capacidad de almacenaje, aumentado varios órdenes de magnitud la capacidad de cálculo del sistema para resolver problemas que con una aproximación secuencial serian absolutamente irresolubles, tanto por consumo de recursos disponibles como por consumo de tiempo.

Sin embargo, aun teniendo en cuenta que los recursos disponibles para la resolución de problemas complejos son mucho

[1]Son aquellos procesadores que combinan uno o más microprocesadores en una sola unidad integrada o paquete.

[2]Un sistema escala horizontalmente si al agregar más nodos al mismo, el rendimiento de este mejora.

mejores y eficientes, este factor no soluciona por si solo el problema. Hay una gran cantidad de factores a tener en cuenta para que un algoritmo pueda funcionar de forma correcta una vez se esté ejecutando en un entorno paralelo.

Aprovechar la mejora en los recursos de computación disponibles para ejecutar algoritmos de forma paralela requiere de buenas implementaciones de dichos algoritmos. Hay que tener en cuenta que si un programa diseñado para ser ejecutado en un entorno monoprocesador no se ejecuta de forma rápida en dicho entorno, será todavía más lento en un entorno multiprocesador. Y es que una aplicación que no ha sido diseñada para ser ejecutada de forma paralela no podrá paralelizar su ejecución aunque el entorno se lo permita (por ejemplo, en un entorno con múltiples procesadores).

En computación, un *cluster* es un conjunto de ordenadores conectados entre si formando una red de computación distribuida.

Así, una infraestructura que va a ejecutar un algoritmo de forma paralela y distribuida debe tener en cuenta el número de procesadores que se están utilizando, así como el uso que hacen dichos procesadores de la memoria disponible, la velocidad de la red de ordenadores que forman el *cluster*, etc. Desde un punto de vista de implementación, un algoritmo puede mostrar una dependencia regular o irregular entre sus variables, recursividad, sincronismo, etc. En cualquier caso es posible acelerar la ejecución del algoritmo siempre que algunas subtareas puedan ejecutarse de forma simultánea dentro del *workflow* completo de procesos que componen el algoritmo. Y es que cuando pensamos en el desarrollo de un algoritmo

tendemos a entender la secuencia de tareas en un modo secuencial, dado que en la mayoría de ocasiones un algoritmo es una secuencia de tareas, las cuales dependen de forma secuencial unas de otras, lo que queda traducido en un código no paralelo cuando dicha aproximación se traslada a la implementación en *software*.

En este capítulo vamos a explorar los diferentes tipos de algoritmos, algunos de ellos muy utilizados en el mundo del *big data*, así como entender la naturaleza del paralelismo para cada uno de ellos.

2.2. ¿Qué es un algoritmo?

Aunque pueda parecer una pregunta obvia, conviene concretar qué es un algoritmo para poder entender cuáles son sus diferentes modalidades y qué soluciones existen para poder acelerar su ejecución en entornos *big data*, esto es, para poder ejecutarlos de forma paralela y distribuida.

Un algoritmo es la secuencia de procesos que deben realizarse para resolver un problema con un número finito de pasos. Algunos de estos procesos pueden tener relaciones entre ellos y otros pueden ser totalmente independientes. La comprensión de este hecho es crucial para entender cómo podemos paralelizar nuestro algoritmo para, posteriormente implementarlo y ejecutarlo sobre un gran volumen de datos.

Así, los componentes de un algoritmo son:

1. Tareas.

2. Dependencias entre las tareas, donde las salidas (*outputs*) de una tarea son las entradas (*inputs*) de la siguiente.

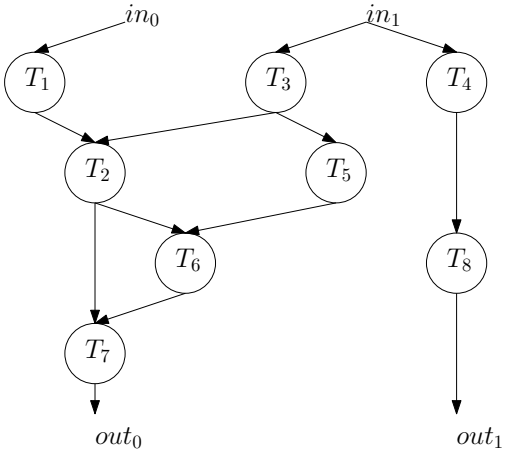
3. Entradas necesarias para el algoritmo.

4. Resultado o salidas que generará el algoritmo.

Estas dependencias pueden describirse mediante un grafo dirigido (*direct graph*, DG) donde el término «dirigido» indica que hay un orden en la secuencia de dependencias, y por lo tanto, que es necesaria la salida de cierta tarea para iniciar otras tareas dependientes.

Un DG es un grafo en el cual los nodos representan tareas y los arcos (o aristas) son las dependencias entre tareas, tal como se han definido anteriormente. La figura 1 muestra un ejemplo de algoritmo representado por su DG.

Figura 1. Representación DG de un algoritmo con dos entradas in_0 e in_1, dos salidas out_0 y out_1, diversas tareas T_i y sus dependencias



Fuente: elaboración propia

Un algoritmo también puede representarse mediante la llamada **matriz de adyacencia**, muy utilizada en teoría de grafos.

La matriz de adyacencia $M_{N\times N}$ es una matriz de $N \times N$ elementos, donde N es el número de tareas. Tal y como se muestra a continuación, el valor 1 indica si existe una dependencia entre las tareas correspondientes (podrían ser valores distintos de 1, lo que otorgaría pesos diferentes a dichas dependencias).

$$M_{N\times N} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

En esta matriz, podemos identificar las entradas fácilmente, ya que sus columnas contienen todos los valores iguales a 0. De forma similar, las salidas se pueden identificar a partir de las filas asociadas, que presentan todos los valores iguales a 0. En resto de nodos intermedios presentan valores distintos de 0 en sus filas y columnas.

A partir del tipo de dependencia entre los procesos, podemos clasificar los algoritmos en secuenciales, paralelos o serie-paralelos.

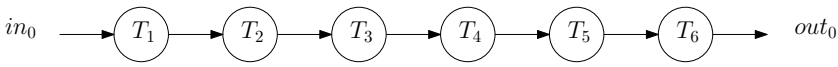
2.2.1. Algoritmos secuenciales

Es el algoritmo más simple y puede entenderse como aquel en el cual cada tarea debe finalizar antes de empezar la siguiente, que depende del resultado de la anterior.

Un ejemplo sería un cálculo en serie, en la cual, cada elemento es el anterior más una cantidad dada. El cálculo de la

serie de Fibonacci, donde cada elemento es la suma de los dos anteriores, puede ejemplificar este caso. La figura 2 muestra el DG de un algoritmo secuencial.

Figura 2. Representación DG de un algoritmo secuencial con una entrada in_0, una salida out_0, las tareas T_i y sus dependencias



Fuente: elaboración propia

Un ejemplo de implementación de la serie de Fibonacci en lenguaje de programación Java se incluye a continuación:

```
public static void main(String[] args) {
  int limit = Integer.parseInt(args[0]);
  int fibSum = 0;
  int counter = 0;
  int[] serie  = new int[limit];

  while(counter < limit) {
    serie = getMeSum(counter, serie);
    counter++;
}

public static int[] getMeSum(int index, int[] serie) {
  if(index==0) {
    serie[0] = 0;
  } else if(index==1) {
    serie[1] = 1;
  } else {
    serie[index]  = serie[index-1] + serie[index-2];
  }
  return serie;
}
```

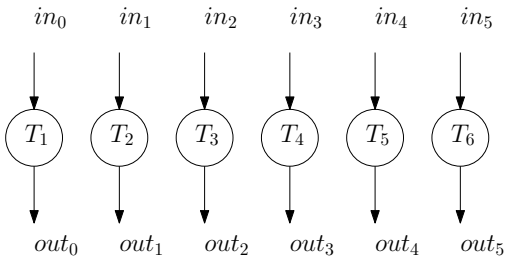
Solo imponiendo la condición para los dos primeros números pueden calcularse todos los valores de la serie hasta un límite especificado. En este caso se resuelve de forma elegante un algoritmo netamente secuencial con el uso de patrones recursivos, muy habituales en el desarrollo de algoritmos secuenciales.

2.2.2. Algoritmos paralelos

A diferencia de los anteriores, en los algoritmos paralelos las tareas de procesado son todas completamente independientes las unas de las otras, y en ningún momento muestran una correlación o dependencia.

A modo de ejemplo, un algoritmo puramente paralelo sería aquel que realiza un procesamiento para datos segmentados, sin realizar ningún tipo de agregación al finalizar los procesos, tal como la suma de elementos de un tipo A y la suma de elementos de un tipo B, dando como resultado las dos sumas.

Figura 3. Representación DG de un algoritmo paralelo con un conjunto de entradas (in_i), un conjunto de salidas (out_i) y diversas tareas (T_i) independientes



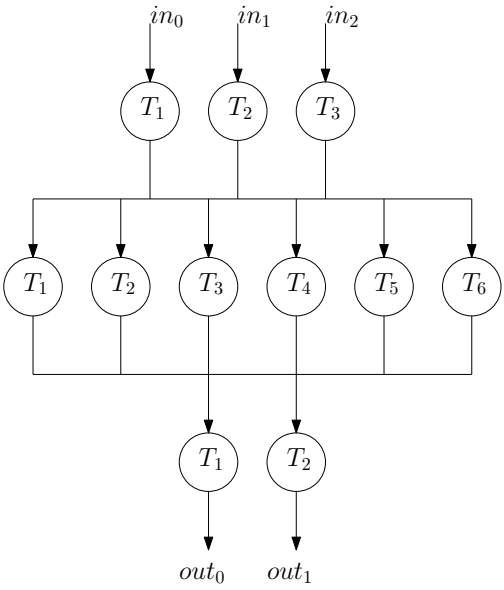
Fuente: elaboración propia

En la figura 3 vemos como cada tarea, que podría ser la suma de elementos segmentados $\{A, B, \ldots, F\}$, se realiza de forma independiente de las otras tareas.

2.2.3. Algoritmos serie-paralelos

A partir de los dos tipos de algoritmos ya descritos, podemos introducir un tercer tipo, que se compone de tareas ejecutadas de forma paralela y tareas ejecutadas de forma secuencial.

Figura 4. Algoritmo con conjuntos de diferentes tareas ejecutadas en paralelo en diferentes ciclos secuenciales



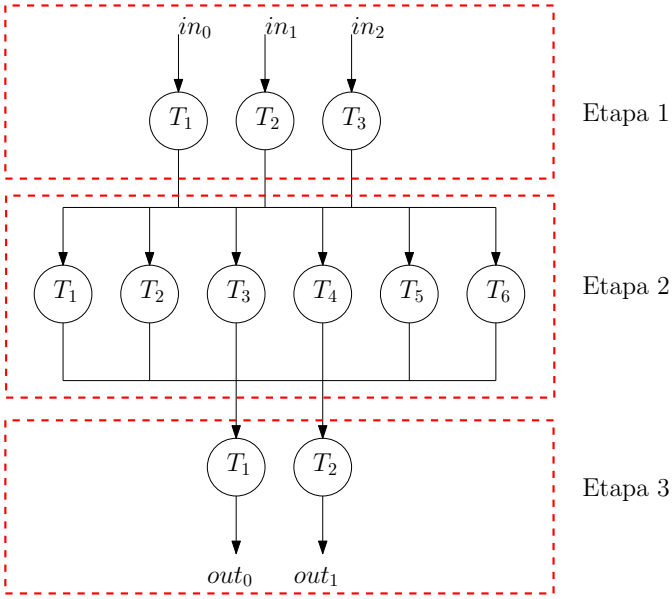
Fuente: elaboración propia

Así, ciertas tareas se descomponen en subtareas independientes entre sí y, tras su finalización, se agrupan los resultados de forma síncrona. Si es necesario se vuelve a dividir la ejecución en nuevas subtareas de ejecución en paralelo, tal y como muestra la figura 4.

Podemos añadir un nivel de abstracción a este tipo de algoritmo, tal que varias tareas que se ejecutan en paralelo componen una etapa (*stage*), el resultado de la cual es la entrada para otro conjunto de tareas que se ejecutaran en paralelo, formando otra etapa. Así, las tareas individuales se ejecutan de forma paralela pero las etapas se ejecutan de forma secuencial, lo que acaba generando un llamado *pipeline*. La figura 5 muestra un posible esquema de esta estructura.

Más adelante en el módulo volveremos a analizar este tipo de algoritmos, ya que es la metodología de procesado de Apache Spark.

Figura 5. Algoritmo serie-paralelo con etapas secuenciales



Fuente: elaboración propia

Un ejemplo de dichos algoritmos seria el paradigma Map-Reduce. Las tareas `Map` son independientes entre sí y su resultado se agrupa en un conjunto (típicamente menor) de tareas `Reduce`, siguiendo la metodología «divide y vencerás» (*divide & conquer*).

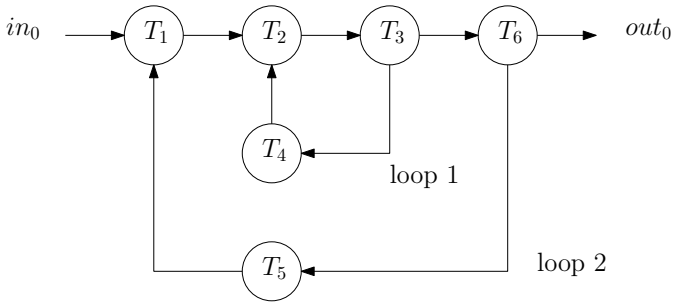
2.2.4. Otros algoritmos

Hay algoritmos que no pueden ser clasificados en ninguna de las categorías anteriores ya que el DG de dichos algoritmos no sigue ningún patrón.

En este caso, podemos distinguir dos tipos de algoritmos:

- DAG (*directed acyclic graph*): en el DAG no hay ciclos y el procesado muestra una «dirección», sin embargo no hay un sincronismo claro entre tareas. La figura 5 es un ejemplo de DAG. Es un tipo de algoritmo importante por ser el grafo que representa la metodología de ejecución de tareas de Apache Spark, que se estudiará más adelante.
- DCG (*directed cyclic graph*): son aquellos algoritmos que contienen ciclos en su procesado. Es habitual encontrar implementaciones en sistemas de procesado de señal, en el campo de las telecomunicaciones o compresión de datos, donde hay etapas de predicción y corrección. La figura 6 muestra un ejemplo de esta estructura.

Figura 6. Representación de un algoritmo con dependencias cíclicas



Fuente: elaboración propia

2.3. Eficiencia en la implementación de algoritmos

El uso de entornos de computación distribuidos requiere de una implementación en *software* adecuada de los algoritmos

para que pueda ser ejecutada de forma eficiente. Ejecutar un algoritmo en un entorno distribuido requiere de un análisis previo del diseño de dicho algoritmo para comprender el entorno en el que se va a ejecutar.

Las tareas que componen dicho algoritmo deben descomponerse en subtareas más pequeñas para poder ejecutarse en paralelo y, en muchas ocasiones, la ejecución de dichas subtareas debe ser síncrona, permitiendo la agrupación de los resultados finales de cada tarea.

Por tanto, paralelizar algoritmos está fuertemente relacionado con una arquitectura que pueda ejecutarlos de forma eficiente. No es posible paralelizar la ejecución de un algoritmo secuencial porque el procesador no va a saber qué tareas puede distribuir entre sus núcleos.

2.3.1. Notación Big O

La notación Big O permite cuantificar la complejidad de un algoritmo a partir del volumen de datos de entrada, que generalmente se identifica con la letra n.

Veamos algunos de los casos más relevantes:

- $O(1)$: es el caso de que un algoritmo requiera un tiempo constante para ejecutarse, independientemente del volumen de datos a tratar. Suele ser bastante inusual. Un ejemplo sería acceder a un elemento de un vector.

- $O(n)$: en este caso, la complejidad del algoritmo se incrementa de forma lineal con el volumen de la entrada de datos. Es decir, el tiempo de cálculo requerido aumenta de forma lineal respecto a la cantidad de datos que debe tratar. Por ejemplo, escoger el valor máximo de un vector de valores tendría esta complejidad, ya que se debe

recorrer una vez todo el vector para poder determinar cual es el valor máximo. Generalmente, cálculos que impliquen un bucle «for» que itere sobre los n elementos de entrada suele tener esta complejidad.

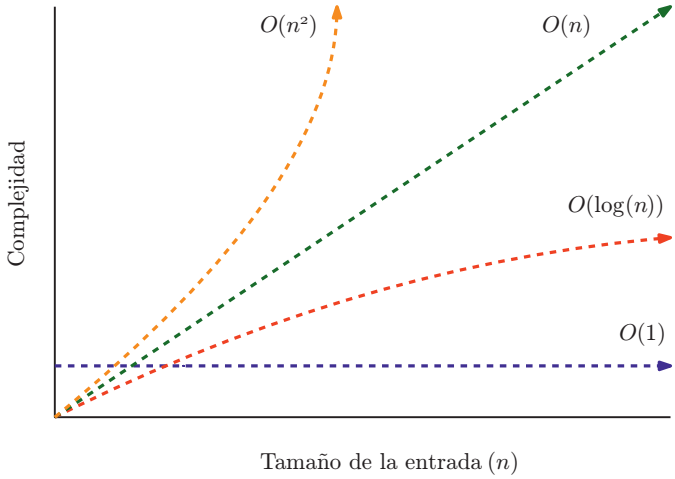
- $O(\log(n))$: también conocida como «complejidad logarítmica». Este tipo de algoritmos son aquellos en que el tiempo no se incrementa de forma lineal, sino que dicho incremento es menor a medida que aumenta el conjunto de datos. Generalmente, este tipo de complejidad se encuentra en algoritmos de búsqueda en árboles binarios, donde no se analizan todos los datos del árbol, solo los contenidos en ciertas ramas. De esta forma, la profundidad del árbol que deberemos analizar no crece que forma proporcional al volumen de las entradas.

- $O(n^2)$: este caso es conocido como «complejidad cuadrática». Indica que el tiempo de cálculo requerido crecerá de forma exponencial a partir del valor de n, es decir, del volumen de datos de entrada. Generalmente, cálculos que impliquen un doble bucle «for» que itere sobre los n elementos de entrada suele tener esta complejidad.

  Los algoritmos con este tipo de complejidad no suelen ser aplicables en datos masivos, ya que en estos casos el valor de n suele ser muy grande. Por lo tanto, no es computacionalmente posible resolver problemas con datos masivos empleando algoritmos de orden $O(n^2)$ o superior, como por ejemplo «cúbicos» $O(n^3)$.

La figura 7 muestra, de forma aproximada, como crece el tiempo de cálculo necesario según el volumen de datos de la entrada.

Figura 7. Representación de la complejidad de los algoritmos según la notación Big O



Fuente: elaboración propia

Un algoritmo puede paralelizarse y de este modo mejorar su tiempo de ejecución, aprovechar mejor los recursos disponibles, etc. Sin embargo, la eficiencia en la ejecución de un algoritmo también depende en gran medida de su implementación. Un ejemplo claro y fácil de entender son los diferentes mecanismos para ordenar elementos en una colección de elementos desordenados.

Ordenación de valores de una colección

La implementación más habitual para ordenar un vector de n elementos de mayor a menor consiste en iterar dos veces el mismo vector, utilizando una copia del mismo, comparando cada valor con el siguiente. Si es mayor intercambian sus posiciones, si no se dejan tal y como están. Este es un algoritmo

llamado «ordenamiento de burbuja» (*bubble sort*).[3] Su implementación es muy sencilla, sin embargo, es muy ineficiente, ya que debe recorrer el vector n veces. En el caso óptimo (ya ordenado), su complejidad es $O(n)$, pero en el peor de los casos es $O(n^2)$. El ordenamiento de burbuja, además, es más difícil de paralelizar debido a su implementación secuencial ya que recorre el vector de forma ordenada para cada elemento, siendo doblemente ineficiente.

Hay otros mecanismos para ordenar vectores mucho más eficientes, como el *quicksort*,[4] probablemente uno de los más eficientes y rápidos. Este mecanismo se basa en el divide y vencerás. El mecanismo del algoritmo no es trivial y su explicación está fuera de los objetivos de este módulo, sin embargo es interesante indicar que este algoritmo ofrece una complejidad del orden de $O(n \log(n))$. Además, al aproximar el problema dividiendo el vector en pequeños problemas es más fácilmente paralelizable.

Otro algoritmo de ordenamiento muy popular es el «ordenamiento por mezcla» (*merge sort*),[5] que divide el vector en dos partes de forma iterativa, hasta que solo quedan vectores de dos elementos, fácilmente ordenables. Este problema es también fácilmente paralelizable debido a que las operaciones en cada vector acaban siendo independientes entre ellas. Estos ejemplos de ordenación sirven para entender que un mismo algoritmo puede implementarse de formas distintas y, además, el mecanismo adecuado puede permitir la paralelización, lo que mejora su eficiencia final.

[3] https://en.wikipedia.org/wiki/Bubble_sort
[4] https://en.wikipedia.org/wiki/Quicksort
[5] https://en.wikipedia.org/wiki/Merge_sort

2.3.2. Computación paralela

En un entorno de computación paralela existen diversos factores relevantes que configuran su capacidad de paralelización.

Entornos multiprocesador

A nivel de un solo computador, algunos de los factores más importantes son:

1. **Numero de procesadores** y **número de núcleos** por procesador.

2. **Comunicación entre procesadores**: buses de comunicaciones en origen, actualmente mediante el sistema de comunicaciones *network on a chip* (NOC).[6]

3. **Memoria**: las aplicaciones pueden particionarse de modo que varios procesos compartan memoria (*threads* o hilos) o de modo que cada proceso gestione su memoria.

   Los sistemas de memoria compartida ofrecen la posibilidad de que todos los procesadores compartan la memoria disponible. Por lo tanto, un cambio en la memoria es visible para todos ellos, compartiendo el mismo espacio de trabajo (por ejemplo, variables).

   Existen arquitecturas de memoria distribuida entre procesadores, en la cual cada uno de ellos tiene su espacio de memoria reservado y exclusivo. Estas arquitecturas son escalables, pero más complejas desde un punto de vista de gestión de memoria.

[6]Network-on-chip (NoC) es un subsistema de comunicaciones en un circuito integrado.

4. **Acceso a datos**: cada proceso puede acceder a una partición de los datos o, por el contrario, puede haber concurrencia en el acceso a datos, lo que implica una capacidad de sincronización del algoritmo, incluyendo mecanismos de integridad de datos.

Entornos multinodo

En redes de múltiples computadores, algunos de los factores más importantes son:

1. En un entorno de **múltiples computadores** es necesario contar con comunicaciones de altas prestaciones entre ellos, mediante redes de alta velocidad, como por ejemplo Gigabit Ethernet.

2. Desde el punto de vista de la arquitectura de memoria, es una arquitectura híbrida. Un computador o nodo con múltiples procesadores es una máquina de memoria compartida, mientras que cada nodo tiene la memoria exclusiva y no compartida con el resto de nodos, siendo una arquitectura de memoria distribuida. Actualmente los nuevos *frameworks* de *big data* simulan las arquitecturas como sistemas de memoria compartida mediante *software*.

Unidad de procesamiento gráfico (GPU)

La unidad de procesamiento gráfico[7] (*graphics processor unit*, GPU) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga

[7]https://en.wikipedia.org/wiki/Graphics_processing_unit

de trabajo del procesador central. Típicamente las GPU están presentes en las tarjetas gráficas de los ordenadores y su dedicación es exclusiva a tareas altamente paralelizables de procesamiento gráfico, incrementando de forma exponencial la capacidad de procesado del sistema.

Aunque inicialmente las GPU se utilizaron para procesar gráficos (principalmente vértices y píxeles), actualmente su capacidad de cálculo se utiliza en otras aplicaciones en lo que se ha llamado *general-purpose computing on graphics processing units*[8] (GPGPU). Son especialmente relevantes en aplicaciones del ámbito de redes neuronales, *deep learning* y, en general, en entornos de altas prestaciones (*high performance computing*, HPC), de las que hablaremos con más detalle posteriormente.

[8]http://cort.as/-Ehg1

Capítulo 3

Introducción al aprendizaje automático

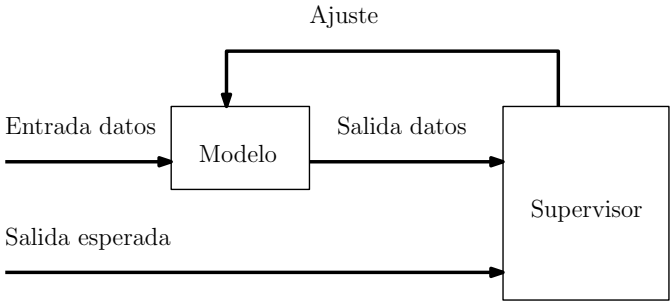
El aprendizaje automático (más conocido por su denominación en inglés, *machine learning*) es el conjunto de métodos y algoritmos que permiten a una máquina aprender de manera automática en base a experiencias pasadas.

Los tipos de métodos y tareas empleadas en el análisis de datos masivos es el mismo que en el caso de aprendizaje automático. En esencia, el principal cambio es que trabajamos con grandes volúmenes de datos, lo que nos obliga a tener especial atención a la complejidad de estos métodos y tareas, para poder desarrollarlas en un tiempo razonable de cómputo. Es decir, no todos los métodos y algoritmos de aprendizaje automático van a poder ser empleados en entornos de datos masivos, y como analistas o científicos de datos deberemos ser capaces de evaluar la complejidad de los métodos y escoger aquél que nos proporcione los resultados deseados con un coste (temporal y espacial) adecuado para el problema a tratar.

3.1. Tipología de métodos

Generalmente, un algoritmo de aprendizaje autónomo debe construir un modelo en base a un conjunto de datos de entrada que representan el conjunto de aprendizaje, lo que se conoce como *conjunto de entrenamiento*. Durante esta fase de aprendizaje, el algoritmo va comparando la salida de los modelos en construcción con la salida ideal que deberían tener estos modelos, para ir ajustándolos y aumentando la precisión. Esta comparación forma la base del aprendizaje en sí, y este aprendizaje puede ser **supervisado** o **no supervisado**. En el aprendizaje supervisado (figura 1), hay un componente externo que compara los datos obtenidos por el modelo con los datos esperados por este, y proporciona retroalimentación al modelo para que vaya ajustándose. Para ello, pues, será necesario proporcionar al modelo con un conjunto de datos de entrenamiento que contenga tanto los datos de entrada como la salida esperada para cada uno de esos datos.

Figura 1. Aprendizaje supervisado



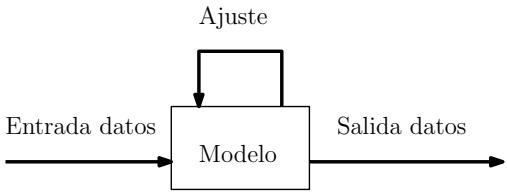
Fuente: elaboración propia

Todas ellas se basan en el paradigma del aprendizaje inductivo. La esencia de cada una de ellas es derivar inductivamente a partir de los **datos** (que representan la información del

entrenamiento), un **modelo** (que representa el conocimiento) que tiene utilidad predictiva, es decir, que puede aplicarse a nuevos datos.

En el aprendizaje no supervisado (figura 2), el algoritmo de entrenamiento aprende sobre los propios datos de entrada, descubriendo y agrupando patrones, características, correlaciones, etc.

Figura 2. Aprendizaje no supervisado



Fuente: elaboración propia

Algunos algoritmos de aprendizaje autónomo requieren de un gran conjunto de datos de entrada para poder converger hacia un modelo preciso y fiable, siendo pues el *big data* un entorno ideal para dicho aprendizaje. Tanto Hadoop (con su paquete de aprendizaje autónomo, Mahout) y Spark (con su solución MLlib) proporcionan un extenso conjunto de algoritmos de aprendizaje autónomo. A continuación los describiremos, indicando su clasificación y para qué plataformas están disponibles.

3.2. Tipología de tareas

Según el objetivo de nuestro análisis, podemos distinguir entre cinco grandes grupos de tareas, que revisaremos brevemente a continuación.

3.2.1. Clasificación

La clasificación (*classification*) es uno de los procesos cognitivos importantes, tanto en la vida cotidiana como en los negocios, donde podemos clasificar clientes, empleados, transacciones, tiendas, fábricas, dispositivos, documentos o cualquier otro tipo de instancias en un conjunto de clases o categorías predefinidas con anterioridad.

La tarea de clasificación consiste en asignar instancias de un dominio dado, descritas por un conjunto de atributos discretos o de valor continuo, a un conjunto de clases, que pueden ser consideradas valores de un atributo discreto seleccionado, generalmente denominado **clase**. Las etiquetas de clase correctas son, en general, desconocidas, pero se proporcionan para un subconjunto del dominio. Por lo tanto, queda claro que es necesario disponer de un subconjunto de datos correctamente etiquetado, y que se usará para la construcción del modelo.

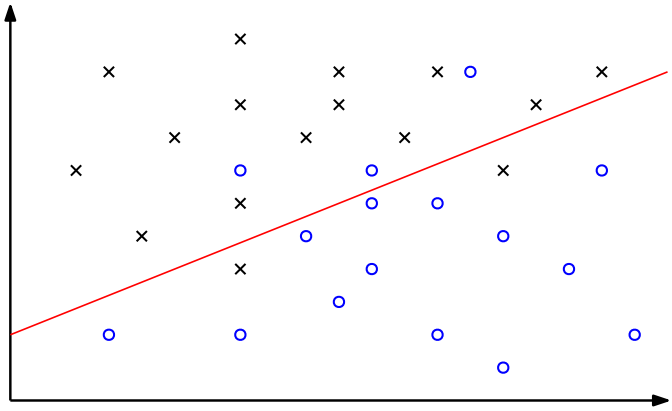
La función de clasificación puede verse como:

$$c : X \to C \tag{3.1}$$

donde c representa la función de clasificación, X el conjunto de atributos que forman una instancia y C la etiqueta de clase de dicha instancia.

Un tipo de clasificación particularmente simple, pero muy interesante y ampliamente estudiado, hace referencia a los problemas de clasificación binarios, es decir, problemas con un conjunto de datos pertenecientes a dos clases, i.e. $C = \{0, 1\}$. La figura 3 muestra un ejemplo de clasificación binaria, donde las cruces y los círculos representan elementos de dos clases, y se pretende dividir el espacio tal que separe a la mayoría de elementos de clases diferentes.

Figura 3. Ejemplo de clasificación



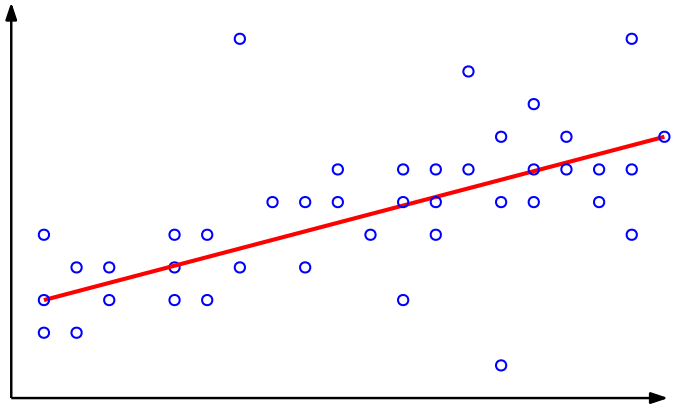
Fuente: elaboración propia

3.2.2. Regresión

Al igual que la clasificación, la regresión (*regression*) es una tarea de aprendizaje inductivo que ha sido ampliamente estudiada y utilizada. Se puede definir, de forma informal, como un problema de «clasificación con clases continuas». Es decir, los modelos de regresión predicen valores numéricos en lugar de etiquetas de clase discretas. A veces también nos podemos referir a la regresión como «predicción numérica».

La tarea de regresión consiste en asignar valores numéricos a instancias de un dominio dado, descritos por un conjunto de atributos discretos o de valor continuo, como se muestra en la figura 4, donde los puntos representan los datos de aprendizaje y la línea representa la predicción sobre futuros eventos. Se supone que esta asignación se aproxima a alguna función objetivo, generalmente desconocida, excepto para un subconjunto del dominio. Este subconjunto se puede utilizar para crear el modelo de regresión.

Figura 4. Ejemplo de regresión lineal



Fuente: elaboración propia

En este caso, la función de regresión se puede definir como:

$$f : X \to \mathbb{R} \tag{3.2}$$

donde f representa la función de regresión, X el conjunto de atributos que forman una instancia y $\mathbb{R}$ un valor en el dominio de los números reales.

Es importante remarcar que una regresión no pretende devolver una predicción exacta sobre un evento futuro, sino una aproximación (como muestra la diferencia entre la línea y los puntos de la figura). Por lo general, datos más dispersos resultarán en predicciones menos ajustadas.

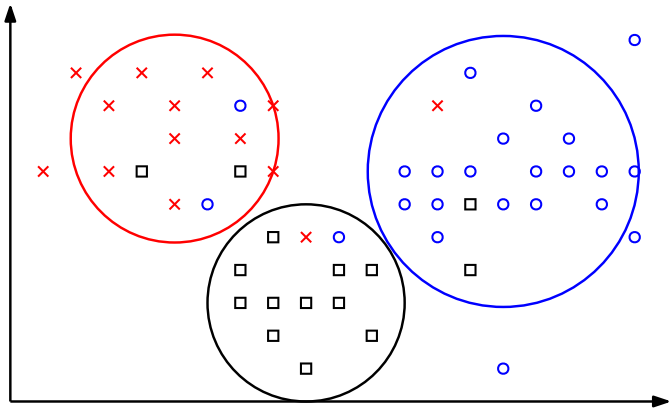
3.2.3. Agrupamiento

El agrupamiento (*clustering*) es una tarea de aprendizaje inductiva que, a diferencia de las tareas de clasificación y regresión, no dispone de una etiqueta de clase a predecir. Puede considerarse como un problema de clasificación, pero donde

no existen un conjunto de clases predefinidas, y estas se «descubren» de forma autónoma por el método o algoritmo de agrupamiento, basándose en patrones de similitud identificados en los datos.

La tarea de agrupamiento consiste en dividir un conjunto de instancias de un dominio dado, descrito por un número de atributos discretos o de valor continuo, en un conjunto de grupos (*clusters*) basándose en la similitud entre las instancias, y crear un modelo que puede asignar nuevas instancias a uno de estos grupos o *clusters*. La figura 5 muestra un ejemplo de agrupamiento, donde las cruces, círculos y cuadros pertenecen a tres clases de elementos distinto que pretendemos agrupar.

Figura 5. Ejemplo de agrupamiento



Fuente: elaboración propia

Un proceso de agrupación puede proporcionar información útil sobre los patrones de similitud presentes en los datos, como por ejemplo, segmentación de clientes o creación de catálogos de documentos. Otra de las principales utilidades del agrupamiento es la detección de anomalías. En este caso, el método de agrupamiento permite distinguir instancias que

con un patrón absolutamente distinto a las demás instancias «normales» del conjunto de datos, facilitando la detección de anomalías y posibilitando la emisión de alertas automáticas para nuevas instancias que no tienen ningún *clúster* existente.

La función de agrupamiento o *clustering* se puede modelar mediante:

$$h : X \rightarrow C_h \tag{3.3}$$

donde h representa la función de agrupamiento, X el conjunto de atributos que forman una instancia y C_h un conjunto de grupos o *clusters*. Aunque esta definición se parece mucho a la tarea de clasificación, una diferencia aparentemente pequeña pero muy importante consistente en que el conjunto de «clases» no está predeterminado ni es conocido *a priori*, sino que se identifica como parte de la creación del modelo.

3.2.4. Reducción de dimensionalidad

La reducción de dimensionalidad se basa en la reducción del número de variables que se vayan a tratar en un espacio multidimensional. Este proceso ayuda a reducir la aleatoriedad y el ruido a la hora de realizar otras operaciones de aprendizaje autónomo, puesto que se eliminan aquellas dimensiones que tengan escasa correlación con los resultados a estudiar.

El problema de la dimensionalidad es uno de los más importantes en la minería de datos y existen varias técnicas para descartar los parámetros que tienen menos peso en nuestro análisis. Las dos técnicas implementadas son la descomposición en valores singulares (SVD, *singular-value decomposition*) y el análisis de componentes principales (PCA, *principal component analysis*), probablemente la más popular.

En ocasiones, reduciendo la dimensionalidad no solo se ahorrará en cálculos a la hora de realizar regresiones o clasificaciones, sino que incluso estas serán más precisas, al reducir ruidos del conjunto de entrenamiento.

3.2.5. Filtrado colaborativo/sistemas de recomendación

Los sistemas de recomendación tratan de predecir los gustos/intenciones de un usuario en base a sus valoraciones previas y las valoraciones de otros usuarios con gustos similares. Son muy usados en sistemas de recomendación de música, películas, libros y tiendas *online*.

Cuando se crea un perfil de gustos del usuario, se crea utilizando dos formas o métodos en la recolección de características: implícitas o explícitas. Formas explícitas podrían ser solicitar al usuario que evalúe un objeto o tema particular, o mostrarle varios temas/objetos y que escoja su preferido. Formas implícitas serían tales como guardar un registro de artículos que el usuario ha visitado en una tienda, canciones que ha escuchado, etc.

Por ejemplo, imagínese la siguiente situación en un sistema de películas en red:

- El usuario A ha valorado como excelente *El padrino* y como mediocre *La guerra de las galaxias*.
- El usuario B ha valorado como mala *El padrino*, como excelente *La guerra de las galaxias* y como excelente *Star Trek*.
- El usuario C ha valorado como excelente *El padrino*, como aceptable *La guerra de las galaxias* y como excelente *Casablanca*.

Basándose en la afinidad del usuario A con el resto de usuarios, un sistema recomendador probablemente recomendaría *Casablanca* al usuario A.

3.3. Fases de un proyecto de aprendizaje automático

En general, los proyectos de aprendizaje automático o, en general, de minería de datos, se basan en tres etapas o fases más o menos independientes:

- La primera fase comprende la captura y preprocesamiento de los datos. En esta fase el objetivo principal es obtener los datos requeridos para poder realizar el análisis y responder las preguntas que han motivado el proyecto. Además de la captura, en general se habla también de preprocesamiento en el caso de ser necesario efectuar operaciones de corrección y/o modificación del formato de los datos antes de poder ser analizados.

- A continuación, se deben almacenar los datos de forma perdurable, para facilitar su posterior explotación. El almacenamiento se puede efectuar mediante sistemas de ficheros o bases de datos. El objetivo, en cualquier de los dos casos, es el mismo.

- La tercera fase del proyecto consiste, propiamente, en el proceso de análisis de los datos. En esta fase es donde se aplican los métodos o algoritmos de aprendizaje automático (o, en general, minería de datos) para obtener los resultados deseados, ya sean modelos predictivos, extracción de conocimiento, etc.

- La fase final del proyecto consiste en recabar el *feedback* de los usuarios respecto a las predicciones o conocimiento generado por el modelo. Esta información se utilizará para reentrenar el modelo e ir mejorando sus capacidades predictivas. En general, con el paso del tiempo los modelos se degradan, ya que los datos con los que han sido entrenados son cada vez más viejos y devienen obsoletos.

Estas etapas son similares (o iguales) en el caso de un proyecto de *big data*. Hace unos años, las herramientas y la tecnología que se empleaban en un proyecto de este tipo eran totalmente diferentes que en un proyecto de minería de datos tradicional. Por ejemplo, en la fase de captura, se utilizaban herramientas diseñadas específicamente para tratar con grandes volúmenes de datos, como Apache Sqoop. En la segunda fase, se empleaban sistemas de ficheros distribuidos o bases de datos NoSQL, desarrolladas para entornos distribuidos y grandes cantidades de datos heterogéneos. Y finalmente, en la fase de análisis se solían emplear herramientas como Mahout o MLib, que fueron desarrolladas en los entornos Apache Hadoop y Apache Spark, respectivamente.

Recientemente, se ha producido un cambio significativo en las herramientas empleadas en la fase de análisis. Las librerías que en su momento dominaron el análisis en entornos *big data* no han conseguido igualar los avances de otras librerías de aprendizaje automático, que haciendo uso de la tecnología multiprocesador y GPU, han conseguido aplicar un amplio abanico de métodos y algoritmos de forma satisfactoria a volúmenes importantes de datos.

En este sentido, actualmente las herramientas y tecnología para el análisis de entornos *big data* es la misma (o muy pare-

cida) que en los proyectos de minería de datos o aprendizaje automático tradicional. Las herramientas específicas para *big data* permiten la captura, manipulación o procesamiento y almacenamiento de datos masivos. Pero el análisis, propiamente, se suele realizar con herramientas no específicas de *big data*, como por ejemplo scikit-learn,[1] Keras[2] o TensorFlow.[3]

Los métodos y algoritmos de aprendizaje automático son muy amplios y variados, pero recientemente los métodos que están acaparando un foco y atención especial son las redes neuronales y los métodos de *deep learning*. Además, estos métodos pueden lidiar con grandes volúmenes de datos, mediante el uso de computadores multiprocesador o empleando la tecnología GPU.

3.4. Redes neuronales y *deep learning*

Actualmente, el aprendizaje automático es una de las disciplinas más atractivas de estudio y con mayor proyección profesional, especialmente tras la explosión del *big data* y su ecosistema tecnológico que ha abierto un enorme panorama de potenciales aplicaciones que no eran viables de resolver hasta ahora. Pero el mayor desafío del aprendizaje automático viene dado de las técnicas de clasificación no supervisada, esto es, que la máquina aprende por sí misma a partir de unos datos no etiquetados previamente.

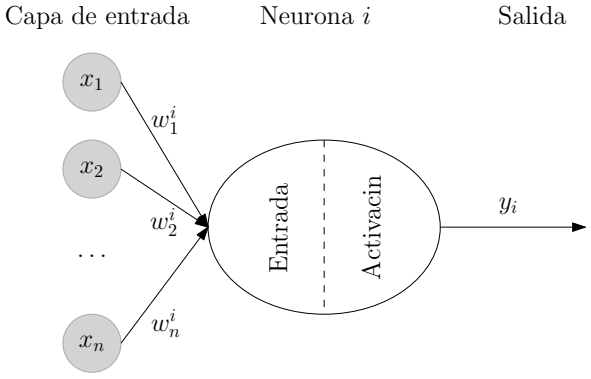
En el enfoque basado en las redes neuronales, se usan estructuras lógicas que se asemejan en cierta medida a la organización del sistema nervioso de los mamíferos, teniendo ca-

[1]http://scikit-learn.org
[2]https://keras.io/
[3]https://www.tensorflow.org/

pas de unidades de proceso (llamadas «neuronas artificiales») que se especializan en detectar determinadas características existentes en los objetos percibidos, permitiendo que dentro del sistema global haya redes de unidades de proceso que se especialicen en la detección de determinadas características ocultas en los datos.

Las redes neuronales artificiales (ANN, *artificial neural networks*) están formadas por un conjunto de neuronas distribuidas en distintas capas. Cada una de estas neuronas realiza un cálculo u operación sencilla sobre el conjunto de valores de entrada de la neurona, que en esencia son entradas de datos o las salidas de las neuronas de la capa anterior, y calcula un único valor de salida, que a su vez, será un valor de entrada para las neuronas de la siguiente capa o bien formará parte de la salida final de la red (figura 6).

Figura 6. Esquema de una neurona

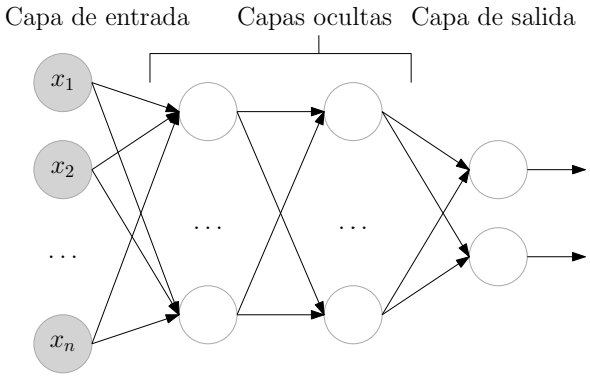


Fuente: elaboración propia

Las neuronas se agrupan formando capas, que son estructuras de neuronas paralelas. Las salidas de unas capas se conectan con las entradas de otras, de forma que se crea una estructura secuencial. La figura 7 presenta un esquema básico

de una red neuronal con la capa de entrada, múltiples capas ocultas y la capa de salida.

Figura 7. Esquema de red neuronal con múltiples capas ocultas



Fuente: elaboración propia

Las redes neuronales no son un concepto nuevo y, aunque su funcionamiento siempre ha sido satisfactorio en cuanto a resultados, el consumo de recursos para entrenar una red neuronal siempre ha sido muy elevado, lo que ha impedido en parte su completo desarrollo y aplicación. Sin embargo, tras la irrupción de las GPU han vuelto a ganar protagonismo gracias al elevado paralelismo requerido para entrenar cada neurona y el potencial beneficio que ofrecen las GPU para este tipo de operaciones.

Estas redes se suelen inicializar con valores aleatorios, y requieren de un proceso de entrenamiento con un conjunto de datos para poder «aprender» una tarea o función concreta. Este proceso de aprendizaje se realiza utilizando el método conocido como Backpropagation.[4] En esencia, este método calcula el error que comete la red en la predicción del valor para

[4]https://en.wikipedia.org/wiki/Backpropagation

un ejemplo dado e intenta modificar los parámetros de todas las neuronas de la red para reducir dicho error.

Este tipo de algoritmos tuvo su época de esplendor hace ya unas décadas. Su principal limitación se encuentra en el proceso de aprendizaje que se da en las redes con cierto número de capas ocultas (capas intermedias, es decir, que se encuentran entre la entrada de datos y la salida o respuesta final de la red). En estos casos, se produce lo que se conoce como el problema de la desaparición o explosión del gradiente,[5] que básicamente provoca problemas en el proceso de aprendizaje de la red.

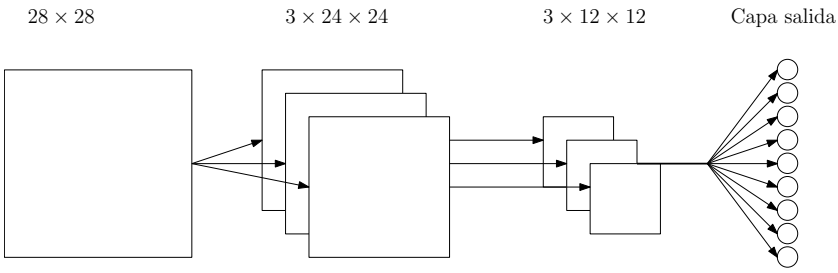
Estos problemas han sido superados años más tarde con la adopción de nuevas funciones de activación (por ejemplo, la función ReLU),[6] dando inicio a lo que se conoce actualmente como *deep learning*. Se ha modificado la estructura básica de las redes neuronales, creando, por ejemplo, redes convolucionales que permiten crear distintas capas donde el conocimiento se va haciendo más abstracto. Es decir, las primeras capas de la red se pueden encargar de identificar ciertos patrones en los datos, mientras que las capas posteriores identifican conceptos más abstractos a partir de estos patrones más básicos. Por ejemplo, si queremos que una red neuronal pueda detectar cuando aparece una cara en una imagen, este enfoque buscaría que las primeras capas de la red se encarguen de detectar la presencia de un ojo en alguna parte de la imagen, de una boca, etc. Así, las siguientes capas se encargarían de combinar esta información e identificar una cara a partir de la existencia de las partes que la forman (ojos, boca, nariz, etc). De esta forma vamos avanzando de una información más básí-

[5]http://cort.as/-FlGL
[6]http://cort.as/-FlGP

ca hacia un conocimiento más abstracto. La figura 8 muestra un esquema, aunque muy general, de una red convolucional.

Existen varios entornos y bibliotecas para trabajar con redes neuronales y *deep learning* que se ejecutan en las potentes GPU modernas, generalmente mediante la API CUDA. Probablemente, una de las más famosas es el *framework* TensorFlow de Google.

Figura 8. Esquema general de una red neuronal convolucional



Fuente: elaboración propia

3.4.1. TensorFlow

TensorFlow[7] es una API de código abierto desarrollada por Google para construir y entrenar redes neuronales. TensorFlow es una potente herramienta para hacer uso extensivo de los recursos locales de un ordenador, ya sea CPU o GPU.

Así, se trata de un sistema de programación en el que representamos cálculos en forma de grafos, donde cada nodo del grafo (llamados *ops*) realiza una operación sobre uno o varios «tensores» (que puede ser un vector, un valor numérico o una matriz) y como resultado devuelve de nuevo un tensor.

[7]https://www.tensorflow.org/

El grafo que representa el TensorFlow describe los cálculos a realizar, de ahí la terminología *flow*. El grafo se lanza en el contexto de una sesión de TensorFlow (*session*), que encapsula en entorno de operación de nuestra tarea, incluyendo la ubicación de las tareas en nuestro sistema, ya sea CPU o GPU.

Aunque TensorFlow dispone de implementaciones paralelizables también puede ser distribuido[8] a través de un *clúster* de ordenadores, en el cual cada nodo se ocupa de una tarea del grafo de ejecución.

Existen implementaciones de TensorFlow en varios lenguajes de programación, como Java, Python, C++ y Go.[9] Asimismo, permite el uso de GPU, lo que puede resultar en un incremento importante de rendimiento en el entrenamiento de redes neuronales profundas, por ejemplo.

En este punto, nos preguntamos: ¿podríamos combinar lo mejor del procesado distribuido (Spark) y una herramienta de aprendizaje automático tan potente como TensorFlow?

Como ya hemos introducido en el caso de utilizar modelos entrenados con scikit-learn, podríamos utilizar las funcionalidades de Spark y su capacidad de propagar variables a los nodos mediante la opción *broadcast* para enviar el modelo a través de múltiples nodos y dejar que cada nodo aplique el modelo a un corpus de datos.

En la actualidad hay muchas iniciativas en las que se combina Spark para la evaluación de modelos y TensorFlow para realizar tareas de entrenamiento, validación cruzada de modelos o aplicación y predicción del modelo.

[8]https://www.tensorflow.org/deploy/distributed

[9]https://golang.org/

Recientemente ha aparecido una nueva API, llamada Deep Learning Pipelines,[10] desarrollada por Databricks (los mismos desarrolladores de Apache Spark), para combinar técnicas de *deep learning*, principalmente TensorFlow, con Apache Spark.

[10]http://cort.as/-Eni0

Parte II

Tipologías y arquitecturas de un sistema *big data*

Capítulo 4

Fundamentos tecnológicos

Como hemos comentado anteriormente, 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 áreas de las tecnologías de la información y comunicaciones, también conocidas como TIC. Desde el desarrollo de nuevos sistemas de almacenamiento de datos, como serian 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 múltiples 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 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 transparente para el usuario su uso, 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 a 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 per-

miten 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*. 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 (White, 2015).

- **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 (Zaharia, Chambers, 2017).

- **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 (Biery, 2017).

En todos estos 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, como los descritos anteriormente en este libro.

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

4.1. Tipología 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 y procesar. Generalmente diferenciamos tres tipos de fuentes de datos masivos. Nótese que esta clasificación aplica a 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 por qué su información no se puede desgranar a 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*)**. 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.

4.2. ¿Cómo procesamos toda esta información?

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 *clusters* 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 paradigma 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[1] a la vez.
- Ha de proveer un entorno de trabajo cómodo para los programadores de aplicaciones.
- Dispone de un sistema de red que conecta 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.

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

[1]A partir de ahora cuando nos refiramos a un conjunto de servidores hablaremos de *cluster*.

2004 desarrolló la metodología de procesado de datos MapReduce [2], 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.

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.

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.

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.

[2] http://research.google.com/archive/mapreduce.html

Cálculo iterativo. Realizar 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, como por ejemplo el entrenamiento de redes neuronales, visto en la sección 3.4.

Posteriormente, en el año 2014, Matei Zaharia creo Apache Spark,[3] 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.

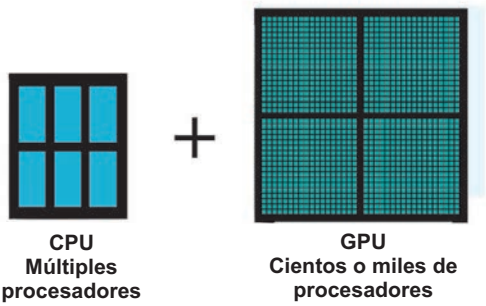
En paralelo a estas mejoras desde el año 2007, empresas como NVIDIA, empezaron a introducir el uso de GPU como alternativa al cálculo tradicional en CPU. Las GPU disponen de procesadores mucho más simples que las CPU, esto permite que en una sola tarjeta gráfica instalada en un servidor puedan integrarse un mayor número de procesadores permitiendo realizar un gran número de cálculos numéricos en paralelo. En un entorno con GPU, el sistema traslada las partes de la aplicación con mayor carga computacional y más altamente paralelizables a la GPU, dejando el resto del código ejecutándose en la CPU.

Una forma sencilla de entender la diferencia entre una GPU y una CPU es comparar la forma en que procesan las diferentes tareas. Una CPU está formada por varios núcleos optimizados para el procesamiento en serie. Típicamente, una CPU puede contar con varios núcleos optimizados (4 u 8 simulados

[3]http://spark.apache.org/

con tecnología *hyperthreading*),[4] mientras que una GPU consta de millares de núcleos más pequeños y eficientes diseñados para manejar múltiples tareas simultáneamente. Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU (véase figura 1), no cualquier problema se puede beneficiar de una implementación mediante GPU.

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



Fuente: elaboración propia

Desde el punto de vista de sus componentes primarios, tanto CPU como GPU son iguales: 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 realizar 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 en paralelo, una y otra vez.

[4] *Hyperthreading* es una tecnología de Intel que permite ejecutar programas en paralelo en un solo procesador, simulando el efecto de tener realmente dos núcleos en un solo procesador.

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.

Pese a que cualquier algoritmo que sea implementable en una CPU puede ser lo también en una GPU, ambas implementaciones pueden no ser igual de eficientes en las dos arquitecturas. En este sentido, los algoritmos con un altísimo grado de paralelismo, llamados *embarrassingly parallel* (embarazosamente paralelizables en su traducción literal), sin necesidad de estructuras de datos complejas y con un elevado nivel de cálculo aritmético son los que mayores beneficios obtienen de su implementación en GPU.

Inicialmente la programación de GPU se hacía con lenguajes de bajo nivel como lenguaje ensamblador (*assembler*) o más específicos para el procesado gráfico. Sin embargo, actualmente el lenguaje más utilizado es CUDA[5] (Compute Unified Device Architecture). CUDA no es un lenguaje en sí, sino un conjunto de extensiones a C y C++ que permite la paralelización de ciertas instrucciones para ser ejecutadas en la GPU del sistema.

Como inconvenientes cabe indicar que las GPU tienen latencias altas, lo que presenta un problema en tareas con un corto tiempo de ejecución. Por el contrario, son muy eficientes

[5]https://en.wikipedia.org/wiki/CUDA

en tareas de elevada carga de trabajo, con cálculos intensivos y con largo tiempo de ejecución.

4.3. Computación científica

Antes de introducir con más detalle las tecnologías más comerciales del *big data* merece la pena hablar de la **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 en 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 la sección 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 estruc-

tural/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 minimice el error de clasificación, encontrar los precios de renovación de un conjunto de seguros que maximice 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 requieren 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.

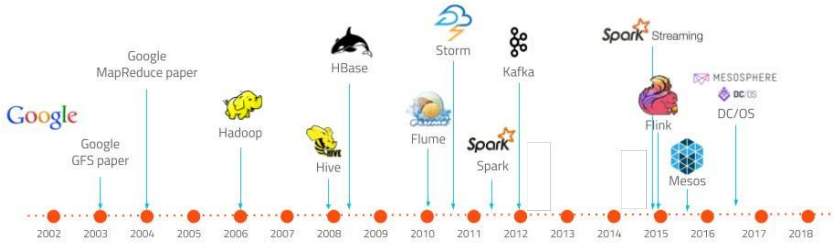
Capítulo 5

Arquitectura de un sistema *big data*

Es posible capturar, almacenar, procesar y analizar *big data* de muchas formas. Cada fuente de datos de las que se consideran de *big data* tiene distintas características, que incluyen las mencionadas anteriormente como frecuencia, volumen, velocidad, el tipo y la veracidad de los datos. Cuando se procesan y almacenan grandes volúmenes de datos, entran en juego dimensiones adicionales, como el gobierno, la seguridad y las políticas. Elegir una arquitectura y desarrollar una solución apropiada de *big data* para cada escenario es un reto, ya que se deben considera muchos factores. En la figura 1 podemos ver un resumen de las principales tecnologías de *big data*, así como su fecha de aparición.

En este apartado introduciremos los principales componentes de una arquitectura *big data*. Empezaremos describiendo la estructura general de estos sistemas, luego hablaremos sobre como almacenar los datos en sistemas de archivos distribuidos o en bases de datos NoSQL. Seguidamente, pasaremos a des-

Figura 1. Línea de tiempo de tecnologías *big data*



Fuente: elaboración propia

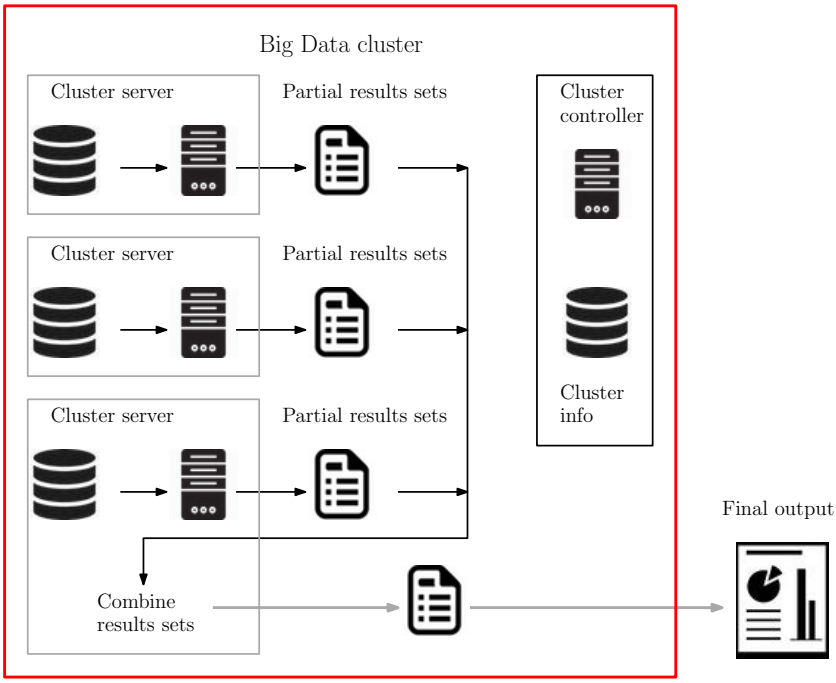
cribir dos tecnologías diferentes para poder procesar grandes volúmenes de información de forma eficiente haciendo uso de la computación distribuida y científica. Finalmente, explicaremos como se gestionan todos los componentes de un *cluster* usando un gestor de recursos, como por ejemplo YARN.

5.1. 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.

La primera característica común a destacar es que cada servidor del *cluster* posee su propio disco, memoria RAM y CPU. Esto permite crear un sistema de computo distribuido con ordenadores heterogéneos y de propósito general, no diseñados de de forma especifica para crear *clusters*. Esto reduce mucho 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

Figura 2. Ejemplo de estructura general de un posible sistema de *big data*



Fuente: elaboración propia

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 como de intensivo sea el intercambio de datos entre los servidores.

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 *cluster* 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 que hay problemas en ciertos nodos.

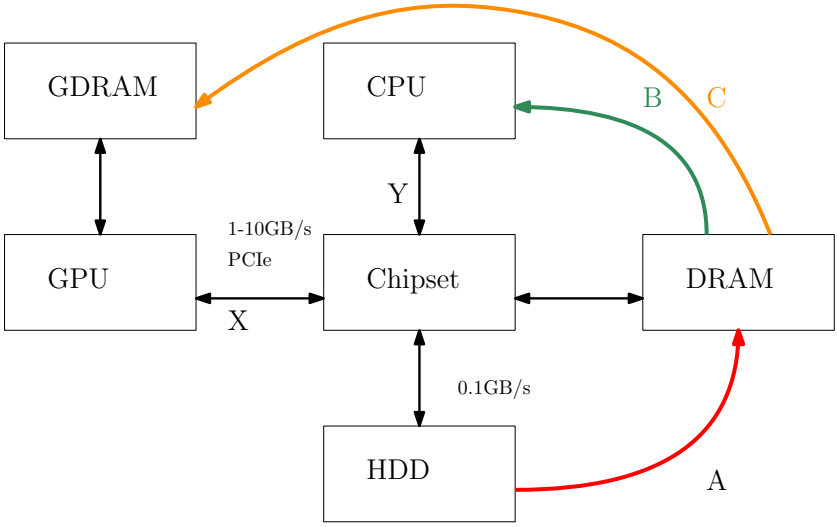
El uso combinado de este conjunto de servidores es posible, como veremos más adelante, gracias al uso de un sistema de ficheros distribuido y al hecho que los cálculos a realizar se implementan en un entorno de programación distribuida, como pueden ser Hadoop o Spark.

5.1.1. 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 únicamente CPU ya que para poder obtener el máximo rendimiento de todos los procesadores de una GPU es necesario garantizar que todos estos reciben un flujo de datos constante durante la ejecución de las tareas del *cluster* (Barlas, 2015).

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.

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



Fuente: elaboración propia

El acrónimo DRAM corresponde a las siglas *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 periodo, 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 $\longrightarrow$ *chipset* $\longrightarrow$ CPU).
- **Flecha C**. Procesamiento de datos con una GPU (transferencia de datos: DRAM $\longrightarrow$ *chipset* $\longrightarrow$ CPU $\longrightarrow$ *chipset* $\longrightarrow$ GPU $\longrightarrow$ GDRAM $\longrightarrow$ GPU).

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. Como estas comparaciones varían cada poco tiempo, aquí nos centraremos en describir cual 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.

Aunque es muy probable que cuando usamos un supercomputador este 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[1] (punto Y en la figura 3), una GPU intercambia datos con DRAM a la velocidad de 1-10 GBps (vea el Punto X de la misma figura). Aunque algunos sistemas pueden alcanzar hasta unos 10 GBps (PCIe v3),[2] en la mayoría de las configuraciones estándar los flujos de datos entre una GPU (GDRAM) y la DRAM del servidor disponen de una velocidad aproximada de 1 GBps.

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.

[1]GBps: gigabytes por segundo.

[2]PCIe V3: Peripheral Component Interconnect Express de tercera generación.

5.2. Sistema de archivos

Una vez ya hemos visto la arquitectura general de un sistema de *big data*, hemos hablado de la diferencia entre cómputo en CPU y GPU, y de que características deben tener las diferentes jerarquías de memoria en un servidor, es el momento de centrarnos en como 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 (*hard disk drive*, HDD) o memorias de estado sólido (*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.

De forma breve podemos decir que un disco duro tradicional 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 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, 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 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 organizarlos con algún tipo de orden. Para acceder a un archivo se debe proporcionar su **ruta** (orden jerárquico de carpetas y sub-carpetas) y el **nombre** de archivo seguido de una **extensión** (por ejemplo, .txt) que indica el contenido del archivo.[3]

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**

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

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 ingles *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 que nos permite acceder a una gran cantidad de datos de forma distribuida, pero sufre de 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, tienen que ser copiados por la red de un nodo a otro. Este proceso es lento y costoso.

Para solucionar este problema se creo 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 la siguiente sección introduciremos su funcionamiento y sus principales componentes.

5.2.1. 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 en profundidad.[4]

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 caracterizan 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 *cluster* 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*. Dentro de estos nodos hay uno especial denominado JobTracker que ejerce como el nodo principal o director de orquesta, mediante el cual se va a distribuir el tratamiento y procesado de los ficheros en los nodos *worker*, que realizarán el trabajo. Este nodo es importante porque almacena los metadatos necesarios para encontrar los ficheros dentro del sistema de archivos.

[4]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 KB.

5.2.2. Bases de datos NoSQL

Durante mucho tiempo, las bases de datos relacionales fueron la única alternativa a los sistemas de ficheros para almacenar grandes volúmenes de información.

Actualmente, los nuevos requerimientos del mundo *big data* hacen que las bases de datos relacionales no puedan solucionar toda la problemática relacionada con el almacenamiento de datos. Esto ha provocado la aparición de nuevos sistemas de bases de datos, llamados NoSQL, que permiten dar una solución a los problemas de escalabilidad y rendimiento del 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.

5.3. 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 como se pueden aplicar las ideas del cálculo distribuido sobre arquitecturas basadas en GPU.

5.3.1. 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) usando *clusters* de ordenadores y (2) usando *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 se pueden calcular 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 *cluster*.

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. Ca-

da vez que un nodo acaba una tarea, tanto un `Map` como un `Reduce`, guarda el resultado en el sistema de archivos para garantizar la consistencia y poder continuar la ejecución en caso de producirse un fallo en el *hardware*.

5.3.2. Apache Spark. Procesamiento distribuido en memoria principal

Como hemos descrito en la sección anterior, MapReduce utiliza el disco duro para guardar los resultados parciales de las funciones de `Map` y `Reduce`. 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.

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 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.

La principal ventaja de Spark respecto a Hadoop, es que guarda todas las operaciones sobre los datos en memoria. Esta es la clave del su buen rendimiento.

En Spark, a diferencia de Hadoop, no utilizaremos una colección de datos distribuidos en el sistema de archivos, sino que usaremos los RDD (*resilient distributed datasets*).[5]

[5]Para una información más detallada sobre RDD consultar: http://cort.as/-EiRa

Los RDD son colecciones lógicas, inmutables y particionadas de registros de datos distribuidos en la memoria principal de los nodos del *cluster* y que pueden ser reconstruidas si alguna partición se pierde (no necesitan ser materializadas pero sí 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 son tolerantes a fallos. Para ello mantienen un registro de las transformaciones realizadas llamado el linaje (*lineage* en inglés) del RDD. Este *lineage* permite que los RDD se reconstruyan en caso de que una porción de datos se pierda debido a un fallo en un nodo del *cluster*.

5.3.3. 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 RMI (*remote method invocation*) de Java para poder comunicarse de forma eficiente entre los diferentes nodos del *cluster*.

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 master 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.

5.3.4. GPUs. Simplicidad y alta paralelización

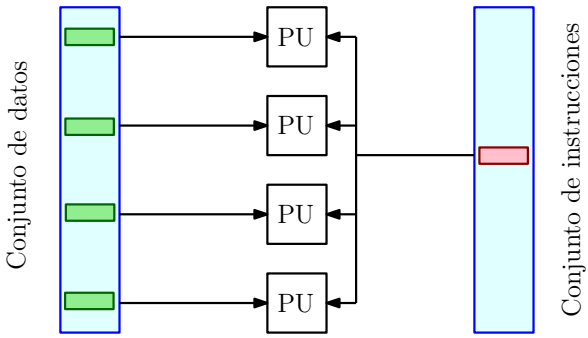
Como ya hemos comentado, los servidores con GPU se basan arquitecturas *many-core*. Estas arquitecturas disponen de una cantidad ingente de núcleos (procesadores) que realizan una misma operación sobre múltiples datos.

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 secciones 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 instrucción es ejecutada de manera sincro-

nizada por todas las unidades de procesamiento. La Figura 4 describe de forma gráfica esta idea.

Figura 4. Estructura funcional del concepto «una instrucción, múltiples datos»



Fuente: elaboración propia

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.

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

5.4.1. Apache MESOS

Apache Mesos[6] 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 *cluster*.

Bajo esta idea, Mesos ofrece una capa de abstracción entre los servidores y los recursos. Es decir, que básicamente lo que nos ofrece 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 *cluster* de Mesos, tendremos un único nodo *master* (del que podemos tener réplicas inactivas), que se ocupará de gestionar todas las peticiones de recursos que reciba el *cluster*. El resto de nodos serán nodos *slaves*, que son los encargados de ejecutar los trabajos de los entornos de ejecución (*e.g.* Spark, Hadoop, ...). Estos nodos reportan su estado directamente al nodo *master* activo. Es decir, el nodo *master* también se encarga del seguimiento y control de los trabajos en ejecución.

[6]http://mesos.apache.org/

5.4.2. YARN (Yet Another Resource Negotiator)

YARN (Yet Another Resource Negotiator)[7] 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.

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.

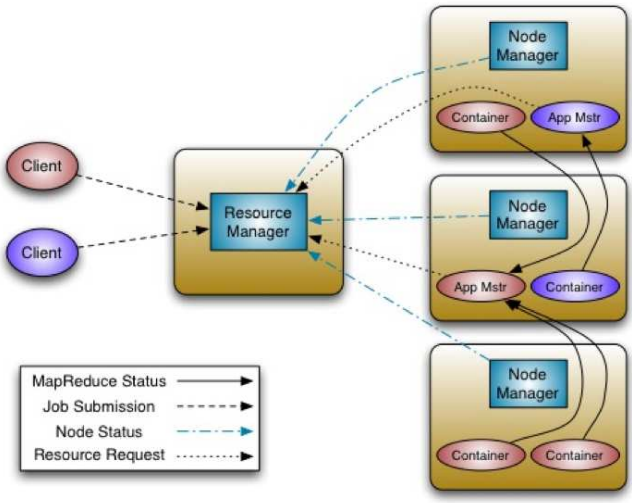
Esto deriva en que ahora, una aplicación, es simplemente un Job (en el sentido tradicional de MapReduce).

En la figura 5 ilustra la arquitectura de YARN para que se pueda entender de forma más clara.

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

[7]http://cort.as/-EiSI

Figura 5. Ejemplo de arquitectura en YARN



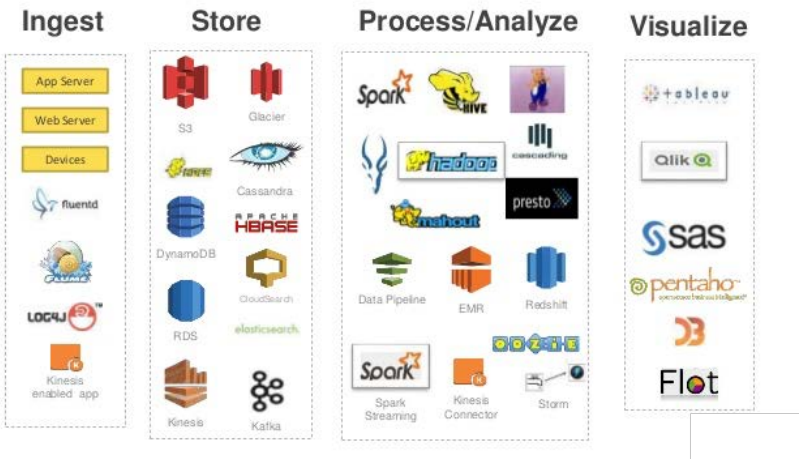
Fuente: https://hadoop.apache.org/docs/

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, es decir, les solicita recursos para poder trabajar.

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

En el mercado actual del *big data* es posible encontrar una gran cantidad de plataformas, *softwares*, 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.

Figura 6. Descripción de las principales capas de un sistema de *big data*



Fuente: Amazon Web Services

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 6 se representan las cuatro capas principales de un sistema *big data*, además se incluyen algunos de los *softwares* y/o tecnologías más comunes que podemos encontrar para resolver los problemas de esa capa.

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

- *Ingest layer.* Es la capa encargada de capturar datos. Encontramos herramientas como Apache Flume,[8] un sistema de gestión de *streams* que permite transformar datos en crudo antes de ser almacenados en el sistema.

[8]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 eficiente. 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[9] permiten una visualización de datos interactiva de forma relativamente sencilla.

[9]https://www.tableau.com/

Capítulo 6

Escenarios de procesamiento distribuido

En este capítulo 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.

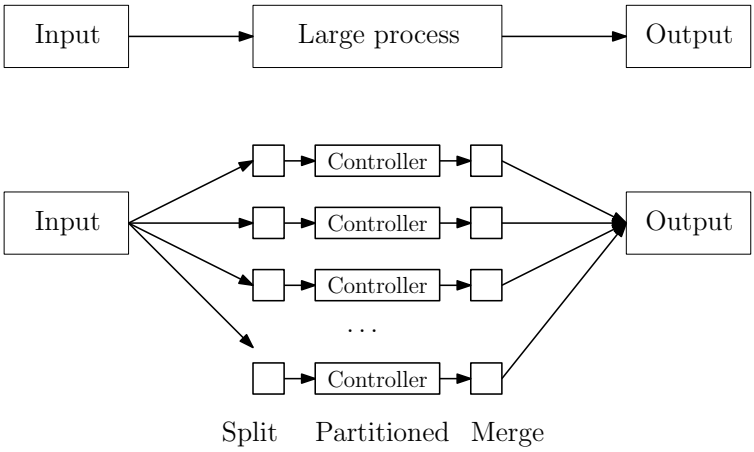
Por norma general, este tipo de ejecuciones se utilizan en tareas repetitivas sobre grandes conjuntos de información. Un ejemplo sería el procesamiento de *logs* de un servidor web. Un fichero de *logs*, o 'bitácora' en español, es un fichero secuencial que almacena todos los eventos que ha procesado un servidor. Normalmente se guardan en un formato estándar para poder ser procesados de forma sencilla.

En este caso, cada noche se procesarían los ficheros de *logs* generados por los servidores web durante ese día. Este procesamiento en *batch* 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 *cluster*.

En la figura 1 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 1. Ejemplo de aplicación de proceso en *batch*



Fuente: elaboración propia

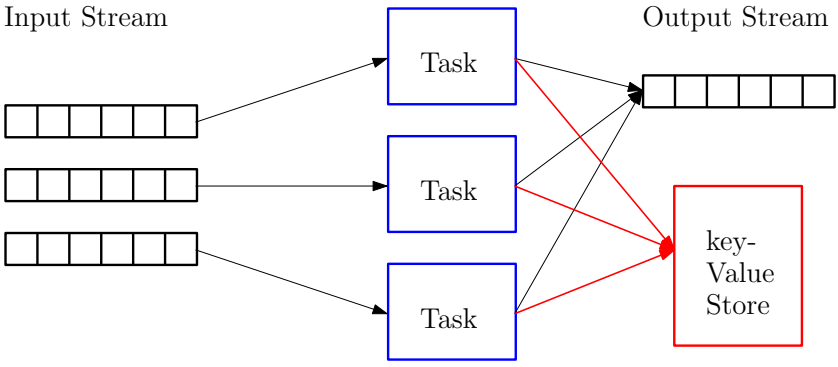
6.2. Procesamiento en *stream*

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 aquellos 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* (Kumar, Singh, 2017).

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

Figura 2. Ejemplo de aplicación de proceso en *stream*



Fuente: elaboración propia

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 2 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*.

Priorización de recursos en una página web. Intentar determinar qué clientes de los que están accediendo a una tienda en línea 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 tien-

da *online* sea mejor y aumente aún más su probabilidad de compra.

6.3. Procesamiento de grafos

Las redes de distinta índole (como por ejemplo, redes sociales o redes de comunicación) siempre han estado muy presentes en nuestra sociedad. Pero hasta hace relativamente poco tiempo era difícil poder capturar esa información para analizarla y procesarla. Recientemente, con la popularización, principalmente de las redes sociales, se han desarrollado nuevas tecnologías, modelos y algoritmos para representar, almacenar y analizar este tipo de información.

Los grafos, que nos permiten representar las redes reales en un computador, permiten representar estructuras y realidades más complejas que los tradicionales datos relacionales, que utilizan el formato de tuplas. En un modelo basado en grafos, cada entidad puede presentar, al igual que los datos relacionales, una serie de atributos en cualquier tipo de formato. Pero además, el formato de grafo permite representar de un modo más rico las relaciones que puedan existir entre las distintas entidades que forman el conjunto de datos.

Estas formas de representación presentan importantes retos para la captura, el almacenamiento y el procesamiento de este tipo de datos. Los métodos tradicionales, desarrollados para trabajar con datos relacionales, no suelen ser los más adecuados para trabajar con grafos.

En un escenario típico de procesamiento de grafos encontraremos modelos de almacenamiento muy distintos a los empleados en los dos escenarios anteriores, haciendo uso de bases de datos NoSQL en grafo en muchos de los casos. Para el

procesamiento y el análisis se emplean modelos y algoritmos propios para grafos, en lo que se conoce como «minería de grafos» (*graph mining*).

Análisis de Twitter. Twitter se ha convertido en una de las redes sociales más populares de la actualidad, con un elevado número de usuarios (que crece día a día) que genera un volumen exponencial de información. La información que se puede obtener se suele representar como un grafo, donde tenemos los nodos (usuarios con sus atributos: edad, localidad, información de perfil, etc.) y un conjunto de relaciones entre ellos («seguir» a otro usuario, hacer un *retweet* o un «me gusta» de uno de sus tweets, etc.). Para almacenar esta información, donde las relaciones entre usuarios y *tweets* es muy importante, las bases de datos relacionales han demostrado no ser suficientemente expresivas para representar un modelo tan rico y dinámico. En el caso del análisis, los algoritmos tradicionalmente empleados en minería de datos tampoco sirven, ya que no son capaces de capturar el significado de las relaciones. Para superar estas limitaciones se han desarrollado nuevas tecnologías para el almacenamiento, como las bases de datos NoSQL en grafo, y para el análisis, como los algoritmos de detección de comunidades o de difusión de información.

6.4. 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*.

Minado de *bitcoins*. 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. En función 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 alfanumé-

ricas aleatorias, los sistemas con GPU son la mejor opción ya que permiten repetir una misma operación matemática en paralelo de forma eficiente.

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 se puede utilizar 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* se puede distribuir o almacenar libremente, ya que solo se utiliza para fines de comparación.

Parte III

Procesamiento por lotes (*batch*)

Capítulo 7

Captura y preprocesamiento por lotes

En este capítulo trataremos las particularidades de los procesos de captura y preprocesamiento de datos por lotes o *batch* en entornos de datos masivos (*big data*).

7.1. Conceptos básicos

El primer paso para cualquier proceso de análisis de datos consiste en la captura de estos, es decir, debemos ser capaces de obtener los datos de las fuentes que los producen o almacenan, para posteriormente, poder almacenarlos y analizarlos.

En este sentido, el proceso de captura de datos se convierte en un elemento clave en el proceso de análisis de estos, ya sean masivos o no. No será posible, en ningún caso, analizar datos si no hemos sido capaces de capturarlos cuando estaban disponibles.

Existen dos grandes bloques cuando hablamos de captura de datos según la naturaleza de producción de los mismos:

- **Datos estáticos**: son datos que ya se encuentran almacenados en algún lugar, ya sea en formato de ficheros (por ejemplo, ficheros de texto, JSON, XML, etc.) o en bases de datos.

- **Datos dinámicos** o en *streaming*: son datos que se producen de forma continua, y que deben ser capturados durante un umbral limitado de tiempo, ya que no estarán disponibles una vez pasado este.

7.2. Captura de datos estáticos

Los datos estáticos se encuentran almacenados de forma perdurable, es decir, a largo plazo. Por lo tanto, la captura de estos datos es similar al proceso de captura que se ha venido haciendo en los procesos de minería «tradicionales».

El objetivo principal es acceder a estos datos y traerlos a los sistemas de almacenamiento de nuestro entorno analítico que, como veremos más adelante, pueden ser sistemas de ficheros distribuidos o bases de datos NoSQL, en el caso de datos masivos.

Los datos estáticos pueden presentarse en su origen, principalmente, de dos formas:

- Datos contenidos en ficheros no estructurados (por ejemplo, ficheros de texto), semiestructurados (como por ejemplo, ficheros XML) o estructurados (como por ejemplo, ficheros separados por coma u hojas de cálculo).

- Datos contenidos en bases de datos, ya sean de tipo relacional (como por ejemplo, MySQL o PostgreSQL) o de tipo NoSQL (Cassandra o neo4J, entre muchos otros).

A partir de una conexión con la fuente indicada, ya sea un fichero o una base de datos, se procederá a una lectura de estos de forma secuencial, integrando los datos procesados en nuestro almacén analítico para su posterior análisis.

Existen múltiples herramientas que facilitan este tipo de tareas, algunas de específicas y otras que engloban todo el proceso de extracción, transformación y carga de datos (*extract, transform and load*, ETL). Además, existen herramientas específicas para lidiar con datos masivos, mientras que existen otras que, aunque pueden gestionar volúmenes relativamente grandes de datos, no están diseñadas específicamente para trabajar en entornos de *big data*.

Un ejemplo de herramienta completa de ETL puede ser la suite Data Integration[1] de Pentaho, que permite realizar las conexiones a distintas fuentes de datos y crear el *pipeline* de transformación de los datos, en caso de ser necesario.

Por otro lado, podemos encontrar herramientas específicas para entornos de datos masivos como, por ejemplo, Apache Sqoop, que está especializada en transferir datos masivos desde orígenes estructurados hacia los sistemas de almacenamiento de Apache Hadoop.

7.2.1. Caso de uso con Apache Sqoop

Apache Sqoop[2] (SQL-to-Hadoop) es una herramienta diseñada para transferir datos entre una base de datos relacional, como por ejemplo MySQL u Oracle, hacia un almacén de datos Hadoop, incluidos HDFS, Hive y HBase, y viceversa. Automatiza la mayor parte del proceso de transferencia de datos, leyendo la información del esquema directamente des-

[1] http://www.pentaho.com/product/data-integration
[2] http://sqoop.apache.org/

de el sistema gestor de la base de datos. Sqoop luego usa MapReduce para importar y exportar los datos hacia y desde Hadoop.

Supongamos que disponemos de una base de datos MySQL trabajando en el puerto 3306 y con una base de datos llamada «ejemplo1» que contiene la tabla «sampledata» con la información que nos interesa transferir a Hadoop. El siguiente fragmento de código muestra la orden que ejecuta Sqoop, indicándole la cadena de conexión a la base de datos MySQL, que es el origen de datos. El parámetro opcional `-m 1` indica que este trabajo debe usar una única tarea de `Map`.

```
1 /srv/sqoop
```

En este ejemplo, hemos especificado que la importación debería usar una única tarea de `Map`, ya que nuestra tabla no contiene una clave principal, que es necesaria para dividir y fusionar múltiples tareas de `Map`. Al utilizar una única tarea de `Map`, deberíamos esperar un solo archivo en el sistema HDFS, tal y como podemos ver:

```
1 /srv/sqoop
```

La instrucción anterior muestra las primeras filas del fichero «part-m-00000», donde podemos ver las diez primeras filas de los datos importados de la base de datos MySQL. El proceso de importación a HBase o Hive es similar al ejemplo que acabamos de ver.

7.2.2. Preprocesamiento de datos masivos

El propósito fundamental de la fase de preprocesamiento o preparación de datos es manipular y transformar los datos brutos (*raw data*) para que el contenido de información sea

coherente, homogénea y consistente. Este proceso es muy importante cuando se recolecta información de múltiples fuentes de datos.

Tareas de preprocesamiento de datos

El preprocesamiento de datos engloba a todas aquellas técnicas de análisis y manipulación de datos que permiten mejorar la calidad de un conjunto de datos, de modo que las técnicas de minería de datos que se aplicarán posteriormente puedan obtener mejor rendimiento.

En este sentido, podemos agrupar las distintas tareas de preprocesamiento en dos grandes grupos, cada uno de los cuales implica múltiples subtareas que dependen, en gran medida, del tipo de datos que estamos capturando e integrando en los sistemas de información analíticos. Estos son:

- Tareas de limpieza de datos (*data cleansing*).
- Tareas de transformación de datos (*data wrangling*).

Limpieza de datos

Las tareas de limpieza de datos agrupan los procesos de detectar y corregir (o eliminar) registros corruptos o imprecisos de un conjunto de datos. En concreto, se espera que estas tareas sean capaces de identificar partes incompletas, incorrectas, inexactas o irrelevantes de los datos, para luego reemplazar, modificar o eliminar aquellos que presenten problemas.

Después de la limpieza, un conjunto de datos debe ser coherente con otros conjuntos de datos similares en el sistema.

Las inconsistencias detectadas o eliminadas pueden haber sido causadas originalmente por errores en los datos originales (ya sean errores humanos o errores producidos en sensores u otros automatismos), por corrupción en la transmisión o almacenamiento, o por diferentes definiciones de diccionario de datos de las distintas fuentes de datos empleadas.

Una de las primeras tareas que debe realizar el proceso de limpieza es la **integración de datos** (*data integration*). El objetivo de esta tarea es resolver problemas de representación y codificación de los datos, con el fin de integrar correctamente datos provenientes de distintas plataformas para crear información homogénea. En este sentido, esta tarea es clave en entornos de datos masivos, donde generalmente recolectamos información desde múltiples fuentes de datos, que pueden presentar diversidad de representación y codificación de los datos.

Un ejemplo muy claro en la tarea de integración de datos se produce cuando recolectamos información sobre el momento en que se han producido ciertos eventos. Existen multitud de formatos para representar un momento concreto en el tiempo.

Por ejemplo, podemos representar el momento exacto en que se ha producido un evento de las siguientes formas:

- A partir de una cadena de texto que representa una fecha en un formato concreto, como puede ser «YYYY-MM-DD hh:mm:ss». En este formato se indica la fecha a partir del año, mes, día, hora, minutos y segundos. Teniendo en cuenta, además, la zona horaria.

- A partir del número de segundos desde la fecha de 1 de enero del año 1970, que es conocido como *Unix Timestamp*. Este sistema es muy empelado en los sistemas operativos Unix y derivados (por ejemplo, Linux) e

> identifica un momento de tiempo concreto a partir de un valor numérico. Por ejemplo, el valor 1502442623 equivale a la fecha 11 de agosto de 2017 a las 9 horas, 10 minutos y 23 segundos.

Por lo tanto, cuando integramos datos de distintas fuentes hay que homogeneizar las fechas para que los análisis posteriores se puedan ejecutar forma satisfactoria.

El proceso real de limpieza de datos puede implicar la eliminación de errores tipográficos o la validación y corrección de valores frente a una lista conocida de entidades. La validación puede ser estricta (como rechazar cualquier dirección que no tenga un código postal válido) o difusa (como corregir registros que coincidan parcialmente con los registros existentes conocidos).

Una práctica común de limpieza de datos es la mejora de datos, donde los datos se completan al agregar información relacionada. Por ejemplo, anexando direcciones con cualquier número de teléfono relacionado con esa dirección.

Por otro lado, la limpieza de datos propiamente dicha, incluye tareas «tradicionales» dentro del proceso de minería de datos, tales como la eliminación de valores extremos (*outliers*), la resolución de conflictos de datos, eliminación de ruido, valores perdidos o ausentes, etc.

Transformación de datos

La transformación de datos es el proceso de transformar y «mapear» datos «en bruto» en otro formato con la intención de hacerlo más apropiado y consistente para las posteriores operaciones de análisis.

Esto puede incluir la agregación de datos, así como muchos otros usos potenciales. A partir de los datos extraídos

en crudo (*raw data*) del origen de datos, estos se pueden formatear para posteriormente almacenarlos en estructuras de datos predefinidas o, por ejemplo, utilizar algoritmos simples de minería de datos o técnicas estadísticas para generar agregados o resúmenes que posteriormente serán almacenados en el repositorio analítico.

Dentro de las tareas de transformación de datos, se incluyen, entre otras, la consolidación de los datos de forma apropiada para la extracción de información, la sumarización de estos o las operaciones de agregación.

Otra tarea importante dentro de este grupo es la reducción de los datos (*data reduction*), que incluye tareas como la selección de un subconjunto de datos o parámetros relevante para los análisis posteriores mediante diferentes métodos para la reducción de datos, tales como la selección de características, selección de instancias o discretización de valores.

7.2.3. Modelo de preprocesamiento

Hay dos factores clave que determinarán la estructura o arquitectura del preprocesamiento de datos. Estos son:

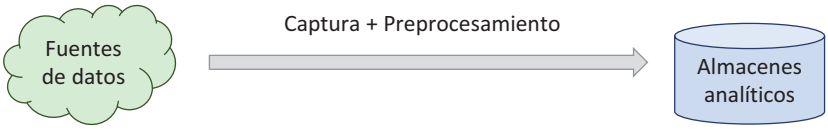
- La tipología del proceso de captura de los datos. En este sentido es importante determinar si el sistema trabajará con datos estáticos o datos dinámicos (en *streaming*).
- La complejidad de las operaciones de limpieza y transformación que se deberán aplicar a los datos antes de almacenarlos en el repositorio analítico.

En el caso de estudio de este capítulo (procesado de datos estáticos), los datos generalmente se encuentran almacenados de forma «estable» en un determinado almacén o fuente

de datos y se suele emplear un modelo de preprocesamiento durante el proceso de captura de los datos.

En este modelo, ejemplificado en la figura 1, los datos son limpiados y transformados durante el proceso de captura y se almacenan una vez se ha concluido esta tarea. Podemos decir que los procesos de captura y preprocesamiento se realizan en serie, pero sin requerir de un sistema de almacenamiento intermedio.

Figura 1. Esquema de una arquitectura sin *staging area*



Fuente: elaboración propia

Aunque el preprocesamiento de datos requiera de cierta complejidad, podemos ir realizando las operaciones necesarias de forma iterativa, ya que los datos de origen se encuentran almacenados de forma indefinida.

Capítulo 8

Almacenamiento de datos estructurados

Los modelos de almacenamiento de datos en entornos de datos masivos y sus herramientas asociadas han ido evolucionando a medida que los requisitos eran más elevados tanto en capacidad como rendimiento. Desde las bases de datos monolíticas a los ya más modernos almacenes de datos (*datawarehouse*), con importantes mejoras de rendimiento, bases de datos con gran capacidad de paralelismo y escalabilidad llegando a los sistemas de ficheros distribuidos. Así, en este capítulo nos centraremos en estos últimos, ya que son actualmente uno de los sistemas más utilizados para el diseño y despliegue de grandes repositorios de datos (comúnmente llamados *data lakes* en entornos de datos masivos o *big data*).

Es este capítulo hablaremos de los sistemas de ficheros distribuidos y veremos su arquitectura y funcionalidades básicas. Aunque existen múltiples sistemas de ficheros distribuidos, en este material nos centraremos en el sistema de ficheros distribuidos de Hadoop (Hadoop Distributed File System, HDFS).

8.1. Almacenamiento de datos masivos

Muchos escenarios que requieren del manejo de grandes volúmenes de datos obligan a las empresas e instituciones a adoptar soluciones de almacenaje capaces de almacenar una gran cantidad de datos a la vez que ofrecen altas prestaciones, escalabilidad y fiabilidad frente a cualquier problema y/o fallo.

Una vez los datos han sido capturados y preprocesados es necesario almacenarlos para su posterior análisis. Incluso en los sistemas basados en tiempo real (*streaming*), se suele almacenar una copia de los datos de forma paralela a su procesamiento, ya que en caso contrario se perderían una vez procesados.

En este sentido, el almacenamiento de datos masivos presenta dos opciones principales:

- Sistemas de ficheros distribuidos.
- Bases de datos, principalmente de tipo NoSQL.

Los sistemas de ficheros distribuidos se basan en el funcionamiento de los sistemas de ficheros tradicionales, gestionados por el sistema operativo, pero en este caso se almacenan los datos de forma transversal en un conjunto (pudiendo ser heterogéneo) de computadoras. En el capítulo 8.2 profundizaremos en los sistemas de ficheros distribuidos, haciendo hincapié en HDFS, uno de los sistemas más conocidos y empleados en la actualidad.

Por otro lado, las bases de datos NoSQL son una opción de almacenamiento masivo y distribuido muy empleado por cualquier empresa o institución que trabaje con *big data*. Veremos una introducción a este tipo de bases de datos, viendo los

distintos tipos y sus principales características en el capítulo 8.3.

8.2. Sistemas de ficheros distribuidos

Podemos identificar, a grandes rasgos, dos grandes sistemas de almacenamiento de altas prestaciones.

En primer lugar, existen los sistemas de acceso compartido a disco o *shared-disk file system*, tales como *storage-area network* (SAN). Dicho sistema permite que varios ordenadores puedan acceder de forma directa al disco a nivel de bloque de datos, siendo el nodo cliente quien traduce los bloques de datos a ficheros.

Los **metadatos** son datos que describen otros datos. En general, un grupo de metadatos se refiere a un grupo de datos que describen el contenido informativo de un objeto al que se denomina recurso.

Hay diferentes aproximaciones arquitectónicas a un sistema de archivos de disco compartido. Algunos distribuyen información de archivos en todos los servidores de un *cluster* (totalmente distribuidos). Otros utilizan un servidor de metadatos centralizado, pero en ambos casos se alcanza el mismo resultado: permitir que todos los servidores tengan acceso a los datos en un dispositivo de almacenaje compartido.

El principal problema de estos sistemas es que su modelo de almacenamiento requiere un subsistema de disco externo relativamente caro (por ejemplo, Fibre Channel / iSCSI) además de conmutadores, adaptadores, etc. Sin embargo, permite que los fallos de disco sean manejados en el subsistema externo.

Otra aproximación, mucho más utilizada actualmente, es un sistema de almacenamiento distribuido o sistemas de archivos distribuidos, donde los nodos no comparten el acceso a datos a nivel de bloque, pero proporcionan una vista unificada, con un espacio de nombres (*namespace*) global. La diferencia reside en el modelo utilizado para el almacenamiento de bloques subyacente. A diferencia del modelo anterior, cada nodo tiene su propio almacenamiento a nivel de bloque de datos. La abstracción de dichos bloques en los propios ficheros se gestiona a un nivel superior.

Estos sistemas usualmente se construyen usando *commodity hardware* o *hardware* estándar, de menor coste (como discos SATA/SAS). Su escalabilidad es superior a los sistemas de disco compartido y, aunque puede ofrecer peores latencias, existen estrategias para compensar dicho problema, como el uso extensivo de caches, replicación, etc.

En este capítulo nos centraremos en el sistema de archivos distribuido más popular actualmente, conocido como HDFS.

8.2.1. ¿Qué es HDFS?

El Sistema de Archivos Distribuidos Hadoop (en adelante HDFS) es un subproyecto de Apache Hadoop[1] y forma parte del ecosistema de herramientas *big data* que proporciona Hadoop (Dunning, Friedman, 2015).

HDFS es un sistema de archivos altamente tolerante a fallos diseñado para funcionar con el llamado «hardware de bajo coste» (*commodity hardware*),[2] lo que permite reducir los costes de almacenamiento significativamente. Proporciona acceso

[1]https://hadoop.apache.org/

[2]La computación de «bajo coste» implica el uso de un gran número de componentes de computación ya disponibles para la computación

de alto rendimiento a grandes volúmenes de datos y es adecuado para aplicaciones de procesamiento distribuido, tal y como su nombre indica.

Tiene muchas similitudes con otros sistemas de archivos distribuidos, pero a su vez presenta diferencias importantes:

- Sigue un modelo de acceso *write once read many*. Esto es, restringiendo rigurosamente la escritura de datos a un escritor o *writer* se relajan los requisitos de control de concurrencia. Además, un archivo una vez creado, escrito y cerrado no necesita ser cambiado, lo que simplifica la coherencia de los datos y permite un acceso de alto rendimiento. Los bytes siempre se anexan al final de un flujo, y los flujos de bytes están garantizados para ser almacenados en el orden escrito.

- Otra propiedad interesante de HDFS es el desplazamiento de la capacidad de computación a los datos en lugar de mover los datos al espacio de la aplicación. Así, un cálculo solicitado por una aplicación es mucho más eficiente si se ejecuta cerca de los datos en los que opera. Esto es especialmente cierto cuando el tamaño del conjunto de datos es enorme. Así se minimiza la congestión de la red y aumenta el rendimiento global del sistema. HDFS proporciona interfaces para que las aplicaciones se muevan más cerca de donde se encuentran los datos.

paralela, para obtener la mayor cantidad de computación útil a bajo coste.

Entre las principales características de HDFS podemos citar:

- Es tolerante a fallos, detectándolos y aplicando técnicas de recuperación rápida y automática. Los fallos en *hardware* suelen ocurrir con cierta frecuencia y teniendo en cuenta que una instancia de HDFS puede consistir en cientos o miles de máquinas, cada una almacenando parte de los datos del sistema de archivos, existe una elevada probabilidad de que no todos los nodos funcionen correctamente en todo momento. Por lo tanto, la detección de fallos y recuperación rápida y automática de ellos es uno de los objetivos clave de HDFS, define su arquitectura y dispone además de mecanismos de reemplazo en caso de fallos graves del sistema HDFS (*hot swap*), como se describirá más adelante. Además es un sistema fiable gracias al mantenimiento automático de múltiples copias de datos para maximizar la integridad.

- Acceso a datos a través de Streaming MapReduce. Las aplicaciones que se ejecutan en HDFS necesitan tener acceso de *streaming* a sus conjuntos de datos, ya que son múltiples los escenarios y las herramientas *big data* que ofrecen procesamiento en *streaming* sobre datos en HDFS. Y, asimismo, debe estar perfectamente integrado en los sistemas de procesado *batch* como, por ejemplo, MapReduce. Así, HDFS no ofrece un buen rendimiento para trabajar de forma interactiva ya que está diseñado para dar alto rendimiento de acceso a grandes volúmenes de datos en lugar de baja latencia de acceso a estos.

- HDFS ha sido diseñado para ser fácilmente portable de una plataforma a otra, lo que lo hace compatible en en-

tornos de *hardware* heterogéneo (*commodity hardware*) y diferentes sistemas operativos. HDFS está implementado utilizando lenguaje Java, de modo que cualquier máquina que admita la programación Java puede ejecutar HDFS. Esto facilita la adopción generalizada de HDFS como una plataforma de elección para un gran conjunto de aplicaciones. Además este factor incrementa su escalabilidad, ya que la integración de un nuevo nodo en un *cluster* HDFS impone pocas restricciones.

- Las aplicaciones que se ejecutan sobre datos en HDFS suelen trabajar con grandes conjuntos de datos, del orden de gigabytes a terabytes o más. HDFS está diseñado para trabajar sobre tamaños de archivo grandes, optimizando todos los parámetros relacionados con el rendimiento, como el movimiento de datos entre nodos y entre los propios *racks*, maximizando la localización de la computación allí donde se encuentran los datos.

- Proporciona diversos interfaces para que las aplicaciones trabajen sobre los datos almacenados, maximizando la facilidad de uso y acceso.

8.2.2. Arquitectura

HDFS no es un sistema de ficheros regular como podría ser ext3[3] o NTFS,[4] que están construidos sobre el *kernel* del sistema operativo, sino que es una abstracción sobre del sistema de ficheros regular. HDFS es un servicio que se ejecuta en un entono distribuido de ordenadores o *cluster* siguiendo una arquitectura de maestro (al que llamaremos NameNode

[3]https://en.wikipedia.org/wiki/Ext3
[4]https://en.wikipedia.org/wiki/NTFS

en adelante) y esclavo (DataNode en adelante). En un *cluster* suele haber un solo NameNode y varios DataNodes, uno por ordenador en el *cluster*.

NameNode y DataNode

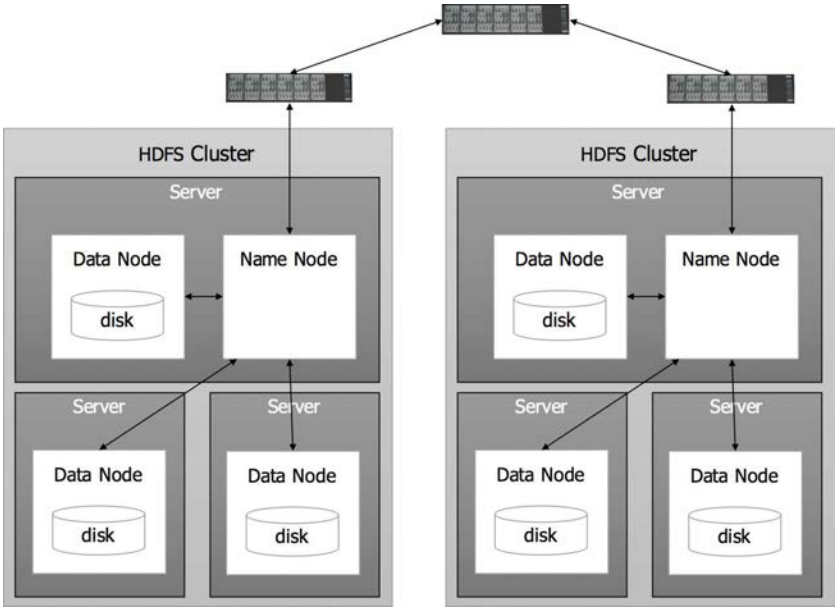
El NameNode gestiona las operaciones del *namespace* del sistema de archivos como abrir, cerrar y renombrar archivos y directorios y regula el acceso del cliente a los archivos. También es responsable de la asignación de los datos a su correspondiente DataNode.

Los DataNode manejan las solicitudes de lectura y escritura de los clientes del HDFS y son responsables de gestionar el almacenamiento del nodo en el que residen, así como crear, eliminar y duplicar los datos de acuerdo con las instrucciones del NameNode.

Los NameNodes y los DataNodes son componentes de *software* diseñados para ejecutarse de una manera desacoplada entre ellos en los nodos del *cluster*. Tal y como muestra la figura 1, la arquitectura típica consta de un nodo en el que se ejecuta el servicio NameNode y posiblemente el servicio DataNode, mientras que cada una de las otras máquinas del *cluster* se ejecuta el DataNode.

Los DataNodes consultan continuamente al NameNode nuevas instrucciones. Sin embargo, el NameNode no se conecta al DataNode directamente, simplemente responde a dichas consultas. Al mismo tiempo, el DataNode mantiene un canal de comunicaciones abierto de modo que el código cliente u otros DataNodes pueden escribir y/o leer datos cuando sean requeridos en otros nodos.

Figura 1. Arquitectura típica de un *cluster* HDFS



Fuente: elaboración propia

En computación distribuida, la **llamada a un procedimiento remoto** (*remote procedure call*, RPC) es un programa que utiliza una computadora para ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambas.

Todos los procesos de comunicación HDFS se basan en el protocolo TCP/IP. Los clientes HDFS se conectan a un puerto TCP (protocolo de control de transferencia) abierto en el NameNode y se comunican con él utilizando un protocolo basado en RPC (*remote procedure call*). A su vez, los DataNodes se comunican con el NameNode usando un protocolo propietario.

Espacio de nombres (*namespace*)

HDFS admite una organización de archivos jerárquica (árboles de directorios) similar a la mayoría de los sistemas de archivos existentes. Un usuario o una aplicación pueden crear directorios y almacenar archivos dentro de estos directorios, y pueden crear, eliminar y mover archivos de un directorio a otro. Aunque la arquitectura de HDFS no lo impide, todavía no es posible crear enlaces simbólicos (*symbolic links*). El NameNode es el responsable de mantener el espacio de nombres del sistema de archivos y cualquier cambio en dicho espacio es registrado.

Organización de datos

El rendimiento de HDFS es mejor cuando los ficheros que hay que gestionar son grandes. Se divide cada archivo en bloques de datos, típicamente de 64 MB (totalmente configurable), por lo tanto, cada archivo consta de uno o más bloques de 64 MB. HDFS intenta distribuir cada bloque en DataNodes separados.

Cada bloque requiere del orden de 150 B de *overhead* en el NameNode. Así, si el número de ficheros es muy elevado (pueden llegar a ser millones) es posible llegar a saturar la memoria del NameNode.

Los archivos de datos se dividen en bloques y se distribuyen a través del *cluster* en el momento de ser cargados. Cada fichero se divide en bloques (del tamaño de bloque definido) que se distribuyen a través del *cluster*.

Al distribuir los datos entre diferentes máquinas del *cluster* se elimina el *single point of failure* (SPOF) o único punto de fallo. En el caso de fallo de uno de los nodos no se pierde un

Cuadro 8.1. Impacto en cuanto a la gestión de memoria del NameNode en la gestión de un solo fichero de 1 GB de tamaño frente a particionar 1 GB en mil ficheros de 1 MB

Estrategia	Metadatos	Recursos
1 fichero de 1 GB	1 entrada en el registro de nombres, 16 bloques de datos, 3 réplicas	49 entradas
1.000 ficheros de 1 MB	1.000 entradas en el registro de nombres, 1.000 bloques de datos, 3 réplicas	4.000 entradas

fichero ya que se puede recomponer a partir de sus réplicas. Esta característica también facilita el procesado en paralelo, ya que diferentes nodos pueden procesar bloques de datos individuales de forma concurrente.

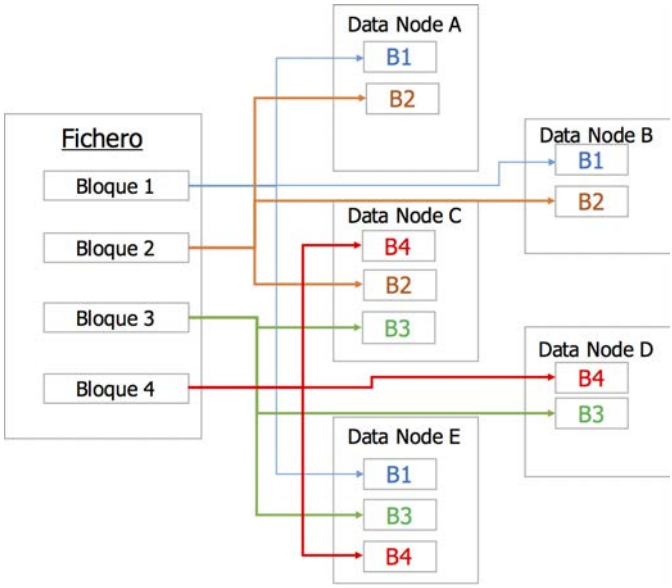
La figura 2 muestra a modo de ejemplo como un fichero dado se divide en cuatro bloques (B1, B2, B3 y B4) que, en el momento de ser cargado en el HDFS, se reparte entre los diferentes nodos. Se almacenan hasta tres copias de cada bloque debido a la replicación de datos (que se describirá en secciones posteriores).

Como ya se ha introducido anteriormente, el tamaño del bloque suele ser de 64 MB o 128 MB, aunque es altamente configurable a través de los parámetros de configuración, que se encuentran en el fichero «hdfs-site.xml».

A continuación se muestra un ejemplo de configuración de dicho valor:

```
<property>
<name>dfs.blocksize</name>
<value>134217728</value>
</property>
```

Figura 2. Distribución de bloques de datos en un *cluster* HDFS entre los diferentes DataNodes. Nótese que se crean tantas copias como réplicas tenga el sistema definidas



Fuente: elaboración propia

Así, trabajar con ficheros grandes supone varias ventajas:

- Mejor rendimiento del *cluster* mediante el *streaming* de grandes cantidades de datos.
- Mejor rendimiento de *cluster* evitando costosas cargas de arranque para cantidades de datos pequeñas.
- Mejor escalabilidad del NameNode asociada con la carga de metadatos de menos ficheros pero de mayor tamaño. Si el NameNode trabaja con pequeños ficheros su operativa es más costosa y puede conducir a problemas de memoria.

- Mayor disponibilidad, ya que el estado del sistema de archivos HDFS está contenido en memoria.

Replicación de datos

Como se ha indicado anteriormente, HDFS es tolerante a fallos mediante la replicación de bloques de datos. Una aplicación puede especificar el número de réplicas de un archivo en el momento en que se crea, y este número se puede cambiar posteriormente. El NameNode es también el responsable de gestionar la replicación de datos.

HDFS utiliza un modelo de asignación de bloques replicados a diferentes nodos pero también puede distribuir bloques a diferentes *racks* de ordenadores (la llamada *rack-aware replica placement policy*), lo que lo hace muy competitivo frente a otras soluciones de almacenamiento distribuido, ya no solo frente a problemas de fiabilidad sino también de eficiencia en el uso de la interconectividad de la red interna del *cluster*.

En efecto, los grandes entornos HDFS normalmente operan a través de múltiples instalaciones de computadoras y la comunicación entre dos nodos de datos en instalaciones diferentes suele ser más lenta que en la misma instalación. Por lo tanto, el NameNode intenta optimizar las comunicaciones entre los DataNodes identificando su ubicación por su «rack ID» o identificador de *rack*. Así, HDFS es consciente de la ubicación de cada DataNode, lo que le permite optimizar la tolerancia a fallos (DataNodes con replicas en *racks* distintos) y balancear también el tráfico de red entre *racks* distintos.

El factor de replicación se define en el fichero de configuración «hdfs-site.xml», y que ejemplificamos a continuación:

```
1 <property>
2 <name>dfs.replication</name>
```

```
3 <value>3</value>
4 </property>
```

Imaginemos el caso en el que un usuario quiere almacenar dos ficheros («file1.log» y «file2.log») en el sistema HDFS de un *cluster* de 5 nodos (A, B, C, D y E).

HDFS divide cada fichero en bloques de datos. Dado el tamaño de los ficheros ejemplo vamos a suponer que «file1.log» se divide en tres bloques, B1, B2 y B3. Mientras que «file2.log» se divide en dos bloques más (B4 y B5). Los bloques se distribuyen entre los diferentes nodos y además, vamos a suponer un factor de replicación 3, de modo que cada uno de los bloques se hacen tres copias. Así, el bloque B1 se almacena en los nodos A, B y D; el bloque B2 en los bloques B, D y E. La distribución completa puede verse a continuación:

Correspondencia entre fichero y bloques:

- «file1.log»: B1, B2 y B3
- «file2.log»: B4 y B5

Correspondencia entre bloques y nodos:

- B1: A, B y D
- B2: B, D y E
- B3: A, B y C
- B4: E, B y A
- B5: C, E y D

El NameNode es quien almacena toda la información relativa a los ficheros, sus correspondientes bloques y su localización en el *cluster*. La correspondencia o mapeo (*mapping*) de cada fichero con sus respectivos bloques está disponible en memoria y también en disco para su recuperación en caso de problemas, sin embargo, el mapeo entre bloques de datos y el nodo en el que están almacenados solo están disponibles en memoria, siendo los DataNodes quienes notifican al NameNode los bloques bajo su supervisión en un proceso llamado *heartbeating*, que se describe más adelante, y que suele tener un valor de unos 10 segundos, típicamente).

En el caso de tener que acceder a los ficheros anteriormente descritos, supongamos que el cliente pide al NameNode el fichero «file2.log» y este le responde con la lista de los bloques de datos que lo componen (B4 y B5 en este caso). Seguidamente el cliente le pregunta dónde encontrarlos y el NameNode le responde con los nodos en que se encuentran.

Que serían para cada bloque:

- B4: Nodos A, B y E
- B5: Nodos C, E y D

El NameNode devuelve al cliente una lista con los nodos ordenados por proximidad al él siguiendo el siguiente criterio:

1. Está en la misma máquina.
2. Está en el mismo *rack* (*rack awareness*).
3. Se encuentra en alguno de los nodos restantes.

El cliente obtiene el archivo solicitando cada uno de los bloques directamente desde los nodos en los que están almacenados y siguiendo los criterios de proximidad ya mencionados. Así, el cliente intentará recuperar el bloque desde el primer nodo de la lista (el nodo más cercano a él tal y como el NameNode le ha reportado) y si este nodo no está disponible, el cliente lo intentará desde el segundo, y si tampoco está disponible, lo intentará con el tercero, etc. Los datos se transfieren directamente entre el DataNode y el cliente, sin involucrar al NameNode, así, la comunicación entre el cliente y el NameNode es mínima, con un tráfico de red bajo, y todo el proceso optimiza en lo posible el uso de ancho de banda de la red de comunicaciones.

8.2.3. Fiabilidad

La fiabilidad es uno de los objetivos importantes de un sistema de archivos como HDFS. Así, en primer lugar es necesario detectar los problemas ya sea en los NameNode, los DataNodes o errores propios de la red.

Disponibilidad (Secondary NameNode)

Como se puede ver, el servicio NameNode debe estar activo de forma continua. Ya que si se detiene, el *cluster* estará inaccesible. ¿Qué medidas se toman para evitar que esto ocurra? HDFS dispone de dos modos de funcionamiento:

- Modo de alta disponibilidad.
- Modo clásico.

En el **Modo de alta disponibilidad** hay un NameNode activo, llamado principal o primario, y un segundo NameNode en

modo *standby*, llamado *mirror*, el cual toma el relevo del NameNode principal en caso de fallo, en un procedimiento llamado *hot swap*.

En el modo clásico hay un NameNode principal o primario y un NameNode secundario que tiene funciones de mantenimiento y optimización, pero no de *backup*. El NameNode secundario es un servicio que se ejecuta en modo *daemon* (es decir, un demonio siempre activo) que se encarga de algunas tareas de mantenimiento para el NameNode tales como:

- Mantiene una imagen del mapeo de ficheros con sus respectivos bloques de datos y su ubicación en los DataNodes (llamado *fsimage*).
- Los cambios en el sistema de archivos no se integran inmediatamente en el archivo de imagen, en su lugar, el archivo de imagen principal se actualiza en modo *lazy* (es decir, solo bajo demanda).
- Los cambios en el sistema de archivos se graban en unos archivos separados conocidos como *edit logs*, análogos a un punto de control o *breakpoints* en un registro de transacciones de una base de datos relacional. En el *edit log* se registra de forma persistente cada transacción que se aplica a los metadatos del sistema de archivos HDFS. Si los archivos *edit log* o *fsimage* se dañan, la instancia de HDFS a la que pertenecen deja de funcionar. Por lo tanto, un NameNode admite múltiples copias de los archivos *fsimage* y *edit log*, y cualquier cambio en cualquiera de estos archivos se propaga de forma síncrona a todas las copias. Cuando un NameNode se reinicia, utiliza la última versión consistente de *fsimage* y *edit log* para inicializarse. De forma periódica se fusionan los

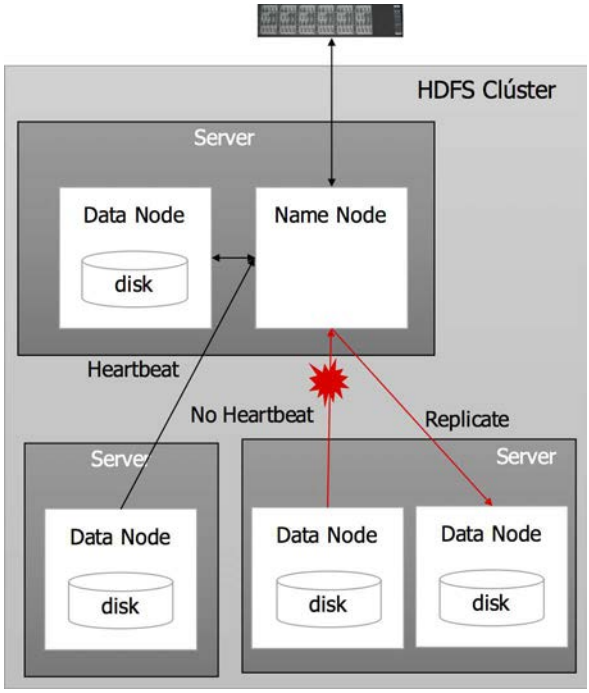
edit log en el archivo de imagen principal *fsimage*, típicamente una vez por hora, lo que acelera el tiempo de recuperación si falla el NameNode. Cuando se inicializa un NameNode, lee el archivo *fsimage* junto con los *edit logs* y aplica las transacciones y la información de estado que se encuentran registradas en dichos archivos.

- No obstante, el NameNode secundario no es un nodo de *backup* que reemplaza al NameNode en caso de fallo, de modo que en escenarios de alta disponibilidad no se suele usar.

HDFS *heartbeats*

Existen multitud de situaciones que pueden causar pérdida de conectividad entre el NameNode y los DataNodes, así, para detectar dichos problemas, HDFS utiliza los *heartbeats* (pulsaciones) que verifican la conectividad entre ambos. Cada DataNode envía mensajes de forma periódica (los *heartbeats*) a su NameNode y este puede detectar la pérdida de conectividad si deja de recibirlos. El NameNode marca como DataNodes caídos a aquellos que no responden a los *heartbeats* y para de enviarles más solicitudes. Los datos almacenados en un nodo caído ya no están disponibles para un cliente HDFS desde ese nodo, que se elimina efectivamente del sistema. Si la caída de un nodo hace que el factor de replicación de los bloques de datos caiga por debajo de su valor mínimo, el NameNode inicia el proceso de creación de una replica adicional para devolver el factor de replicación a un estado normal. Este proceso se ilustra en la figura 3.

Figura 3. Diagrama de replicación debido a fallo en el *heartbeat*



Fuente: elaboración propia

Reequilibrado de bloques de datos

Los bloques de datos HDFS no siempre se pueden colocar uniformemente entre los DataNodes, lo que significa que el espacio que se usa para uno o más de dichos nodos puede estar infrautilizado. Para evitar que esto ocurra, HDFS realiza tareas de equilibrado de bloques entre nodos utilizando varios modelos:

- Un modelo puede mover bloques de datos de un DataNode a otro automáticamente si el espacio libre en dicho nodo cae demasiado abajo.

- Otro modelo consiste en crear dinámicamente réplicas adicionales y reequilibrar otros bloques de datos en un *cluster* si se produce una subida repentina en la demanda de un archivo dado.

Un motivo habitual para reequilibrar bloques es la adición de nuevos DataNodes a un *cluster*. Al colocar nuevos bloques, el NameNode sigue varios criterios para su asignación:

- Políticas de replicación, tal como se han descrito anteriormente.
- Uniformidad en la distribución de datos entre nodos.
- Reducción de uso de ancho de banda por el tráfico de datos en la red interna.

Así, el reequilibrado de bloques de datos del *cluster* de HDFS es un mecanismo que se utiliza para sostener la integridad de sus datos.

HDFS también utiliza mecanismos para validar los datos del sistema HDFS almacenando a su vez sumas de comprobación (o *chekcsums*)[5] en archivos separados y ocultos en el mismo espacio de nombres que los datos reales. Así, cuando un cliente recupera archivos del HDFS, puede verificar que los datos recibidos coincidan con la suma de comprobación almacenada en el archivo asociado.

[5]Es una función *hash* que tiene como propósito principal detectar cambios accidentales en una secuencia de datos para proteger la integridad de estos, verificando que no haya discrepancias entre los valores obtenidos al hacer una comprobación inicial y otra final tras la transmisión.

8.2.4. Interfaz HDFS

Para acceder al servicio HDFS disponemos de varias alternativas, aunque una de las más habituales es trabajar con la línea de comandos.

HDFS organiza los datos de cara al usuario en ficheros y directorios de forma parecida a los sistemas de ficheros más habituales, de modo que el usuario puede consultar el contenido de los directorios mediante comandos parecidos a los de los sistemas operativos de la familia Unix/Linux. Usando el prefijo «hdfs dfs» o «hadoop fs» antes de cada comando (son equivalentes).

A continuación mostraremos algunos comandos básicos de uso de HDFS. Sin embargo, para poder conocer todas las opciones disponibles, basta con invocar al siguiente comando:

```
1 > hdfs dfs -help
```

Para poder consultar el contenido de los diferentes directorios del sistema de ficheros usaremos el comando «ls» que lista los contenidos de los directorios. Una importante diferencia con el sistema de ficheros Unix es que no existe la opción «cd», con lo que no es posible movernos por el sistema de ficheros. El siguiente comando:

```
1 > hdfs dfs -ls
```

Muestra los contenidos del directorio del usuario en el sistema (que llamaremos *user* en estos ejemplos). Así, un comando equivalente sería:

```
1 > hdfs dfs -ls /home/user
```

Para listar los contenidos de un directorio dado llamado <directorio>, usaremos el comando:

```
> hdfs dfs -ls <directorio>
```

Para crear un directorio, usaremos el comando:

```
> hdfs dfs -mkdir <nombre-directorio>
```

Es importante remarcar que no se puede crear más de un directorio a la vez. Si queremos crear un directorio dentro de otro directorio que no existe, hay que crear el primero antes de crear el segundo:

```
> hdfs dfs -mkdir /directorio1/directorio2
```

Si <directorio1> no existe, habrá que crearlo primero para poder crear el segundo dentro de este:

```
> hdfs dfs -mkdir /directorio1
> hdfs dfs -mkdir /directorio1/directorio2
```

De forma similar a otros sistemas de ficheros, se pueden dar o restringir permisos de acceso y lectura de directorios y ficheros usando el comando «chmod».

```
> hdfs dfs -chmod rwx /directorio
```

Donde <rwx> puede ser especificado en formato numérico (por ejemplo, si <rwx> es igual a 777, se darán permisos de lectura, escritura y ejecución a todos los usuarios del sistema). Es importante tener en cuenta que para acceder a directorios en HDFS el directorio debe tener permisos de ejecución «x», como se ha indicado en secciones anteriores.

Una vez creado un directorio, debemos ser capaces de cargar archivos del sistema de ficheros local a los directorios dentro HDFS. Para tal fin usaremos el comando «put».

```
> hdfs dfs -put <fichero-origen> <directorio-destino-hdfs>
```

Para recuperar archivos del sistema HDFS al sistema de ficheros local usaremos el comando «get».

```
> hdfs dfs -get <directorio-origen-hdfs> <directorio-destino>
```

También en posible copiar y mover directorios mediante el comando «cp» para copiar y el comando «mv» para mover y renombrar ficheros.

```
> hdfs dfs -cp <directorio-origen-hdfs> <directorio-destino-
    hdfs>
> hdfs dfs -mv <directorio-origen-hdfs> <directorio-destino-
    hdfs>
```

Trabajar sobre el sistema HDFS a través de la línea de comandos es muy habitual por su similitud al funcionamiento de la línea de comandos de Unix. Sin embargo el ecosistema Hadoop ofrece también un conjunto de herramientas que de algún modo permiten trabajar sobre el sistema de ficheros HDFS ofreciendo funcionalidades diversas como Sqoop, ya presentado anteriormente u otras herramientas como el propio monitor vía web de Hadoop o (Hue).[6]

8.3. 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[7](*atomicity, consistency, isolation, durability*). Estas propiedades ACID permiten garantizar que los datos

[6]http://gethue.com/

[7]http://es.wikipedia.org/wiki/ACID

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 de 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 seria Twitter, donde millones de usuarios acceden al servicio de forma concurrentes 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, 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 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 como almacenan la información:

- **Clave-valor**. Este formato es el más típico. Podemos entenderlo como un HashMap donde cada elemento esta 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 *cluster*. 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 archi-

vos 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.

- **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 *cluster* y puede realizar operaciones sobre los documentos.

- **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.

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

Capítulo 9

Análisis de datos estáticos

En este capítulo veremos las principales técnicas para el análisis de grandes volúmenes de datos estáticos. Dicho de otra forma, nos centraremos en ver el funcionamiento de los principales *frameworks* desarrollados para el procesamiento por lotes (*batch*). En este sentido, empezaremos viendo en detalle el modelo de funcionamiento de MapReduce, que durante muchos años lideró las plataformas para el procesamiento de grandes volúmenes de datos. A partir de aquí veremos sus principales limitaciones, y cómo estas propiciaron la aparición de Apache Spark, que se ha convertido en el estándar *de facto* en la industria.

9.1. Apache Hadoop y MapReduce

MapReduce es una metodología de procesado de datos distribuidos teniendo un modelo de programación para dar soporte a la computación paralela sobre grandes conjuntos de datos en *cluster* de computadoras (Bengfort, Kim, 2016). El

nombre MapReduce está inspirado en los nombres de dos procesos o funciones esenciales del algoritmo: `Map` y `Reduce`.

9.1.1. Abstracción

Como ya hemos indicado, las dos tareas principales que realiza este modelo son:

- `Map`: esta tarea es la encargada de «etiquetar» o «clasificar» los datos que se leen desde el disco, típicamente de HDFS, en función del procesamiento que estemos realizando.
- `Reduce`: esta tarea es la responsable de agregar los datos etiquetados por la tarea `Map`. Puede dividirse en dos etapas, la `Shuffle` y el propio `Reduce` o agregado.

Todo el intercambio de datos entre tareas utiliza estructuras llamadas parejas <clave, valor>(<key, value> en inglés) o tuplas.

En el siguiente ejemplo vamos a mostrar, desde un punto de vista funcional, en que consiste una tarea MapReduce: Imaginemos que tenemos tres ficheros con los siguientes datos:

Fichero 1:

```
1 Carlos, 31 anos, Barcelona
2 Maria, 33 anos, Madrid
3 Carmen, 26 anos, Coruna
```

Fichero 2:

```
1 Juan, 12 anos, Barcelona
2 Carmen, 35 anos, Madrid
3 Jose, 42 anos, Barcelona
```

Fichero 3:

```
Maria, 78 anos, Sevilla
Juan, 50 anos, Barcelona
Sergio, 33 anos, Madrid
```

Dados estos ficheros, podríamos preguntarnos ¿Cuántas personas hay en cada ciudad?

Para responder esa pregunta definiremos una tarea `Map` que leerá las filas de cada fichero y las «etiquetará» cada una en función de la ciudad que aparece, que es el tercer campo de cada línea, que sigue la estructura: nombre, edad, ciudad.

Para cada línea, nos devolverá una tupla de la forma: <ciudad, cantidad>.

Al final de la ejecución de la tarea `Map` en cada fichero tendremos:

- Fichero 1: (Barcelona, 1), (Madrid, 1), (Coruna, 1)
- Fichero 2: (Barcelona, 1), (Madrid, 1), (Barcelona, 1)
- Fichero 3: (Sevilla, 1), (Barcelona, 1), (Madrid, 1)

Como vemos, nuestras tuplas están formadas por una clave (*key*), que es el nombre de la ciudad, y un valor (*value*) el número de veces que aparece en la línea, que siempre es 1 en el caso de la tarea `Map`, que no realiza ninguna agregación, simplemente etiqueta y transforma.

La tarea `Reduce` se ocupará de agrupar los resultados según el valor de la clave. Así, recorrerá todas las tuplas agregando los resultados por misma clave y devolviéndonos:

```
(Barcelona, 4)
(Sevilla,1)
(Madrid, 3)
(Coruna, 1)
```

Obviamente, en este ejemplo la operación no requería un entorno distribuido. Sin embargo, en un entorno con millones de registros de personas una operación de estas características podría ser muy efectiva.

Conectando con lo descrito en secciones anteriores, una operación MapReduce es un procesado en modo *batch*, ya que recorre de una vez todos los datos disponibles y no devuelve resultados hasta que ha finalizado.

Es importante tener en cuenta que las funciones de agregación (*reducer*) deben ser conmutativas y asociativas.

Implementación MapReduce en Hadoop

Desde un punto de vista de proceso en entorno Hadoop, vamos a describir cuales son los componentes de una tarea MapReduce:

1. Generalmente, un *cluster* está formado por varios nodos controlados por un nodo maestro. Cada nodo almacena ficheros localmente y son accesibles mediante el sistema de ficheros HDFS (típicamente es HDFS, aunque no es un requisito necesario). Los ficheros se distribuyen de forma homogénea en todos los nodos. La ejecución de un programa MapReduce implica la ejecución de las tareas de `Map` en muchos o todos los nodos del *cluster*. Cada una de estas tareas es equivalente, es decir, no hay tareas `Map` específicas o distintas a las otras. El objetivo es que cualquiera de dichas tareas pueda procesar cualquier fichero que exista en el *cluster*.

2. Cuando la fase de `Map` ha finalizado, los resultados intermedios (tuplas <clave, valor>) deben intercambiarse entre las máquinas para enviar todos los valores con la

misma clave a un solo `Reduce`. Las tareas `Reduce` también se ejecutan en los mismos nodos que las `Map`, siendo este el único intercambio de información entre tareas (ni las tareas `Map` ni las `Reduce` intercambian información entre ellas, ni el usuario puede interferir en este proceso de intercambio de información). Este es un componente importante de la fiabilidad de una tarea MapReduce, ya que si un nodo falla, reinicia sus tareas sin que el estado del procesamiento en otros nodos dependa de él.

Sin embargo, en el proceso de intercambio de información entre `Map` y `Reduce`, podemos introducir dos nuevos conceptos: partición (*partition*) y *shuffle*.

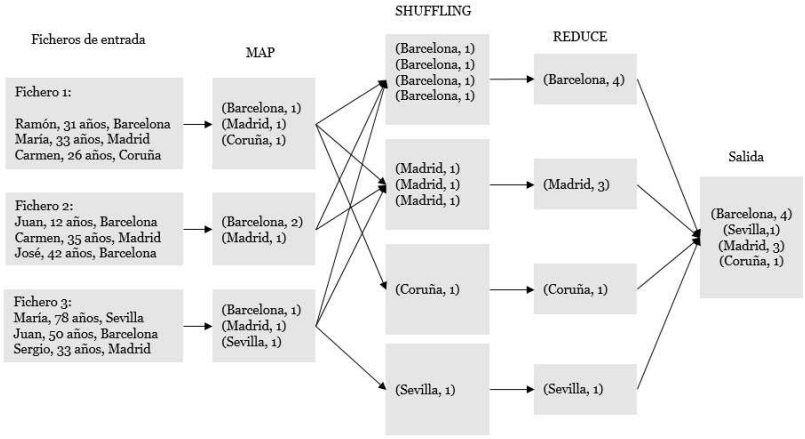
Cuando una tarea `Map` ha finalizado en algunos nodos, otros nodos todavía pueden estar realizando tareas `Map`. Sin embargo, el intercambio de datos intermedios ya se inicia y estos se mandan a la tarea `Reduce` correspondiente. El proceso de mover datos intermedios desde los `mapper` a los `reducers` se llama `Shuffle`. Se crean subconjuntos de datos, agrupados por <clave>, lo que da lugar a las particiones, que son las entradas a las tareas `Reduce`. Todos los valores para una misma clave son agregados o reducidos juntos, indistintamente de cual era su tarea `mapper`. Así, todos los `mapper` deben ponerse de acuerdo en donde mandar las diferentes piezas de datos intermedios.

Cuando el proceso `Reduce` ya se ha completado, agregando todos los datos, éstos son guardados de nuevo en disco. La figura 1 nos muestra de forma gráfica el ejemplo descrito.

Limitaciones de Hadoop

Hadoop es la implementación del paradigma MapReduce que solucionaba la problemática del cálculo distribuido utili-

Figura 1. Diagrama de flujo con las etapas `Map` y `Reduce` del ejemplo descrito



Fuente: elaboración propia

zando las arquitecturas predominantes hace una década. Sin embargo, actualmente presenta importantes limitaciones:

- Es complicado aplicar el paradigma MapReduce en muchos casos, ya que una tarea debe descomponerse en subtareas `Map` y `Reduce`. En algunos casos, no es fácil esta descomposición.
- Es lento. Una tarea puede requerir la ejecución de varias etapas MapReduce y, en ese caso, el intercambio de datos entre etapas se llevará a cabo utilizando ficheros, haciendo uso intensivo de lectura y escritura en disco.
- Hadoop es un *framework* esencialmente basado en Java que requiere la descomposición en tareas `Map` y `Reduce`. Sin embargo, esta aproximación al procesamiento de datos resultaba muy dificultosa para el científico de datos

sin conocimientos de Java. De modo que diversas herramientas han aparecido para flexibilizar y abstraer al científico del paradigma MapReduce y su programación en Java. Sin embargo, el científico de datos debe conocer dichas herramientas, cada una con propiedades distintas, lenguajes distintos y muy poca (o ninguna) compatibilidad entre ellas.

9.1.2. Ejemplo de aplicación de MapReduce

$$M_{2\times 3} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \ , \ N_{3\times 2} = \begin{bmatrix} 6 & 3 \\ 5 & 2 \\ 4 & 1 \end{bmatrix}$$

Tal y como se ha descrito con anterioridad, el proceso funciona de modo que cada elemento de la primera fila de la primera matriz M se multiplica por cada elemento de la primera columna N y así sucesivamente para el resto de filas y columnas de cada matriz. El resultado numérico del ejercicio es el siguiente:

$$\begin{aligned} M_{2\times 3} \cdot N_{3\times 2} = P_{2\times 2} = \\ = \begin{bmatrix} 1\cdot 6 + 2\cdot 5 + 3\cdot 4 & 1\cdot 3 + 2\cdot 2 + 3\cdot 1 \\ 4\cdot 6 + 5\cdot 5 + 6\cdot 4 & 4\cdot 3 + 5\cdot 2 + 6\cdot 1 \end{bmatrix} \end{aligned}$$

Es un proceso altamente paralelizable ya que cada cálculo es independiente del otro, tanto la multiplicación entre filas y columnas como entre los elementos de cada una, ya que es una operación conmutativa y asociativa. Para este ejercicio, la unidad de computación paralelizable será cada una de las sumas de multiplicaciones de la matriz P.

El algoritmo de multiplicación debe componerse de etapas `Map` y `Reduce`, así que se debe buscar la estrategia necesaria para poder realizar los cálculos adecuándolos a estas dos etapas, y devolviendo el resultado deseado.

`Map`

La etapa `Map` debe formar estructuras con la forma <clave,valor> que nos permitan agrupar los resultados en la etapa `Reduce` a partir de dichas claves.

Para entender la formulación del ejercicio debemos definir primero la notación, especialmente en lo referente a los índices:

$$M_{i\times j} \cdot N_{j\times k} = P_{i\times k}$$

Donde:

- i es el número de filas en la matriz M y toma valores de 1 a 2.
- j es el número de columnas de la matriz M. Para poder realizar el producto de matrices, debe coincidir con el número de filas de la matriz N y para este ejercicio toma valores de 1 a 3.
- k es el número de columnas de la matriz N y toma valores de 1 a 2, igual que el numero de filas de la matriz M.

Para cada elemento $m_{i,j}$ de la matriz M, creamos pares repetidos como:

$$(M, i, j), x \rightarrow ((i, 1), x)(i, 2), x) \ldots ((i, A), x)$$

donde A es el numero de filas de la Matriz M.

Para cada elemento $n_{j,k}$ de la matriz N, crearemos pares como:

$$(N, i, j), x \rightarrow ((1, j), x)(2, j), x) \ldots ((A, j), x),$$

donde A es el numero de columnas de la Matriz N.

Aplicando esta fórmula al ejemplo, encontramos los siguientes valores `Map` para la matriz M:

$$((1,1),1)((1,2),1)((1,3),1)$$
$$((1,1),2)((1,2),2)((1,3),2)$$
$$((1,1),3)((1,2),3)((1,3),3)$$
$$((2,1),4)((2,2),4)((2,3),4)$$
$$((2,1),5)((2,2),5)((2,3),5)$$
$$((2,1),6)((2,2),6)((2,3),6)$$

Aplicando esta fórmula al ejemplo, encontramos los siguientes valores `Map` para la matriz N:

$$((1,1),6)((2,1),6)((3,1),6)$$
$$((1,1),5)((2,1),5)((3,1),5)$$
$$((1,1),4)((2,1),4)((3,1),4)$$
$$((1,2),3)((2,2),3)((3,2),3)$$
$$((1,2),2)((2,2),2)((3,2),2)$$
$$((1,2),1)((2,2),1)((3,2),1)$$

La etapa `Shuffle` agrupa los valores en las mismas «particiones» para poder ser procesados por la etapa `Reduce`. Así, el `Shuffle` agrupará por la primera `key`:

$$((1,1),1)((1,1),6)$$
$$((1,1),2)((1,1),5)$$
$$((1,1),3)((1,1),4)$$
$$\ldots$$

Y así para cada clave (1,2), (1,3), etc.

`Reduce`

El proceso de `Reduce` toma los valores agrupados por cada clave y debe encontrar el mecanismo de agrupación. Tomemos todos los valores de cada clave y los separamos en dos listas, los correspondientes a la matriz M y los de la matriz N. No mantenemos una referencia a qué vector pertenece cada valor (M o N), pero programáticamente puede añadirse.

Por ejemplo, para la clave (1,1):

$$(1,1), 1 \cdot 6 + 2 \cdot 5 + 3 \cdot 4$$

Así, tomamos cada elemento ordenado por el segundo índice, y multiplicamos cada valor por el correspondiente de la otra lista según la ordenación, sumando el resultado.

El proceso de `Reduce` debe repetir la misma operación para todas las claves disponibles.

Podemos concluir que, aun siendo un paradigma estricto, nada impide crear pares de <clave, valor> complejas que el `Reduce` pueda procesar de modo que se consiga la agrupación deseada. Sin embargo, un proceso matemático relativamente común como el del ejemplo es relativamente complejo de implementar en MapReduce.

9.1.3. Limitaciones

MapReduce fue un paradigma muy novedoso para poder explorar grandes archivos en entornos distribuidos. Es un algoritmo que puede desplegarse, en su implementación Hadoop, en equipos de bajo coste o convencionales (o también llamado *commodity*), lo que lo hizo muy atractivo. Varias herramientas que ofrecen funcionalidades diversas en el ecosistema *big data* como Hive o Sqoop se han desarrollado implementan-

do MapReduce. Sin embargo, el algoritmo presenta algunas limitaciones importantes:

- Utiliza un modelo forzado para cierto tipo de aplicaciones, que obliga a crear etapas adicionales MapReduce para ajustar la aplicación al modelo. Puede llegar a emitir valores intermedios extraños o inútiles para el resultado final, e incluso creando funciones `Map` y/o `Reduce` de tipo identidad, es decir, que no son necesarias para la operación a realizar pero que deben crearse para adecuar el cálculo al modelo.

- MapReduce es un proceso *file-based*, lo que significa que el intercambio de datos se realiza utilizando ficheros. Esto produce un elevado flujo de datos en lectura y escritura de disco, afectando a su velocidad y rendimiento general si un procesado concreto está formado por una cadena de procesos MapReduce, almacenando datos intermedios también en disco.

Para solucionar estas limitaciones apareció Apache Spark, que cambia el modelo de ejecución de tareas, haciéndolo más ágil y polivalente, como se presentará más adelante.

9.2. Apache Spark

Como hemos visto en la sección anterior, MapReduce es un potente algoritmo para procesar datos distribuidos, altamente escalable y relativamente simple. Sin embargo, presenta muchas debilidades en cuanto a rendimiento y versatilidad para implementar algoritmos más complejos. Así, en esta sección exploraremos cuáles son los *frameworks* más utilizados en el

procesado de grandes volúmenes de datos para tareas de análisis de datos y aplicación a algoritmos complejos, centrando nuestra atención en Apache Spark (Zaharia *et al.*, 2015).

Apache Spark[1] es un *framework* para el procesamiento de datos distribuidos que ofrece un alto rendimiento y es compatible con todos los servicios que forman parte del ecosistema de Apache Hadoop.

En esta sección nos centraremos en el modo interno de funcionamiento de Spark y como paraleliza sus tareas, convirtiéndolo en el *framework* más popular para análisis en entornos de datos masivos.

9.2.1. *Resilient distributed dataset* (RDD)

Para entender el procesamiento distribuido de Spark, debemos conocer uno de sus componentes principales, el *resilient distributed dataset* (RDD en adelante) sobre el que se basa su estructura de paralelización de trabajo.

Un RDD es una entidad abstracta que representa un conjunto de datos distribuidos por el *cluster* y una capa de abstracción encima de todos los datos que componen nuestro corpus de trabajo, independiente de su volumen, ubicación en el *cluster*, etc.

Las principales características de los RDD son:

- Es inmutable, una condición importante ya que evita que uno o varios *threads* actualicen el conjunto de datos con el que se trabaja.

[1]https://spark.apache.org

- Es *lazy loading*, es decir, solo se cargan y se accede a los datos cuando se necesitan.
- Puede almacenarse en memoria caché.

El RDD se construye a partir de bloques de memoria distribuidos en cada nodo, dando lugar a las llamadas «particiones». Este particionamiento lo realiza Spark y es transparente para el usuario, aunque también puede tener control sobre él. Si los datos se leen del sistema HDFS, cada partición se corresponde con un bloque de datos del sistema HDFS.

El concepto partición tiene un peso importante para comprender cómo se paraleliza en Spark. Cuando se lee un fichero, si este es mayor que el tamaño de bloque definido por el sistema HDFS (típicamente 64 MB o 128 MB), se crea una partición para cada bloque (aunque el usuario puede definir el número de particiones que quiere crear cuando está leyendo el fichero, si lo desea). Así:

- Un RDD es un *array* de referencias a particiones en nuestro sistema.
- La partición es la unidad básica para entender el paralelismo en Spark y cada partición se relaciona con bloques de datos físicos en nuestro sistema de almacenaje o de memoria.
- Las particiones se asignan con criterios como la localidad de datos (*data locality*) o la minimización de tráfico en la red interna.
- Cada partición se carga en memoria volátil (típicamente, RAM) antes de ser procesada.

Un RDD se puede construir a partir de:

1. Creación de un objeto «colección» paralelo, que creará una colección paralela en el nodo del controlador, hará particiones y las distribuirá entre los nodos del *cluster*, concretamente, en la memoria.

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

2. Creación de RDD desde fuentes externas, como por ejemplo HDFS o S3 de Amazon. En este caso se crearán particiones por bloque de datos HDFS en los nodos donde los datos están físicamente disponibles.

3. Al ejecutar cualquier operación sobre un RDD existente. Al ser inmutable, cuando se aplica cualquier operación sobre un RDD existente, se creará un nuevo RDD.

Sobre un RDD se pueden aplicar transformaciones o acciones, pero no se accede a los datos que componen un RDD hasta que se ejecuta sobre ellos una acción, siguiendo el modelo *lazy loading* ya mencionado. Así, cuando se trabaja con RDD a los cuales se aplican transformaciones, en realidad se está definiendo una secuencia de procesos a aplicar sobre el RDD que no se pondrán en marcha hasta la invocación de una acción sobre ella.

Tomemos la siguiente secuencia de comandos en Python para Spark:

```
data = spark.textFile("hdfs://...")
words = data.flatMap(lambda line : line.split(" "))
    .map(lambda word : (word, 1))
    .reduceByKey(lambda a, b : a + b)
words.count()
```

Esta porción de código muestra como se carga un RDD (variable «data») a partir de un fichero ubicado en el sistema distribuido HDFS. A continuación, se aplican hasta tres transformaciones (`flatMap`, `map` y `reduceByKey`). Sin embargo, ninguna de estas transformaciones se va a ejecutar de forma efectiva hasta la llamada al método «count()», que nos devolverá el número de elementos en el RDD «words». Es decir, hasta que no se ejecute una acción sobre los datos, no se ejecutan las operaciones que nos los proporcionan.

El programa en el cual se ejecuta el código mostrado es llamado «driver», y es el encargado de controlar la ejecución, mientras que las transformaciones son las operaciones que se paralelizan y se ejecutan de forma distribuida entre los nodos que componen el *cluster*. Los responsables de la ejecución en los nodos son conocidos como «executors». Es entonces cuando se accede a los datos distribuidos que componen el RDD. Así, si hay algún problema en alguna de las transformaciones solo será posible detectarla una vez se ejecute la acción que desencadena las transformaciones asociadas.

9.2.2. Modelo de ejecución Spark

DAG (*directed acyclic graph* o grafo dirigido acíclico) es un estilo de programación para sistemas distribuidos, que se presenta como una alternativa al paradigma MapReduce. Mientras que MapReduce tiene solo dos pasos (`Map` y `Reduce`), lo que limita la implementación de algoritmos complejos, DAG puede tener múltiples niveles que pueden formar una estructura de árbol, con más funciones tales como `map`, `filter`, `union`, etc.

En un modelo DAG no hay ciclos definidos ni un patrón de ejecución previamente definido, aunque si existe un orden

o dirección de ejecución. Apache Spark hace uso del modelo DAG para la ejecución de sus tareas en entornos distribuidos.

El DAG de Spark está compuesto por arcos (o aristas) y vértices, donde cada vértice representa un RDD y los ejes representan operaciones a aplicar sobre el RDD.

Las transformaciones sobre RDD pueden ser categorizadas como:

- *Narrow operation*: se utiliza cuando los datos que se necesitan tratar están en la misma partición del RDD y no es necesario realizar una mezcla de dichos datos para obtenerlos todos. Algunos ejemplos son las funciones `filter`, `sample`, `map` o `flatMap`.
- *Wide operation*: se utiliza cuando la lógica de la aplicación necesita datos que se encuentran en diferentes particiones de un RDD y es necesario mezclar dichas particiones para agrupar los datos necesarios en un RDD determinado. Ejemplos de *wide transformation* son: `groupByKey` o `reduceByKey`. Estas suelen ser las tareas de agregación y los datos provenientes de diferentes particiones se agrupan en un conjunto menor de particiones.

Cada RDD mantiene una referencia a uno o más RDD originales, junto con metadatos sobre qué tipo de relación tiene con ellos. Por ejemplo, si ejecutamos el siguiente código:

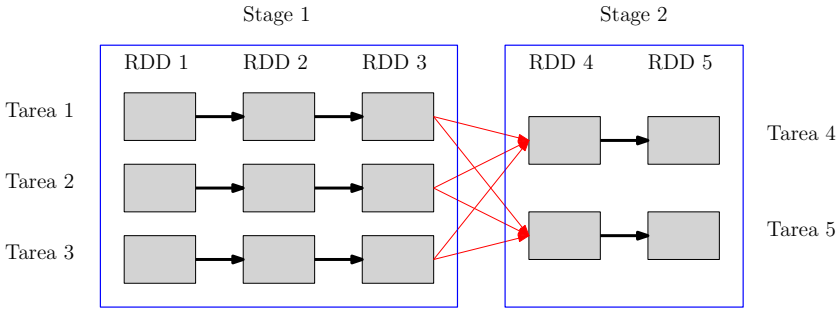
```
1 val b = a.map()
```

El RDD *b* mantiene una referencia a su padre, el RDD *a*. Es el llamada RDD *lineage*.

Al conjunto de operaciones sobre las que se realizan transformaciones preservando las particiones del RDD (*Narrow*)

se las llama «etapa» (*stage*). El final de un *stage* se presenta cuando se reconfiguran las particiones debido a operaciones de transformación tipo *wide*, que inician el llamado intercambio de datos entre nodos (`shuffling`), iniciando otro *stage* (véase figura 2).

Figura 2. Modelo de procesado de RDDs en Spark, diferenciando tareas y *stages* o etapas en función del tipo de transformación a aplicar



Fuente: elaboración propia

Las limitaciones de MapReduce se convirtieron en un punto clave para introducir el modelo de ejecución DAG en Spark.

Con la aproximación de Spark, la secuencia de ejecuciones, formadas por los *stages* y las transformaciones, las acciones a aplicar y su orden queda reflejado en un grafo DAG que se optimiza el plan de ejecución y minimiza el tráfico de datos entre nodos.

Cuando una acción es invocada para ser ejecutada sobre un RDD, Spark envía el DAG con los *stages* y las tareas a realizar al DAG Scheduler, quien transforma un plan de ejecución lógica (es decir, RDD de dependencias construidas mediante transformaciones) a un plan de ejecución físico (utilizando *stages* o etapas y los bloques de datos).

El DAG Scheduler es un servicio que se ejecuta en el *driver* de nuestro programa. Tiene tres tareas principales:

1. Determina las ubicaciones preferidas para ejecutar cada tarea y calcula la secuencia de ejecución óptima:

    - Primero se analiza el DAG para determinar el orden de las transformaciones.
    - Con el fin de minimizar el mezclado de datos, primero se realizan las transformaciones *narrow* en cada RDD.
    - Finalmente se realiza la transformación *wide* a partir de los RDD sobre los que se han realizado las transformaciones *narrow*.

2. Maneja los posibles problemas y fallos debido a una eventual corrupción de datos en algún nodo: cuando cualquier nodo queda no operativo (bloqueado o caído) durante una operación, el administrador del *cluster* asigna otro nodo para continuar el proceso. Este nodo operará en la partición particular del RDD y la serie de operaciones que tiene que ejecutar sin haber pérdida de datos.

Así, es el DAG Scheduler quien calcula el DAG de etapas para cada trabajo, realiza un seguimiento de qué RDDs y salidas de etapa se materializan y encuentra el camino mínimo para ejecutar los trabajos. A continuación, envía las etapas al Task Scheduler.

Las operaciones a ejecutar en un DAG están optimizadas por el DAG Optimizer, que puede reorganizar o cambiar el

orden de las transformaciones si con ello se mejora el rendimiento de la ejecución. Por ejemplo, si una tarea tiene una operación `map` seguida de una operación `filter`, el DAG Optimizer cambiará el orden de estos operadores, ya que el filtrado reducirá el número de elementos sobre los que realizar el `map`.

9.2.3. MLlib

Entre las varias API de trabajo de Spark, inicialmente la más popular es la API de aprendizaje automático (*machine learning*). La comunidad de contribuidores a esta API es la más grande del ecosistema Spark y en cada nueva versión se incluyen nuevas técnicas. Aunque recientemente la aparición de nuevos *frameworks* como TensorFlow, del que hablaremos más adelante, han desacelerado el desarrollo de librerías MLlib de Spark.

El análisis de datos es uno de los campos de trabajo más atractivos en el sector tecnológico y uno de los que tienen mayor demanda profesional. En dicho campo se trabaja de forma intensiva utilizando las herramientas de minería de datos más completas, como R, Python o Matlab.

Estas plataformas presentan un alto grado de madurez, incorporan multitud de librerías de aprendizaje automático y, también, muchos casos de uso y experiencias desarrollados y acumulados a lo largo de los años. Sin embargo, Spark ofrece una implementación paralela.

ML Pipelines

Como *framework* para trabajar con la API de aprendizaje automático de Spark, se ofrece la herramienta *pipeline*, que

permite crear cadenas de trabajo propias de una tarea de minería de datos. Una *pipeline* consta de varios componentes:

- *Dataframe* de entrada.
- *Estimator*, que es el responsable de generar un modelo.
- Parámetros para generar los modelos.
- *Transformer*, que es el modelo generado a partir de unos parámetros concretos.

Una *pipeline* consta de una cadena de estimadores y transformaciones. El DAG de un ML Pipeline no tiene por qué ser secuencial, donde la entrada de cada *stage* es la salida de la anterior, pero debe poder ser representado mediante ordenación topológica.[2]

Programación paralela en Spark

Se ha mencionado en diferentes partes de este capítulo como Spark distribuye tareas entre sus nodos. En esta sección mostraremos algunas ideas de cómo lo realiza.

Cuando Spark distribuye tareas entre los diferentes nodos, estos deben ser capaces de ejecutar el código correspondiente a dichas tareas sobre las porciones de datos correspondientes. Sin embargo, dichas tareas deben tener alguna información de referencia sobre la ejecución a realizar, variables que el código a ejecutar debe conocer. Tener variables que se envían juntamente con las tareas para que cada nodo las pueda modificar no sería eficiente, así que Spark utiliza dos tipos de variables para hacer su ejecución eficiente: las variables *broadcast* y las variables acumuladoras.

[2]https://en.wikipedia.org/wiki/Topological_sorting

Variables *broadcast*

Permiten al programador mantener una variable de solo lectura almacenada en caché en cada nodo en lugar de enviar una copia de ella junto con las tareas. Pueden utilizarse, por ejemplo, para dar a cada nodo una copia de un gran conjunto de datos de entrada a una aplicación o algoritmo.

A modo de ejemplo, consideremos un usuario con un conjunto de datos de entrenamiento de un modelo dado, suficientemente pequeño para caber en una sola máquina. En este caso podría utilizar scikit-learn para entrenar su algoritmo y luego usar Spark para hacer *broadcast* del modelo a los nodos y aplicarlo de forma distribuida en un conjunto de datos mucho más grande.

Variables acumuladoras

Un acumulador (*accumulator*) es una variable compartida entre nodos que permite la adición de valores en operaciones asociativas y conmutativas. Por defecto, solo soporta valores numéricos, aunque pueden añadirse otro tipo de datos extendiendo algunas clases u objetos base.

Estas variables son una pieza fundamental en la implementación y ejecución en paralelo de tareas en Apache Spark, ya que permiten a las tareas ejecutadas en los nodos compartir información global.

Spark *vs.* scikit-learn

Scikit-learn es uno de los paquetes de análisis de datos y aprendizaje automático para Python más populares (no el único), con una gran cantidad de implementaciones maduras

y consolidadas de multitud de algoritmos y técnicas de análisis estadístico.

Scikit-learn[3] y Spark no son competencia directa ya que trabajan en dos dominios diferentes y son dos buenos ejemplos de *frameworks* para realizar análisis de datos, pero para entornos de computación diferentes.

Apache Spark, por su parte, ofrece implementaciones de técnicas similares (aunque un número muy menor que scikit-learn) pero dichas implementaciones están desarrolladas para ser ejecutadas en entornos distribuidos, mientras que scikit-learn, aunque permite la definición de trabajos en paralelo en casi todos los algoritmos, está restringido al uso de los núcleos del ordenador en que se está ejecutando. En ambos casos, existe una amplia documentación y comunidad de soporte.

Así, podría concluirse que:

1. Cuando se trabaja con conjuntos de datos realmente grandes (GB, TB o incluso escala PB), es mejor utilizar Spark y la API MLlib para procesar dichos datos.

2. Cuando los conjuntos de datos encajan en la memoria de una sola máquina, se pueden construir algoritmos en Python usando scikit-learn junto con Pandas y otras librerías similares. Además, Spark no ofrece un rendimiento óptimo si se utiliza en un solo ordenador.

Spark MLlib trata de resolver tres grandes problemas que nos encontramos cuando trabajamos en entornos *big data* sobre los cuales queremos aplicar técnicas de aprendizaje automático:

[3]http://scikit-learn.org

1. Latencia del disco.
2. Limitaciones de memoria en una sola máquina.
3. Facilidad de interacción con grandes colecciones de datos.

Implementaciones de MLlib

Como se ha comentado, Apache Spark MLlib[4] provee de implementaciones de varias técnicas de aprendizaje automático. Estas implementaciones incluyen:

- Técnicas de regresión y clasificación supervisada.
- Técnicas de clasificación no supervisada (*clustering*).
- Recomendadores (*collaborative filtering*).
- Técnicas de reducción de dimensionalidad.
- Técnicas de extracción de características (*feature extraction*).
- Gestión y evaluación de modelos.
- Otras técnicas estadísticas.

SparkSQL

La API SparkSQL[5] es la continuación del proyecto Shark y su objetivo es proveer herramientas de procesado de datos de forma estructurados en entornos distribuidos. En un principio,

[4]http://spark.apache.org/docs/latest/ml-guide.html
[5]http://spark.apache.org/sql/

Apache Shark fue un *framework* para integrar Hive en Spark. Sin embargo, como ya se ha indicado, Hive es una herramienta orientada a procesos *batch* y la industria buscaba aplicaciones con latencias menores que Hive no podía ofrecer. Además, Shark heredó una gran cantidad de aplicaciones y código Hive difícil de mantener y optimizar. Así, SparkSQL nace con la idea de tomar las fortalezas de Hive, evitando sus debilidades. En el núcleo de SQLSpark se encuentra el *catalyst optimizer* un *framework* extensible que hace uso de las funcionalidades del lenguaje Scala para optimizar la ejecución de consultas.

Desde la versión 2 de Spark, además, la API de SparkSQL se convierte en un elemento clave para trabajar con datos distribuidos.

DataFrames y Datasets

SparkSQL introduce la API DataFrame, una librería que permite trabajar con los datos distribuidos como si fueran tablas. El DataFrame es la principal abstracción en SparkSQL y se construye en base a un RDD de Spark, pero a diferencia de este, el DataFrame contiene objetos del tipo *Row*, mientras que el RDD contiene «elementos» genéricos. Un Dataset es un Dataframe tipado, esto es, de un tipo concreto definido, no de una colección de *rows*. Cuando un Dataframe se codifica a un tipo concreto (utilizando los llamados *encoders* de Spark), se convierte en un Dataset, con nombres de campos (columnas en el dataset) y tipos concretos (*string, double, boolean*, etc.).

Así, un Dataset se caracteriza por su esquema, que contiene la información relativa a sus datos (metadatos), como por ejemplo, nombres de los campos, tipos, etc. Los Datasets son muy flexibles y pueden construirse a partir de ficheros con datos estructurados, como ficheros Parquet y JSON, o pueden

ser el resultado de la transformación de un RDD ya existente o la agregación de otros Datasets.

Un Dataset comparte muchas características con los RDD: son inmutables, una consulta sobre un Dataset da como resultado otro Dataset (también en modo *lazy*) y una acción desencadena la ejecución de la consulta en sí (de forma idéntica a como una acción en un RDD desencadena las transformaciones definidas sobre el RDD previamente), sin embargo, los Datasets tienen importantes mejoras en cuanto a rendimiento y consumo de memoria. Incluyendo una potente API que permite realizar consultas sobre los datos distribuidos mediante funciones como: `select()`, `where()`, `sort()`, `join()`.

Así como para trabajar sobre RDD, Spark utiliza el elemento SparkContext, desde Spark 2.0, se introduce la SparkSession, que encapsula el SparkContext y SQLContext, así como una serie de funcionalidades, parámetros de configuración y de contexto adicionales para trabajar con Datasets.

En el siguiente ejemplo veremos como la primera llamada carga un fichero en un dataset llamado «peopleDF» utilizando el comando `spark.read.cas`, donde «spark» es la variable SparkSession.

```
1 peopleDF = spark.read.csv("/path/to/file/fichero.csv")
2 peopleDFFiltered = peopleDF.select(peopleDF("name"), peopleDF
    ("age"))
3 peopleDFFiltered.sort(peopleDF.age.desc())
```

Las consultas también pueden ejecutarse de forma SQL directa como:

```
1 peopleDF.registerTempTable('people')
2 sqlContext('SELECT name, age FROM people WHERE age>15')
```

O pueden realizarse las consultas utilizando todas las capacidades de la API, con los metodos *select*, *where* o *filter*, como se muestra en esta porción de código en Java:

```
peopleDF.select(col("name"), col("age")).filter(col("age").
    greater(15))
```

Los resultados pueden guardarse en ficheros Parquet, tablas Hive u otros formatos específicos con la ayuda de librerías externas. El Dataframe es un nivel de abstracción superior a un RDD; así, el RDD del que deriva puede obtenerse como:

```
rddPadre = dataFrame.rdd
```

E inversamente, un RDD puede convertirse a un DataFrame mediante la llamada:

```
sqlContext.createDataFrame(rdd)
```

¿Cuál es la mejor herramienta para realizar consultas SQL sobre datos distribuidos? He aquí algunas de sus características comparativas:

- Hive: orientado a *batch*, es una herramienta estable y altamente escalable para cantidades de datos enormes, pero no ofrece capacidades de consulta interactiva.
- Impala: es la más rápida, muy orientada a la inteligencia de negocio (*business intelligence*, BI) y a consultas SQL específicamente.
- Spark SQL: como parte de la plataforma Spark, es altamente integrable con el resto de herramientas Spark, lo que la dota de un enorme potencial para definir cadenas de procesado complejas, incluyendo consultas SQL, *machine learning*, etc.

Parte IV

Procesamiento en flujo (*streaming*)

Capítulo 10

Captura y preprocesamiento de datos dinámicos

En este módulo trataremos las particularidades de los procesos de captura y preprocesamiento de datos dinámicos en entornos de datos masivos (*big data*).

Iniciaremos este capítulo analizando el proceso de captura de datos, haciendo especial énfasis en la captura de datos dinámicos o en *streaming*. Veremos qué son los datos en *streaming* y los componentes que intervienen en su creación, distribución y consumo. Posteriormente veremos más en detalle cómo se puede realizar la distribución de datos para facilitar la escalabilidad y garantizar la alta disponibilidad.

Continuaremos viendo las arquitecturas básicas que dan soporte a estas tareas, prestando atención a los datos en *streaming*, que por su casuística requieren de soluciones más complejas y apartadas de los procesos analíticos «tradicionales».

10.1. Conceptos básicos

El primer paso para cualquier proceso de análisis de datos consiste en la captura de los mismos, es decir, debemos ser capaces de obtener los datos de las fuentes que los producen o almacenan, para posteriormente, poder almacenarlos y analizarlos.

En este sentido, el proceso de captura de datos se convierte en un elemento clave en el proceso de análisis de datos, ya sea masivos o no. No será posible, en ningún caso, analizar datos si no hemos sido capaces de capturarlos cuando estaban disponibles.

Los **datos dinámicos** o en *streaming* son datos que se producen de forma continua, y que deben ser capturados durante un umbral limitado de tiempo, ya que no estarán disponibles pasado un determinado periodo de tiempo.

10.2. Captura de datos en *streaming*

Los datos en *streaming* (Garofalakis *et al.*, 2016; Kleppmann, 2016) son datos que se generan (y publican) de forma continua. Normalmente este tipo de datos se genera de forma concurrente por múltiples agentes y los datos generados suelen ser de pequeño tamaño (del orden de kilobytes).

Ejemplos de datos en *streaming* son los datos enviados por sensores que se encuentren en la calle, datos bursátiles, datos cardiovasculares enviados por un *smart watch*, datos sobre las actividades que realiza un usuario de una aplicación móvil o datos sobre redes sociales (*clicks*, *me gusta*, compartir información, etc.).

La existencia de este tipo de datos no es nueva. De hecho las tecnologías de bases de datos han ido proponiendo soluciones que tratan datos en *streaming* desde hace décadas. No obstante, la irrupción de nuevas tecnologías que permiten generar y enviar más datos y con más frecuencia (internet de las cosas y *smart cities*, por ejemplo), las mejoras en los sistemas distribuidos, la irrupción de los microservicios, la evolución de los sistemas analíticos y la adopción de una cultura analítica de forma masiva, ha provocado un resurgimiento del interés sobre los *streams* de datos.

Este libro presentará una breve introducción sobre el tema, proponiendo una definición de *stream* de datos, mostrando cómo pueden enviarse los datos desde los puntos de creación a los puntos de consumo y estudiando cómo pueden prepararse los datos para dar respuestas a las necesidades analíticas, introduciendo algunas de las tecnologías de envío y procesamiento más relevantes.

10.2.1. *Streams* de datos

Un *stream* de datos puede verse como un conjunto de parejas ordenadas (D, t), donde D es un conjunto de datos de interés y t es un número real positivo. Las distintas parejas están ordenadas en función de su valor de t.

La variable t de cada pareja indica el orden en el que se produjeron/enviaron los datos. Así pues, existirá una función de orden sobre t que indica indica que datos se generaron/enviaron primero.

Siguiendo esta representación, una secuencia de temperaturas enviada por un sensor que se encuentra en la calle podrá verse de la siguiente forma:

```
1 (<25 grados, 70% humedad>, 1483309623)
```

```
(<25 grados, 70% humedad>, 1483309624)
(<26 grados, 60% humedad>, 1483309625)
(<15 grados, 90% humedad>, 1483309626)
```

donde *<grados, humedad>* indica los datos de interés (las medidas de temperatura y humedad) y el número indica la fecha en que se recogieron estas medidas utilizando tiempo Unix.[1]

En función de si la t viene determinada por un momento en el tiempo (un sensor mide la temperatura cada milisegundo por ejemplo) o por el orden causal de determinados eventos (un internauta ha eliminado un producto de su cesta de la compra) estaremos hablando de *streams* originados por el **tiempo** o por **eventos**.

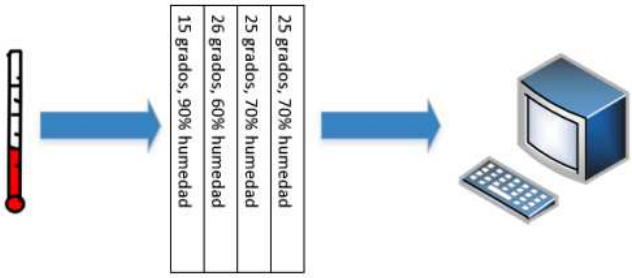
Normalmente, cuando se habla de *streams*, se utiliza una visualización orientada a colas, donde se representan los datos de forma consecutiva en una lista en función de cuando se originaron (su valor t). A veces se obvia el valor de la t en la representación, mostrando solo el orden causal de los datos. Por tanto, el *stream* anterior podría representarse como se muestra en la figura 1.

En el ejemplo de la figura 1 se ha asumido que el sistema tendrá un consumidor. Es decir un servicio que consultará los datos que vayan publicándose en el *stream* para realizar alguna tarea, como por ejemplo monitorizar los valores recibidos y hacer saltar una alarma cuando la temperatura o la humedad estén fuera de unos rangos de control prefijados.

Utilizar sistemas de envío y procesamiento en *streaming* es beneficioso en escenarios donde nuevos datos se generan de forma continua. Son sistemas de interés general, que se encuentran presentes en todo tipo de negocio. Desde un pun-

[1]Para representar una fecha en tiempo Unix se utiliza un número natural. El valor de este número indica los segundos transcurridos desde la medianoche del 1 de enero del 1970.

Figura 1. Representación del *stream* de datos generado por un sensor de temperatura



Fuente: elaboración propia

to de vista analítico, las posibilidades del procesamiento de *streams* son muchas y muy variadas. Las más simples son la recolección y procesamiento de *logs*, la generación de informes o cuadros de mando y la generación de alarmas en tiempo real. Las más complejas incluyen análisis de datos más complejos, como el uso de algoritmos de aprendizaje automático o de procesamiento de eventos.

Desde un punto de vista funcional, podríamos dividir los sistemas en *streaming* en tres componentes:

- **Productores**: son los sistemas encargados de generar y enviar los datos de forma continua. El sistema productor de datos puede ser muy variado, desde complejos sistemas que generan información bursátil o meteorológica, hasta sencillos sensores distribuidos por el territorio que miden y envían información sobre la temperatura o la polución. Es común que existan distintos productores

en un mismo sistema. En el ejemplo de la figura 1 solo hay un productor: el sensor que se encarga de medir la temperatura y la humedad ambiente y enviar los valores de la medición.

- **Consumidores**: son los sistemas encargados de recoger y procesar los datos producidos. El número de consumidores asociado a un *stream* de datos puede ser superior a uno. Estos sistemas pueden tener múltiples objetivos, en el contexto analítico, por ejemplo, estos sistemas pueden realizar desde simples agregaciones de datos, hasta complejos procesos para transformar y enriquecer los datos con el objetivo de poblar otros sistemas y crear otros *streams* de datos. En el caso del ejemplo tenemos solo un consumidor: un computador que recoge las temperaturas y humedades y comprueba si están dentro de un rango aceptable.

- **Mensajería**: es el sistema que se encarga de transmitir los datos, desde su productor hasta sus potenciales consumidores. Este sistema puede ser muy simple y utilizar conexiones directas para enviar los datos entre productores y consumidores. No obstante, se tienden a utilizar sistemas de mensajería más complejos que garanticen una alta tolerancia a fallos y una mayor escalabilidad.

10.2.2. Envío y recepción de datos en *stream*

Esta sección trata sobre los sistemas de mensajería. Estos sistemas permiten el intercambio de datos entre distintos ordenadores.

Cuando queremos compartir datos entre distintos dispositivos podemos tomar distintas alternativas. Quizá la alternati-

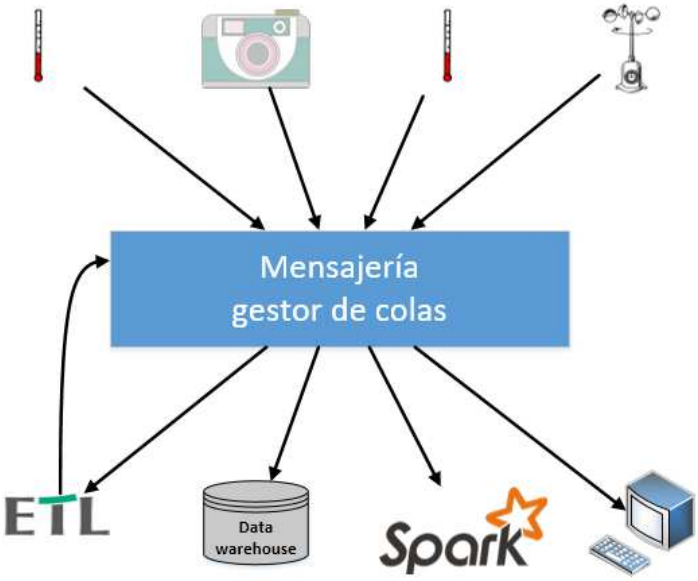
va más sencilla sería utilizar una arquitectura cliente-servidor. En un entorno cliente-servidor los datos se generan en el cliente (el productor) y se envían directamente al servidor (el consumidor). Recordemos que en el caso general este sistema podrá tener múltiples productores y consumidores, por tanto tendríamos que implementar un sistema cliente servidor con tantos clientes como productores haya y tantos servidores como consumidores haya.

Este sistema es muy simple pero presenta diversos problemas. El más relevante es su difícil evolución, ya que al añadir (o eliminar) nuevos productores (o consumidores), deberemos modificar todos los programas servidor (o cliente) para establecer las conexiones directas entre los nuevos elementos de la red. Otros problemas de esta aproximación es su baja tolerancia a fallos (en casos de caídas de red o de servidores los mensajes enviados podrían perderse irremediablemente) y su difícil escalabilidad (al no haber infraestructura propia para la gestión del envío y recepción de los mensajes, para mejorar el rendimiento de los envíos deberíamos mejorar la velocidad de la red, añadir nuevos nodos en el sistema no provocaría ninguna mejora).

Una alternativa a la arquitectura cliente-servidor sería utilizar sistemas de mensajería para realizar el envío de los datos en *stream*. Dichos sistemas añaden una estructura de red intermedia para desacoplar los sistemas creadores de los sistemas consumidores. Estos sistemas proporcionan también una mayor disponibilidad, escalabilidad y tolerancia a fallos. En la figura 2 podemos ver un ejemplo de sistema de mensajería.

El sistema de la figura 2 se compone de cuatro productores y cuatro consumidores. Los productores son de distinto tipo: dos sensores de temperatura, una cámara fotográfica y

Figura 2. Ejemplo de utilización de sistema de mensajería



Fuente: elaboración propia

un anemómetro (sensor de viento). La información recolectada por estos sistemas se envía al sistema de gestión de colas. Este es el encargado de garantizar que la información enviada (los mensajes) lleguen a sus potenciales consumidores. Estos consumidores podrían ser, por ejemplo, un *data warehouse* que almacene la información recolectada hasta el momento, un cuadro de mando, un sistema Spark para hacer analíticas en tiempo real y un proceso de extracción, transformación y carga que agregue y refine los datos. Este sistema ETL podría generar nuevos datos, como por ejemplo, estimar la temperatura de determinados puntos geográficos en función de las temperaturas recibidas, y añadir esta nueva información

en un nuevo *stream* de datos. En este caso el proceso ETL se comportaría también como productor para otro *stream* de datos.

Los sistemas de mensajería siguen básicamente dos modelos: el de **colas** y el de **publicación/suscripción**. En ambos sistemas los mensajes de distribuyen temáticamente en distintos *streams* (ya sea en colas o en temas). Una cola puede tener asignados un conjunto de consumidores válidos; cada mensaje de la cola se envía a uno de ellos. En el sistema de publicación/suscripción, los mensajes de un tema se difunden a todos los consumidores suscritos al tema.

El sistema intermedio de mensajería es lo que se denomina *message oriented middleware* (o MOM) y puede ser *software* o *hardware*. Este sistema asume la responsabilidad de transferir los mensajes entre aplicaciones, interconectando distintos elementos de un sistema distribuido y facilitando la comunicación entre ellos. Este *middleware* deberá gestionar los problemas de interoperabilidad de datos que puede haber debido a la heterogeneidad entre los distintos productores (distintos protocolos de comunicación o tipos de datos incompatibles) y la heterogeneidad entre productores y consumidores. También deberá proporcionar medidas para promover una alta disponibilidad en el proceso de intercambio de información.

Las ventajas de utilizar este tipo de sistemas son las siguientes:

- **Permite el envío de datos de forma asíncrona**: los datos recibidos se guardan en una cola y se van procesando a medida que el consumidor los solicita.
- **Permite desacoplar los distintos componentes del sistema distribuido**: se podrán añadir o eliminar crea-

dores o consumidores de información sin necesidad de modificar/ajustar los nodos del sistema distribuido.

- **Permite ofrecer una alta disponibilidad**: facilita la implementación de medidas para garantizar que los mensajes no se pierden ante caídas de red o de nodos, como podrían ser la replicación de los datos enviados o la definición de protocolos para garantizar que los datos de un *stream* se eliminan solo cuando han sido consumidos por un número mínimo de consumidores.

- **Facilita la escalabilidad**: facilita la escalabilidad horizontal permitiendo, por ejemplo, distribuir un *stream* en distintos nodos de la red.

Existen distintos protocolos de mensajería que pueden utilizarse para realizar el envío de mensajes entre productores y consumidores. Los más populares en el contexto del *big data* y de *internet of things* son quizá AMQP[2] (Advance Message Queuing Protocol), MQTT[3] (Message Queue Telemetry Transport) y STOMP[4] (Simple streaming Text Oriented Messaging Protocol).

Estos protocolos son adoptados por distintos sistemas de mensajería que implementan una capa *middleware* para el envió y recepción de mensajes. Algunos de los más populares son: Apache Kafka, Apache ActiveMQ[5] y RabbitMQ.[6]

A continuación introduciremos Apache Kafka con el objetivo de mostrar el funcionamiento de este tipo de sistemas.

[2]https://www.amqp.org/
[3]http://mqtt.org/
[4]https://stomp.github.io/
[5]https://www.amqp.org/
[6]https://www.rabbitmq.com/

10.2.3. Apache Kafka

Apache Kafka[7] es un sistema de mensajería distribuido y replicado de código abierto. Tiene su origen en LinkedIn, donde se creó para gestionar los flujos de información entre sus diferentes aplicaciones.

Kafka se ejecuta en un *cluster*, que puede estar compuesto por uno o más nodos. En Kafka, los *streams* pueden fragmentarse y distribuirse entre distintos nodos, permitiendo así escalabilidad horizontal. También permite la replicación de los mensajes enviados para proveer de una alta disponibilidad en caso de fallos en la red o en los nodos del *cluster*. De hecho, Kafka proporciona las siguientes garantías:

1. Los mensajes enviados por un productor se añadirán al *stream* en el orden de envío.

2. Los consumidores pueden obtener los mensajes en el orden en que fueron enviados.

3. Siendo *N* el número de replicas de un determinado *stream*, el sistema garantizará la disponibilidad de los datos siempre y cuando el número de nodos que fallen no supere el valor de *N-1*; es decir, siempre que quede al menos una replica en funcionamiento.

Los datos enviados a Kafka se almacenan en registros. Estos registros se asignan a distintos *streams* de datos en función de su contenido (categoría o propósito). Cada uno de estos *streams* de datos o categorías se denomina *topic*. Los registros se componen por una tripleta de la forma <clave, valor, timestamp>.

[7]https://kafka.apache.org/

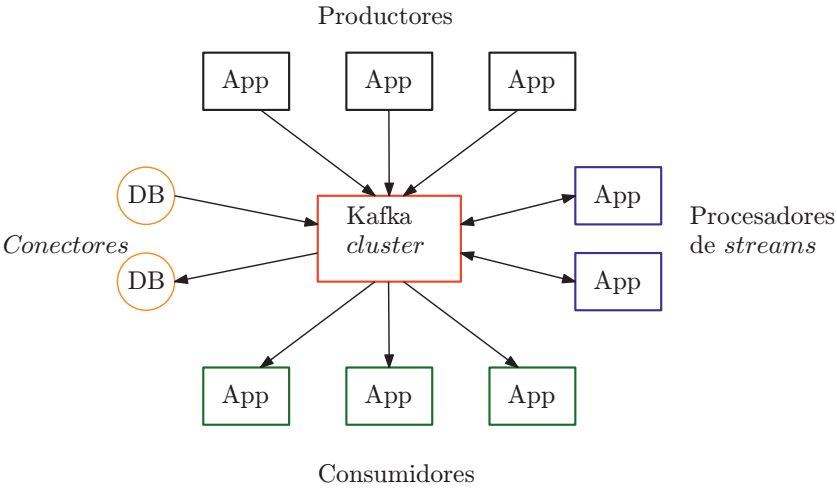
Otra característica interesante de Kafka es que, a diferencia de otros sistemas de mensajería, el consumo de un dato por parte de un productor no elimina el dato del *stream*. Los datos enviados a Kafka tienen fecha de caducidad (se puede configurar el tiempo que deben estar disponibles) y se mantienen en el sistema durante ese tiempo. Cuando se llega a la fecha límite los datos se eliminan. Eso permite mejorar disponibilidad, ya que los consumidores pueden leer datos en cualquier momento, incluso cuando otros consumidores ya hayan consumido el dato. Por otra parte, traspasa la decisión sobre qué datos leer a cada consumidor. El consumidor que consume datos de un *stream* deberá saber qué datos leyó, qué datos le interesa leer y en qué orden (en Kafka un consumidor puede consumir los datos en un orden distinto al orden de llegada).

Arquitectura de Apache Kafka

Los principales componentes que podemos encontrarnos en Kafka son los siguientes, tal y como se muestra en la figura 3:

- **Productor**: son los sistemas que envían datos a Kafka. Estos datos se envían en forma de registro y podrán ser publicados en uno o más *topics*.
- ***Cluster***: el sistema encargado de la recepción, fragmentación, distribución, replicación y envío de mensajes. Puede estar compuesto de distintos nodos, cada uno de ellos denominado *broker*.
- **Consumidor**: son los sistemas destino del *stream* de datos. Se pueden suscribir a uno o más *topics* para obtener los mensajes de los mismos.

Figura 3. Principales componentes de Kafka



Fuente: elaboración propia

- **Conector**: permiten crear consumidores (o productores) reusables que consuman (o publiquen) datos en uno o más topics a partir de bases de datos o aplicaciones existentes.

- **Procesador de *streams***: permiten crear aplicaciones para preprocesar y enriquecer los datos de un *stream* de datos. Estos sistemas actúan como consumidores de uno o más *topics*, realizan operaciones sobre los datos proporcionados, y escriben los datos resultantes en uno o más *topics*.

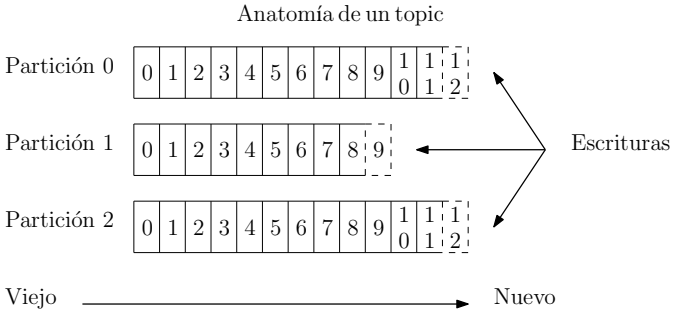
Representación interna de los *streams* en Kafka

Tal y como se ha comentado, los datos se distribuyen en *topics* según su categoría. Por tanto, cuando un productor envía datos a Kafka, deberá indicar a qué *topic* corresponden.

Por otro lado, los *topics* pueden estar ligados a cero, uno o más consumidores.

Como se puede ver en la figura 4 los *topics* se distribuyen en distintas particiones disjuntas. Esta distribución se realiza en aras de la escalabilidad horizontal, la distribución de carga y la paralelización.

Figura 4. Distribución de un *topic* en 3 particiones



Fuente: elaboración propia

En la figura 4 se puede ver un ejemplo en el que se ha dividido un *topic* en 3 particiones. Cada partición es una secuencia de registros inmutable (una vez añadido un registro a la secuencia no puede modificarse). En cada partición los registros van añadiéndose de forma secuencial. Cada registro tiene un número llamado *offset* (su posición dentro de la secuencia) que lo identifica unívocamente dentro de la partición.

Por defecto, escoger en qué partición debe almacenarse cada registro es tarea de los productores. Por tanto, estos serán los responsables de garantizar una correcta distribución de datos en las distintas particiones de un *topic*.

Cuando un consumidor quiere consultar los datos de un *stream* deberá saber el *topic* y la partición en el que se han asignado los datos de interés. Para indicar qué datos se quie-

ren consumir, el consumidor deberá indicar también el *offset* de los registros que desea obtener. Tal y como se ha comentado anteriormente, el consumo de un registro no eliminará el registro del *topic*.

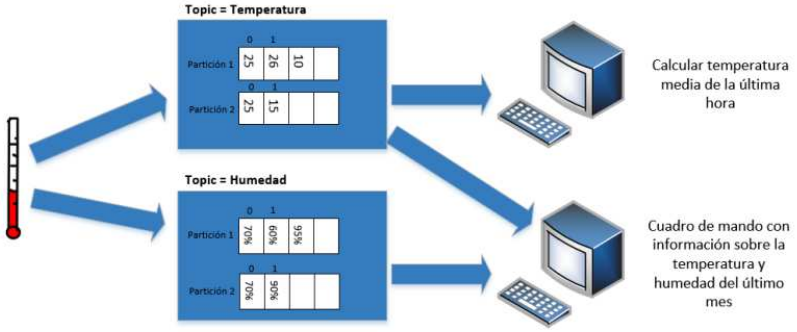
A título de ejemplo, vamos a ver cómo se gestionarían los siguientes datos en *streaming* utilizando Apache Kafka. Supongamos que los datos se generan cada segundo según las medidas de temperatura y humedad de un sensor ubicado en la calle y que se envían vía *streaming* a dos consumidores: uno que calcula la temperatura media de la última hora y otro que muestra un cuadro de mando con información de temperatura y humedad. Los datos a enviar serían los siguientes

```
1 (<25 grados, 70% humedad>, 1483309623)
2 (<25 grados, 70% humedad>, 1483309624)
3 (<26 grados, 60% humedad>, 1483309625)
4 (<15 grados, 90% humedad>, 1483309626)
5 (<10 grados, 95% humedad>, 1483309627)
```

Según estos datos, se ha decidido utilizar dos *topics* distintos: uno para enviar datos sobre temperatura (llamado *Temperatura*) y otro para enviar datos sobre humedad ambiente (llamado *Humedad*). Supongamos también que decidimos utilizar dos particiones para cada *topic* y que lo que hacemos es distribuir medidas consecutivas a particiones distintas. Por tanto, enviaríamos los valores de temperatura 25 (línea 1), 26 (línea 3) y 10 (línea 5) a la partición 1 del *topic Temperatura*, y los valores 25 (línea 2) y 15 (línea 4) a la partición 2 del *topic Temperatura*. Por otro lado, los valores 70 (línea 1), 60 (línea 3) y 95 (línea 5) se enviarían a la partición 1 del *topic Humedad* y el resto de valores a la partición 2. El estado de las particiones después de enviar los datos de ejemplo serían los mostrados en la figura 5.

Respecto a los consumidores, el primero solo necesita datos de temperatura para calcular la temperatura media. Por tanto, solo consultará datos del *topic Temperatura*. El segundo consumidor deberá proporcionar información sobre la temperatura y la humedad y, por tanto, deberá consultar ambos *topics*. Recordad que los consumidores deberán saber el topic, la partición y el *offset* de los datos que quieren consultar en cada momento.

Figura 5. Ejemplo de uso de Apache Kafka



Fuente: elaboración propia

Distribución y replicación

En Kafka las particiones son la unidad de distribución y replicación.

Las particiones se distribuirán entre distintos servidores del *cluster*, distribuyendo las particiones de un *topic* en distintos nodos siempre que sea posible.

Las particiones se replicarán en distintos servidores según un factor de replicación indicado por el usuario. La gestión

de réplicas se realizará siguiendo una estrategia *master-slave* asíncrona en la que una partición actúa de líder. La partición líder se encarga de gestionar todas las peticiones de lectura y escritura. Podrá haber cero o más particiones secundarias (o *followers* según terminología Kafka). Estas particiones secundarias replicarán de forma asíncrona las escrituras del líder y tendrán un rol pasivo: no podrán recibir peticiones de lectura/escritura de los consumidores/productores. En caso de que la partición líder caiga, una de las particiones secundarias lo sustituirá.

En un *cluster* Kafka, siempre que sea posible, cada nodo actuará como líder de una de sus particiones con el objetivo de garantizar que la carga del sistema se encuentre bien balanceada.

10.2.4. Apache Flume

Apache Flume[8] es una herramienta cuya principal funcionalidad es recoger, agregar y mover grandes volúmenes de datos provenientes de diferentes fuentes hacia el repositorio HDFS en casi en tiempo real. Es útil en situaciones en las que los datos se crean de forma regular y/o espontánea, de modo que a medida que nuevos datos aparecen ya sean *logs* o datos de otras fuentes, estos son almacenados en el sistema distribuido automáticamente.

El uso de Apache Flume no se limita a la agregación de datos desde *logs*. Debido a que las fuentes de datos son configurables, Flume permite ser usado para recoger datos desde eventos ligados al tráfico de red, redes sociales, mensajes de correo electrónico a casi cualquier tipo de fuente de generación de datos dinámica.

[8]https://flume.apache.org/

La arquitectura de Flume la podemos dividir en:

- **Fuente externa**: se trata de la aplicación o mecanismo, como un servidor web o una consola de comandos, desde el cual se generan eventos de datos que van a ser recogidos por la fuente.

- **Agente (*agent*)**: es un proceso Java que se encarga de recoger eventos desde la fuente externa en un formato reconocible por Flume y pasárselos transaccionalmente al canal. Si una tarea no se ha completado, debido a su carácter transaccional, esta se reintenta por completo y se eliminan resultados parciales.

- **Canal**: un canal actuará de almacén intermedio entre el agente y el sumidero (descrito a continuación). El agente será el encargado de escribir los datos en el canal y permanecerán en él hasta que el sumidero u otro canal los consuman. El canal puede ser «memoria», de modo que los datos se almacenan en la memoria volátil (RAM) o pueden ser almacenados en disco, usando ficheros como almacén de datos intermedios o una base de datos.

- **Sumidero (*sink*)**: será el encargado de recoger los datos desde el canal intermedio dentro de una transacción y de moverlos a un repositorio externo, otra fuente o a un canal intermedio. Flume es capaz de almacenar datos en diferentes formatos HDFS como por ejemplo texto: SequeneceFiles,[9] JSON[10] o Avro.[11]

[9]https://wiki.apache.org/hadoop/SequenceFile
[10]http://www.json.org/
[11]https://avro.apache.org/

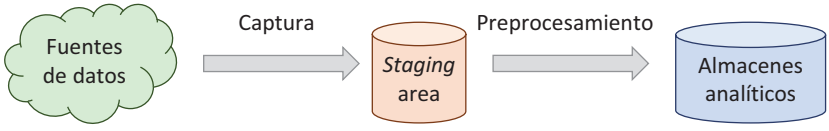
10.3. Arquitecturas de datos en *streaming*

Las tareas de preprocesamiento en datos continuos son similares a las vistas en la sección 7.2.2 para los datos estáticos. Por lo tanto, se sugiere al lector que la revise en caso de duda.

En el caso de trabajar con datos dinámicos, hay que tener en cuenta la velocidad de producción de estos y el hecho de que un error o fallo en cualquiera de los procesos de preprocesamiento puede provocar la pérdida de datos. Para evitar esto, se suele emplear un modelo de preprocesamiento independiente del proceso de captura de los datos.

En este modelo, ejemplificado en la figura 6, los datos son capturados y almacenados en un sistema de almacenamiento intermedio, conocido como *staging area*, donde los datos son almacenados de forma temporal. Las tareas de preprocesamiento cogen los datos de estos sistemas de almacenamiento intermedio y realizan la operaciones oportunas sobre los datos, y posteriormente almacenan los datos en los repositorios analíticos finales, donde se almacenan de forma duradera para su posterior análisis.

Figura 6. Esquema de una arquitectura que incorpora *staging area*



Fuente: elaboración propia

La principal ventaja de la arquitectura con *staging area* es la independencia entre los procesos de captura y preprocesamiento de datos. Es decir, los dos procesos quedan «desacoplados», con lo cual evitamos que uno puedo influenciar de forma negativa en el desarrollo del otro.

Esto es especialmente indicado en el caso de trabajar con datos en *streaming*, donde se pueden producir picos de producción de datos que lleguen a colapsar el sistema de preprocesamiento, especialmente en el caso de que este deba realizar alguna tarea de cierta complejidad. Por el contrario, si se usa una arquitectura de *staging area*, esta «absorbe» el pico de crecimiento de datos, mientras que el preproceso de datos continúa de forma independiente.

También es especialmente interesante en el caso de realizar operaciones de preprocesamiento de cierta complejidad, que podrían llegar a provocar la saturación del proceso de captura y con ello la pérdida de datos, dado que el sistema podría no ser capaz de procesar y almacenar la información al ritmo que se produce.

Capítulo 11

Almacenamiento de datos dinámicos

En los entornos en que se trabaja con datos en flujo, datos continuos o datos en *streaming*, básicamente nos encontramos con una característica diferencial muy relevante frente a los entornos de datos estáticos o procesamiento por lotes (*batch*). Esta característica es la velocidad de respuesta de la base de datos frente a las nuevas inserciones y consultas. Necesitamos sistemas gestores que puedan proporcionar una velocidad de respuesta muy elevada, para que podamos garantizar que podemos almacenar y procesar los datos que van llegando de forma continua.

En este capítulo veremos algunas de las bases de datos utilizadas en entornos de datos dinámicos, sus principales características y algunos ejemplos concretos de implementaciones que son ampliamente utilizados en la actualidad.

11.1. Almacenamiento de datos dinámicos

Como hemos comentado, la velocidad de respuesta en el acceso a los datos es clave en entornos de datos dinámicos. En este sentido debemos considerar la velocidad en las inserciones de nuevos datos y, también, la velocidad para acceder a los datos almacenados, generalmente a partir de consultas sobre la base de datos.

Los sistemas gestores de bases de datos diseñados para entornos en *streaming* deben proporcionar un acceso muy rápido, en las inserciones de nuevos datos y también en el acceso a los datos almacenados.

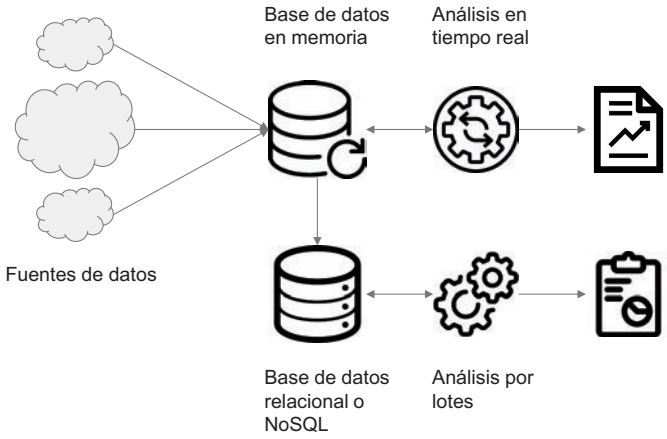
Generalmente, se utilizan dos bases de datos (o dos grupos de bases de datos) para dar soporte a un entorno de estas características:

- En primer lugar, se emplea una base de datos con acceso muy rápido a estos, que se encargará de ir almacenando los que van llegando y proporcionar el acceso por parte de los procesos de análisis en tiempo real. Las **bases de datos en memoria** (*in-memory databases*) son un tipo de bases de datos muy empleadas en estos escenarios, ya que proporcionan una alta velocidad en escritura y lectura de los datos.

- En segundo lugar, los datos suelen ser transferidos a un sistema de almacenamiento «definitivo», donde todos los datos son almacenados para realizar análisis posteriores tipo *batch*. Es decir, esta segunda base de datos sirve como almacén histórico donde poder consultar todos los datos y poder analizarlos por lotes. En este caso, se suelen emplear sistemas de almacenamiento como los

vistos en el capítulo 8, que permiten almacenar de forma persistente grandes cantidades de datos.

Figura 1. Ejemplo básico de almacenamiento en un entorno de procesamiento de datos dinámicos



Fuente: elaboración propia

La figura 1 muestra el ejemplo básico de esta arquitectura. Podemos ver que los datos son recogidos por una base de datos en memoria, que permite escrituras y lecturas con una alta velocidad (muy superior al resto de bases de datos relacionales o NoSQL). Los procesos de analítica en tiempo real emplean esta base de datos en memoria, que contiene los últimos datos recogidos, pero generalmente no contiene el conjunto completo de datos históricos. El análisis en tiempo real debe ser muy rápido, con lo cual se suele emplear solo una ventana reciente de los datos capturados, es decir, un subconjunto de estos, priorizando los datos más recientes.

De forma puntual, los datos de la base de datos en memoria son transferidos a un sistema de almacenamiento persistente,

que almacenará el conjunto completo de datos. En estos casos es habitual emplear bases de datos NoSQL o sistemas de ficheros distribuidos, ya que la cantidad de datos a almacenar suele ser muy grande y se incrementa con el tiempo. La función de este almacenamiento persistente es doble: por un lado, permite almacenar el conjunto completo de datos, ya que no sería eficiente hacerlo en la misma base de datos donde realizamos en análisis en tiempo real. Y, en segundo lugar, posibilita realizar otros tipos de análisis más complejos empleando procesos por lotes, que pueden facilitar la generación de otro tipo de información relevante.

11.2. Bases de datos en memoria

Las bases de datos en memoria (*in-memory databases*, IMDB) son un tipo de sistemas gestores de bases de datos que almacenan la información, principalmente, en la memoria principal de la computadora, a diferencia de los sistemas de bases de datos tradicionales que emplean el almacenamiento en disco. Esta diferencia permite a las bases de datos en memoria ser mucho más rápidas, ya que el acceso al disco es más lento que el acceso a la memoria, los algoritmos de optimización internos son más simples y ejecutan menos instrucciones de CPU. También son conocidas como sistema de base de datos de memoria principal (*main memory database system*, MMDB) o base de datos residente en memoria (*memory resident database*, MRDB).

Uno de los principales problemas con el almacenamiento de datos en memoria es la volatilidad de la memoria RAM. Específicamente, en el caso de una pérdida de energía, los datos almacenados en la RAM volátil se pierden. Por lo tanto,

este tipo de sistemas deben proveer alternativas para almacenar algún tipo de *backup* o copia de los datos para poder ser recuperados en caso de fallo en el sistema o de pérdida de energía.

Es interesante remarcar que el hecho de ser «en memoria» es independiente del modelo de base de datos, que puede ser relacional o cualquiera de los tipos de bases de datos NoSQL, entre otros. Así, existen bases de datos en memoria de tipo relacional, orientado a columnas, clave-valor, etc.

11.2.1. Soporte ACID

En las bases de datos es habitual hablar del soporte que ofrecen a las características ACID: atomicidad, consistencia, aislamiento y durabilidad (en inglés, *atomicity*, *consistency*, *isolation* y *durability*). En el caso de las bases de datos en memoria, es habitual encontrar soporte para las tres primeras propiedades, pero no es directo el soporte a la cuarta propiedad. Las bases de datos en memoria almacenan datos en dispositivos de memoria volátiles, que pierden la información almacenada cuando el computador pierde la energía o se reinicia. En este caso, se puede decir que los IMDB carecen de soporte para la parte de «durabilidad» de las propiedades ACID.

Aun así, muchos de los sistemas IMDB actuales han añadido la propiedad de «durabilidad» a partir de varios mecanismos, entre los cuales destacamos:

- Almacenar en memoria no volátil *snapshots* o *checkpoints* de forma periódica, que registran el estado de la base de datos en un momento dado en el tiempo. Este método ofrece durabilidad parcial, ya que es posible

perder algunos datos desde el último volcado a memoria no volátil.

- Almacenar el registro de transacciones, que registra los cambios de la base de datos en un sistema no volátil y facilita la recuperación automática de una base de datos en memoria.
- Sistemas de memoria principal no volátil, como por ejemplo NVDIMM[1] o NVRAM.[2]
- Sistemas de alta disponibilidad, que emplean la replicación de la base de datos en caso de fallo o caída de la base de datos principal.

11.2.2. Redis

Redis (REmote DIctionary Server)[3] es un almacén de estructura de datos en memoria de código abierto que implementa una base de datos distribuida de clave-valor en memoria con durabilidad opcional. En este sentido, es posible configurar la política de expiración, durabilidad y replicación de datos. Su principal ventaja frente a otros sistemas de almacenamiento de datos es su elevada velocidad de escritura y lectura de datos.

Redis ofrece funcionalidades como base de datos, caché y agente de mensajes (*message broker*). Por lo tanto, Redis va más allá de lo que podríamos considerar una base de datos, y puede ser utilizado como un sistema de publicación-suscripción. Los comandos `SUBSCRIBE` y `PUBLISH` son la base

[1]https://en.wikipedia.org/wiki/NVDIMM
[2]http://cort.as/-Enaf
[3]https://redis.io/

para que un cliente pueda suscribirse a un canal que les permite recibir todos los mensajes del productor, y a este ejecutar la publicación de mensajes en el canal indicado.

Redis popularizó la idea de un sistema que puede considerarse al mismo tiempo un almacén y un caché de datos, donde estos se escriben y leen siempre desde la memoria principal de la computadora. Para permitir su durabilidad, también se almacenan de forma que se puedan reconstruir en la memoria una vez que se reinicia el sistema.

El modelo de acceso a datos se basa en los comandos `SET` y `GET`, que permiten escribir y leer datos, respectivamente. Además, Redis incorpora los comandos `MULTI`, `EXEC` y `ROLLBACK` para implementar el concepto de transacción en la gestión de sus datos. Los comandos incluidos dentro de la transacción se almacenan y ejecutan en el orden indicado de forma atómica. La lista de comandos de Redis es muy extensa, y no es el objetivo de este libro una revisión de todos ellos.

Tipos de datos. Aunque normalmente se clasifica Redis como una base de datos clave-valor, es importante remarcar que soporta múltiples tipos de datos.

Concretamente, Redis es capaz de manipular los siguientes tipos de datos:

- Cadenas de caracteres (*strings*).
- Números enteros (*integer values*).
- Listas (*lists*), que contienen múltiples valores ordenados que pueden ofrecer funcionalidades de colas y pilas.
- *Hashes*, que permiten anidar pares de clave-valor.

- Conjuntos (*sets*), que contienen colecciones no ordenadas de elementos sin repetición y facilitan, entre otras operaciones, la intersección y unión de conjuntos.

- Conjuntos ordenados (*sorted sets*), que contienen colecciones ordenadas de elementos sin repetición.

Durabilidad. La opción más básica y simple de funcionamiento de Redis, que sería el uso como un sistema caché, mantiene los datos en memoria y no proporciona persistencia de datos. Esta opción, cuando el objetivo es proporcionar un sistema caché de alto rendimiento, es razonable, ya que la persistencia a disco siempre introduce un incremento del tiempo de latencia de los datos. Aun así, en el caso de emplear Redis como sistema de caché, es posible almacenar los pares de clave-valor en un disco para su persistencia mediante el comando `SAVE`.

Adicionalmente, el comando `SAVE` se puede parametrizar para permitir que el sistema almacene una instantánea (*snapshot*) de la estructura de datos cada cierto tiempo o cada vez que se modifican un cierto número de registros. Este sistema permite mantener la durabilidad parcial de los datos, pero se perdería información en el caso de caída o fallo del sistema (todos los datos escritos desde el último *snapshot*).

En el caso de requerir la durabilidad total de los datos, Redis proporciona un método basado en los ficheros de solo-escritura. En este caso, el sistema mantiene un registro de todos los comandos ejecutados, de forma que es capaz de reproducir los datos actuales en caso de caída o fallo del sistema.

Finalmente, otra opción que facilita la durabilidad de los datos es la replicación maestro-esclavo (*master-slave*). En este esquema se configuran uno o más servidores para actuar

como esclavo de un servidor principal. Todas las operaciones realizadas sobre el servidor principal se propagan a todos los esclavos que, por lo tanto, mantienen el mismo conjunto de datos que el servidor principal. En caso de fallo o caída del servidor principal, cualquiera de la réplicas puede sustituirle sin pérdida de datos.

Capítulo 12

Análisis de datos dinámicos

Aunque pueda parecer que el análisis de datos dinámicos es un campo relativamente nuevo, sus fundamentos teóricos se basan en enfoques estadísticos y computacionales bien establecidos. Aunque no ha sido hasta los últimos años que esta área de investigación ha experimentado un gran crecimiento debido al incremento de datos disponibles.

El principal problema cuando se analizan datos dinámicos es que dicho análisis debe realizarse bajo restricciones de recursos computacionales y en un periodo de tiempo determinado. También es necesario tener en cuenta que la velocidad a la que se genera nueva información no es siempre constante. Una vez resueltos estos problemas con las técnicas de captura y almacenamiento explicadas en los apartados anteriores, los tipos de problemas que podemos resolver son parecidos a los que resolvemos en el escenario del procesamiento en *batch*: tareas de clasificación, de regresión o de *clustering*, así como sistemas de detección de valores atípicos o *outliers*. Por falta de espacio en este apartado no cubriremos completamente estas técnicas, en su lugar, analizaremos diferentes formas

de aproximarnos a este tipo de análisis explicando las ideas principales y dando algunos ejemplos de cómo resolver tareas concretas tales como contar eventos en un flujo infinito de datos.

12.1. Soluciones basadas en datos y basadas en tareas

Las soluciones proporcionadas en este campo se pueden clasificar en *data-based* y en *task-based* dependiendo de su enfoque.

La idea detrás de las soluciones basadas en datos es usar un subconjunto de información del conjunto total de datos para realizar los análisis requeridos. Las diversas técnicas que se han utilizado en este sentido pueden dividirse en dos categorías:

- **Métodos de muestreo**: ya sea recogiendo muestras al azar del flujo de datos o seleccionando al azar fragmentos (subconjuntos) del flujo, los métodos de muestreo descartan parte de los datos entrantes, mientras realizan los análisis con los datos muestreados. El principal problema con este enfoque es que es difícil saber cuándo elegir una muestra o qué registros se deben almacenar, porque no hay un conocimiento previo del tamaño del conjunto de datos o de la estructura de información.

- **Métodos de resumen**: usan datos agregados o medidas estadísticas calculadas (que se recalculan continuamente) para proporcionar la información necesaria para los algoritmos de análisis de datos. En este caso, es la pérdida de información y/o precisión junto con la

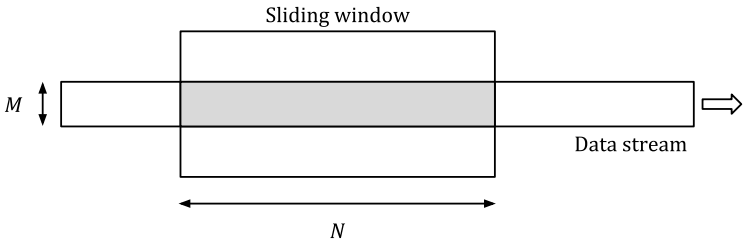
incapacidad para controlar las fluctuaciones de las distribuciones de los datos lo que hace que estos métodos no sean tan válidos como uno desearía.

Por otro lado, las soluciones basadas en tareas no realizan transformaciones en de datos, sino que modifican los métodos de extracción de datos para permitir su uso dentro del flujo. Estos métodos se pueden dividir en dos categorías:

- **Algoritmos aproximados**: son un tipo de algoritmos que están diseñados para resolver problemas donde encontrar una solución exacta es computacionalmente muy difícil o imposible, obteniendo un resultado aproximado junto con unos umbrales de confianza o de bondad del resultado. Es decir, en lugar de calcular soluciones exactas, solo garantizan una solución con un cierto límite de error.

- **Ventana deslizante o (*sliding window*)**: este tipo de métodos tienen un patrón muy extendido dentro de las ciencias de la computación que se conoce como *algoritmos de procesamiento en línea (online)*. Estos algoritmos procesan los datos de entrada elemento a elemento de forma secuencial, es decir, en el orden en que los elementos de entrada están disponibles, sin necesidad de que todos los elementos estén disponibles desde el inicio. Las aplicaciones que utilizan esta metodología mantienen una una *ventana deslizante* en la que guardan los datos más recientes. A medida que se reciben nuevos datos entrantes, esta ventana «avanza», por lo que las nuevas observaciones se guardan en su interior, como se puede ver en la figura 1. Los análisis de datos se realizan utilizando los datos disponibles dentro de la

Figura 1. Esquema de procesamiento en flujo utilizando un enfoque de *ventana deslizante*. En la figura, N es el *tamaño* de la ventana deslizante, en términos del número de muestras de la secuencia que se almacena, mientras que M es el número de *atributos* de las muestras



Fuente: elaboración propia

ventana y de forma opcional añadiendo versiones resumidas de los datos más antiguos, en forma de medidas estadísticas o datos agregados.

- **Agregación de la salida**: en este tipo de métodos se tiene en cuenta la capacidad de cálculo disponible para realizar el análisis en dispositivos con recursos limitados. Al adaptarse a la disponibilidad de recursos y a las velocidades de flujo de datos cuando los recursos se están agotando por completo, los resultados se fusionan y almacenan.

En las próximas secciones veremos como aplicar estas técnicas en dos casos concretos: (1) calcular estadísticos como la media o la desviación estándar y (2) contar eventos infinitos usando *sketches* que son estructuras de datos muy usadas en algoritmos aproximados.

12.2. Cálculo *online* de valores estadísticos

Es relativamente sencillo calcular de forma *online* valores estadísticos como el mínimo o el máximo. Veamos, por ejemplo, de forma detallada, cómo se haría en el caso del mínimo: para este cálculo solo es necesario inicializar una variable `min` a un valor muy grande.[1] Luego, cada vez que se recibe una nueva instancia en el flujo de datos S, se compara dicha instancia con la variable `min`. Si esta instancia es menor, se actualiza la variable `min` con su valor. Si repetimos esta comparación y posible actualización cada vez que llega un nuevo valor, conseguiremos almacenar en la variable `min` el valor más pequeño que hemos observado en S en cualquier momento del tiempo.

Otros valores estadísticos como la media son igualmente sencillos de calcular de forma *online*. En el caso de la media, solo necesitaremos guardar dos valores: la suma total de los valores `acc` del flujo S, junto con una variable auxiliar `counter` que cuente el número de elementos que hemos sumados. Es sencillo ver que podemos recuperar la media μ en cualquier momento realizado la siguiente división

$$\mu = \frac{acc}{counter} \tag{12.1}$$

Veamos ahora cómo se pueden calcular de forma *online* otros valores estadísticos más complejos. Para ello, en esta sección explicaremos como calcular la desviación estándar de forma incremental.

[1] O a un valor muy pequeño en el caso del máximo.

12.2.1. Método de Welford para calcular la desviación estándar

La desviación estándar es una medida de cuánto difiere un conjunto de datos de su media. Nos dice cuán dispersos están esos datos. Un conjunto de datos que esté agrupado en torno a un solo valor tendría una pequeña desviación estándar, mientras que un conjunto de datos que se encuentran distribuidos por todo un dominio de datos tendría una gran desviación estándar.

Dada una muestra $x_1, \ldots, x_N$, la desviación estándar se define como la raíz cuadrada de la varianza:

$$s^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}$$

donde $\bar{x}$ es la media de la muestra. La definición de la desviación estándar se puede convertir directamente en un algoritmo que calcula tanto la varianza S^2 y la desviación estándar S procesando dos veces los datos: (i) para calcular la media en una pasada sobre los datos y (ii) realizar una segunda pasada que calcule las diferencias al cuadrado respecto de la media. Sin embargo, hacer estas dos pasadas no es lo ideal, y puede ser poco práctico en entornos de procesamiento en flujo.

Un cálculo fácil nos da la siguiente identidad, que sugiere un método para calcular la varianza en una sola pasada, simplemente acumulando las sumas de x_i y de x_i^2:

$$\begin{aligned} s^2 &= \frac{\sum_{i=1}^{N}(x_i^2 - 2\bar{x}x_i + \bar{x}^2)}{N-1} \\ &= \frac{\sum x_i^2 - 2N\bar{x}^2 + N\bar{x}^2}{N-1} \\ &= \frac{\sum x_i^2 - N\bar{x}^2}{N-1} \end{aligned}$$

El pseudocódigo para el cálculo de varianza en una sola pasada podría tener el siguiente aspecto:

```
variance(samples):
  sum := 0
  sumsq := 0
  for x in samples:
    sum := sum + x
    sumsq := sumsq + x ** 2
  mean := sum/N
  return (sumsq - N × mean ** 2)/(N-1)
```

Aunque este método puede resultar válido a simple vista, no hay que usarlo nunca. Este es uno de esos casos donde un enfoque matemáticamente simple resulta dar resultados incorrectos por ser numéricamente inestable. En casos simples, el algoritmo parecerá funcionar bien, pero eventualmente puede encontrar un conjunto de datos que tenga el siguiente problema: si la varianza es pequeña en comparación con el cuadrado de la media, y calcular la diferencia provoca una cancelación donde se eliminan los dígitos iniciales significativos, el resultado tiene un gran error relativo. De hecho, incluso puede llegar a calcular una variación negativa, algo que es matemáticamente imposible.

El método de Welford es un método que realiza una única lectura del flujo de datos y que se puede utilizar para calcular la varianza. Se puede derivar al observar las diferencias entre las sumas de las diferencias al cuadrado para las muestras N y $N-1$. Es realmente sorprendente lo simple que resulta dicha diferencia:

$$
\begin{aligned}
&(N-1)S_N^2 - (N-2)S_{N-1}^2 \\
&= \sum_{i=1}^{N}(x_i - \bar{x}_N)^2 - \sum_{i=1}^{N-1}(x_i - \bar{x}_{N-1})^2 \\
&= (x_N - \bar{x}_N)^2 + \sum_{i=1}^{N-1}((x_i - \bar{x}_N)^2 - (x_i - \bar{x}_{N-1})^2) \\
&= (x_N - \bar{x}_N)^2 + \sum_{i=1}^{N-1}(x_i - \bar{x}_N + x_i - \bar{x}_{n-1})(\bar{x}_{N-1} - \bar{x}_N) \\
&= (x_N - \bar{x}_N)^2 + (\bar{x}_N - x_N)(\bar{x}_{N-1} - \bar{x}_N) \\
&= (x_N - \bar{x}_N)(x_N - \bar{x}_N - \bar{x}_{N-1} + \bar{x}_N) \\
&= (x_N - \bar{x}_N)(x_N - \bar{x}_{N-1})
\end{aligned}
$$

Esto significa que podemos calcular la varianza en una sola pasada usando el siguiente algoritmo:

```
variance(samples):
  M := 0
  S := 0
  for k from 1 to N:
    x := samples[k]
    oldM := M
    M := M + (x–M) / k
    S := S + (x–M) × (x–oldM)
  return S / (N–1)
```

En la literatura podemos encontrar un gran abanico de comparaciones de este segundo método con su versión tradicional en dos pasadas. En todos los casos se observa que ambos métodos funcionan exactamente igual y devuelven el mismo resultado.

12.3. Técnicas de resumen para el procesado aproximado de datos en flujo

Si bien los *sketches* son técnicas relativamente recientes, han tenido un impacto significativo en varios dominios especializados que procesan grandes cantidades de datos estructurados, como los datos de flujo. Como hemos explicado anteriormente, los algoritmos aproximados crean un resumen compacto de los datos observados. Cada actualización en el flujo puede causar que este resumen se modifique, de modo que en cualquier momento se puede usar el *sketch* para (aproximadamente) responder ciertas consultas sobre el flujo de datos completo. Si seleccionamos el resumen correcto o, en general, la estructura de *sketch* correcta, estos se convierte en una herramienta muy útil para el procesamiento de consultas aproximado, en el mismo sentido que un buen muestreo, o un histograma o incluso una representación en forma de *wavelet*.

En esta sección, describiremos las propiedades básicas de los *sketches*, así como el funcionamiento de dos *sketches* diferentes basados en analizar la frecuencia de aparición de los elementos dentro del flujo de datos: el Bloom filter (Bloom, 1970) y el contador Count-Min (Cormode, Muthukrishnan, 2005).

12.3.1. Propiedades básicas de los sketches

Las principales propiedades de un *sketch* son las siguientes:

- **Consultas soportadas**. Cada *sketch* se define para admitir un determinado conjunto de consultas y puede responderlas de manera aproximada mediante un procedimiento de consulta más o menos complejo.

- **Tamaño del *sketch***. En general, cualquier *sketch* tiene uno o más parámetros que determinan su tamaño. La precisión del *sketch* (es decir, la probabilidad de que las consultas se respondan correctamente) depende de estos parámetros. Es importante darse cuenta que el tamaño del *sketch* no debe depender de la cantidad de elementos que se procesarán o resumirán, sino de la cantidad de error que se quiera obtener.

- **Inicialización del *sketch***. Normalmente es trivial: el *sketch* se inicializa completamente a ceros, ya que la entrada vacía se resume (implícitamente) como el vector cero. Sin embargo, si la operación de actualización del *sketch* se define en términos de funciones *hash*, puede ser necesario inicializar estas funciones de una manera diferente.

- **Tiempo de actualización**. Comúnmente, la operación de actualización del *sketch* es muy rápida, ya que solo afecta ciertas partes del resumen y, por lo tanto, el tiempo por actualización es mucho menor que actualizar cada entrada en el *sketch*. En consecuencia, siempre obtendremos complejidades de actualización sublineales

- **Tiempo de consulta**. Cada algoritmo de *sketching* tiene su propia forma de usar el resumen para responder aproximadamente las consultas. El tiempo para hacer esto también varía: en algunos casos es lineal con respecto al tamaño del *sketch*, mientras que en otros casos puede ser mucho menor.

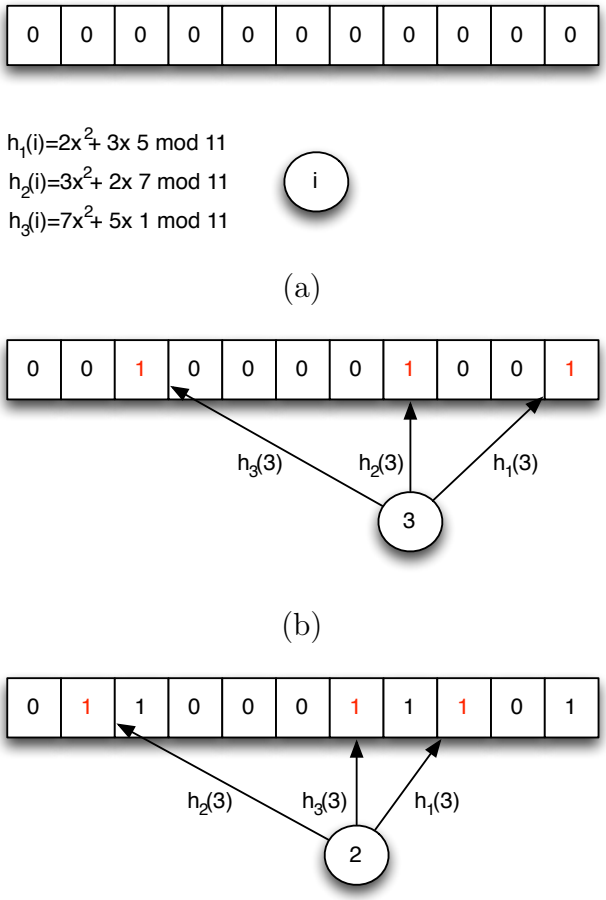
12.3.2. Bloom *filters sketches*

Ahora, presentamos uno de los *sketches* más utilizados: el Bloom filter (Bloom, 1970) como primer ejemplo de *sketch*. Un filtro de Bloom es una forma realmente compacta de representar un subconjunto $X \subset \mathcal{U}^*$ de un dominio numérico $\mathcal{U} = \{1, 2, ..., M\}$, donde denotamos $\mathcal{U}^* = \mathcal{U} \times U \times \ldots$ como un conjunto de longitud arbitraria. Un Bloom filter consiste en una cadena binaria B de longitud $w < M$ inicializada a ceros, y d funciones *hash* independientes $h_1, \ldots, h_d$ como elementos de mapeo de $\mathcal{U}$ a $\{1, 2, \ldots, w\}$. Para cada elemento i en el conjunto X, el *sketch* establece $B[h_j(i)] = 1$ para todos $1 \leq j \leq d$. Por lo tanto, cada actualización lleva tiempo igual a $O(d)$. Un elemento puede ser un solo valor o un vector. Ilustremos cómo funciona este método utilizando el siguiente ejemplo:

> Supongamos que tenemos un filtro de Bloom que contiene tres funciones *hash* diferentes ($d = 3$) y un vector de tamaño $w = 11$ como se muestra en la figura 2. Inicialmente, todos los valores son iguales a 0 (como denota la figura 2.(A)). Luego, se debe procesar un nuevo elemento $i = 3$: se calculan las funciones *hash* $h_j(i)$ y sus salidas se usan para actualizar el vector (figura 2.(B)). Posteriormente, un segundo elemento $i = 2$ llega y se repite el mismo proceso (figura 2.(C)).

En el ejemplo anterior, después de procesar las entradas $i = 3$ y $i = 2$, es posible comprobar si un elemento concreto i ha aparecido en el flujo de datos: si hay algunos $j \in \{1, 2, 3\}$ para los cuales $B[h_j(i)] = 0$, entonces el elemento no está presente, de lo contrario se concluye que el elemento i está en S. Por lo tanto, el uso de filtros de Bloom garantiza que no

Figura 2. Ejemplo de un Bloom filter *sketch*



Fuente: elaboración propia

tengamos falsos negativos, pero puede reportar falsos positivos. En (Broder, Mitzenmacher, 2003), los autores estudiaron la tasa de falsos positivos en función de $n = |X|$, w y d, asumiendo que las funciones de *hash* son completamente aleatorias. La fórmula obtenida es bastante compleja, pero la conclusión principal de ese trabajo es la siguiente: dados unos límites en n y w, para minimizar la tasa de falsos positivos, se pueden establecer valores óptimos de d. Es decir, la relación óptima entre estos tres valores es $d = (w/n) \ln 2$.

Si queremos responder consultas más complejas como, por ejemplo, «¿Cuántos elementos $i = 3$ se han procesado?» Tenemos que reemplazar el vector de mapa de bits por un vector de contadores. Luego, cuando llega un nuevo elemento i, aumentamos los contadores correspondientes en la matriz en 1, *i.e.* $B[h_j(i)] \rightarrow B[h_j(i)] + 1$, para todos $j = 1, \ldots, d$. Hay que darse cuenta que podemos gestionar la eliminación de elementos restando 1 a los contadores. En cualquier momento, para responder una consulta sobre la cantidad de veces que ha aparecido un elemento i, calculamos $\text{mín}_{1 \leq j \leq d}\{B[h_j(i)]\}$.

12.3.3. Count-min *sketch*

La idea del *sketch* count-min es muy similar a la variación del *sketch* del filtro de Bloom descrito en el último párrafo de la sección anterior, pero considerando un vector (o fila) diferente para cada función *hash* h_j.

El *sketch* count-min (Cormode, Muthukrishnan, 2005) se define como una matriz C de $d \times w$ contadores, como se describe en la figura 3. Como podemos observar, para cada una de las filas d una función *hash* independiente h_j asigna el dominio de entrada $\mathcal{U} = \{1, 2, \ldots, M\}$ uniformemente al rango $\{1, 2, ..., w\}$. Sea $X \subset \mathcal{U}^*$ el flujo de elementos que uno quiere

procesar y resumir en el *sketch* C. El *sketch* C se forma entonces como

$$C[j,k] = \sum_{1 \leq i \leq M : h(i) = k} f(i).$$

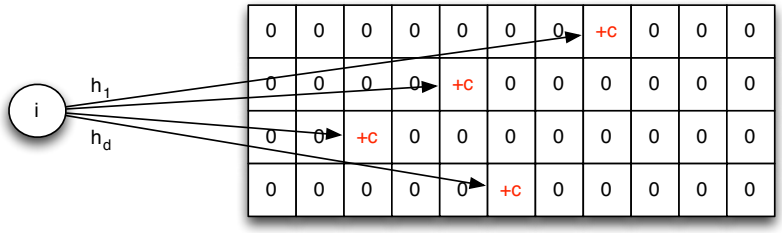
Es decir, la k-ésima columna de la i-ésima fila es la suma de las frecuencias f.[2] De todos los elementos procesados $i \in X$ que se asignan mediante la j-ésima función *hash* que evalúa k.

El tamaño del *sketch*, que se define por los valores d y w, depende de dos parámetros ϵ y δ que son seleccionados por el usuario para determinar su precisión (error de aproximación) y la probabilidad de superar los límites de precisión, respectivamente. Hay que tener en cuenta que el tamaño global del *sketch* es $d \cdot k$ y, por lo tanto, el boceto solo tiene sentido si $d \cdot k << M$.

El *sketch* count-min tiene un eficiente algoritmo de actualización: cuando el elemento $i \in X$ debe procesarse, el valor $h_j(i)$ se calcula para cada $1 \leq j \leq d$, y el valor de actualización es añadido a la entrada $C[j, h_j(i)]$ en la matriz de *sketch* C. Por lo tanto, cada actualización se procesa en el tiempo $O(d)$, ya que cada evaluación de la función *hash* lleva tiempo constante. La figura 3 ilustra este proceso: un elemento i se asigna a una entrada en cada fila j por la función *hash* h_j, y el valor de actualización c se agrega a cada entrada correspondiente . Es importante darse cuenta que asignamos $c = 1$ si solo estamos contando elementos, pero c se puede establecer en otros valores cuando el *sketch* estima otras funciones del vector de frecuencia.

[2] $f(i)$ denota la cantidad de elementos en S que tienen valor $i \in \mathcal{U}$. Por lo tanto, estos valores $f(i)$ representan un conjunto de frecuencias, y también se puede considerar que definen un vector f de dimensión $M = |\mathcal{U}|$.

Figura 3. Estructura de datos del *sketch* count-min



Fuente: elaboración propia

Finalmente, la respuesta estimada a una consulta de frecuencia para algún elemento $i \in \mathcal{U}$ se calcula como

$$\min_{1 \leq j \leq d} \{C[j, h_j(i)]\}.$$

Indicaremos el resultado de dicha consulta como $C(i)$.

Es posible encontrar más detalles sobre las funciones que se pueden calcular en este tipo de *sketches*, así como un estudio completo sobre las colisiones de *hash* y cómo seleccionar los parámetros de rendimiento ϵ y δ en (Cormode, Muthukrishnan, 2005).

En este libro no detallaremos otro tipo de *sketches*. Pero es importante conocer que hay muchos otros tipos, como el descrito en (Alon *et al.*, 1996), que permite calcular estadísticos como el promedio o la varianza.

Parte V

Procesamiento de grafos

Capítulo 13

Representación y captura de grafos

En los últimos años, la representación de datos en formato de red ha experimentado un importante auge a todos los niveles. Este formato permite representar estructuras y realidades más complejas que los tradicionales datos relacionales, que utilizan el formato de tuplas. En un formato semiestructurado cada entidad puede presentar, al igual que los datos relacionales, una serie de atributos en formato numérico, nominal o categórico. Pero además, el formato de red permite representar de un modo más rico las relaciones que puedan existir entre las distintas entidades que forman el conjunto de datos. Un claro ejemplo de esta situación lo presentan las redes sociales.

La literatura utiliza los términos «red» y «grafo» de manera indistinta. Generalmente podemos encontrar referencias a redes o a grafos sin apenas matices, aunque existe una sutil diferencia entre ambos términos: se recomienda emplear el término «red» para referirse a la entidad en el mundo real que

estamos describiendo y «grafo» para referirse a su representación matemática (Barabási, Pósfai, 2016).

> Cuando hablamos de Twitter como una red social en el mundo real donde los usuarios envían microtextos, emplearemos el término «red». Por el contrario, si descargamos parte (o la totalidad) de sus datos y lo representamos (generalmente en un computador), hablaremos de «grafo».

En este capítulo revisaremos los tipos básicos de grafos, así como sus formas más habituales de captura y representación.

13.1. Conceptos básicos de grafos

En esta sección introduciremos la definición y notación básica de la teoría de grafos. Los grafos son la forma más natural de representación de las redes reales, y es en este sentido en el que necesitamos introducir los conceptos básicos para poder representar las redes reales (Pérez-Solá, Casas-Roma, 2016).

Un grafo es una pareja de conjuntos $G = (V, E)$, donde $V = \{v_1, v_2..., v_n\}$ es el conjunto de nodos o vértices y $E = \{e_1, e_2, \ldots, e_m\}$ es un conjunto de aristas que unen dos nodos $e_i = \{v_i, v_j\}$ de forma bidireccional, es decir, el nodo v_i está conectado al nodo v_j y viceversa. En este caso, hablamos de **grafos no dirigidos**, **bidireccionales** o **simétricos**.

Cuando las relaciones no son bireccionales, hablaremos de **grafos dirigidos**, unidireccionales o asimétricos. En este caso, se representa el grafo como pareja de conjuntos $G = (V, A)$, donde es el conjunto de nodos o vértices, igual que en el caso anterior, y $A = \{a_1, a_2, \ldots, a_m\}$ es un conjunto de arcos que unen dos nodos $a_i = \{v_i, v_j\}$ de forma unidireccional, *i.e.* el nodo v_i está conectado al nodo v_j.

Se llama **orden** de G a su número de nodos, $|V|$, que por convenio es referenciado por la letra n. Asimismo, el número de aristas, $|E|$, es referenciado por la letra m y se le llama **tamaño** del grafo.

Los **nodos adyacentes** o vecinos, denotados como $\Gamma(v_i)$, se definen como el conjunto de nodos unidos a v_i a través de una arista. En este caso, el **grado** de un nodo se define como el número de nodos adyacentes, es decir, $|\Gamma(v_i)|$, aunque generalmente, el grado del vértice v_i se denota como $deg(v_i)$. La **secuencia de grados** (*degree sequence*) es una secuencia numérica de n posiciones en que cada posición i indica el grado del nodo v_i.

En un grafo dirigido G, se define a los sucesores de un nodo v_i, $\Gamma(v_i)$, como el conjunto de nodos a los cuales se puede llegar usando un arco desde v_i. Se define el grado exterior de un nodo como el número de sucesores $|\Gamma(v_i)|$. De forma similar, se puede definir a los antecesores de un nodo v_i, $\Gamma^{-1}(v_i)$, como el conjunto de nodos desde los cuales es posible llegar a v_i usando un arco. Se define el grado interior de un nodo como el número de antecesores, es decir, $|\Gamma^{-1}(v_i)|$.

En el caso de las redes sociales, los datos se suelen representar utilizando los grafos, dado que permiten una representación natural de las relaciones existentes entre un conjunto de usuarios (representados mediante «nodos» o «vértices» en el contexto de los grafos) de la red. Existen varios formatos de grafo que permiten representar cada una de las redes existentes en la realidad y que se adaptan a las particularidades de cada una de ellas. Por ejemplo, podemos encontrar redes con relaciones simétricas, como por ejemplo Facebook, donde si el usuario A es amigo del usuario B, necesariamente B es amigo de A. Por otro lado, podemos encontrar ejemplos de re-

des asimétricas, como por ejemplo Twitter, que se representan mediante grafos dirigidos o asimétricos, y donde la relación de «seguir» del usuario A hacia un usuario B no tiene por qué ser recíproca.

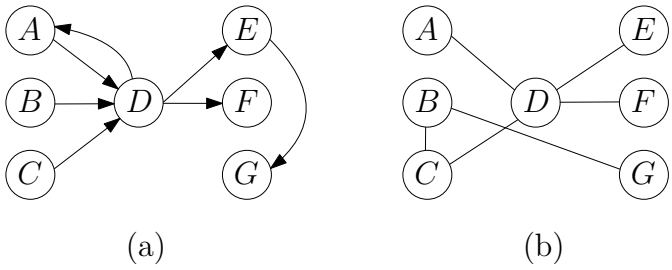
Representación mediante grafos

Vamos a suponer que deseamos modelar las relaciones entre los usuarios de Twitter, mostrando la relación «seguir» que se establece entre dos usuarios de la red. Para representar la información que nos interesa, podemos utilizar un grafo dirigido o asimétrico, en donde creamos un arco entre los nodos A y B si el usuario A «sigue» al usuario B. La figura 1 (a) muestra un posible grafo donde podemos ver que los usuarios A, B y C «siguen» al usuario D. Por su parte, el usuario D «sigue» a los usuarios A, E y F.

A continuación veremos como modelar las relaciones en una red simétrica o no dirigida, como puede ser, por ejemplo, Facebook. En esta red los usuarios establecen relaciones de «amistad» bidireccionales, es decir, si un usuario A es «amigo» de un usuario B, entonces implícitamente el usuario B también es «amigo» del usuario A. La figura 1 (b) muestra un posible grafo de ejemplo en el que vemos las relaciones de amistad entre siete usuarios. Podemos ver que el usuario A es amigo de D, el cual es también amigo de A y de C, E y F.

En estos casos, la propia estructura del grafo contiene información de gran utilidad para el análisis y estudio de las redes.

Figura 1. Representación de un grafo dirigido o asimétrico (a) y un grafo no dirigido o simétrico (b)



Fuente: elaboración propia

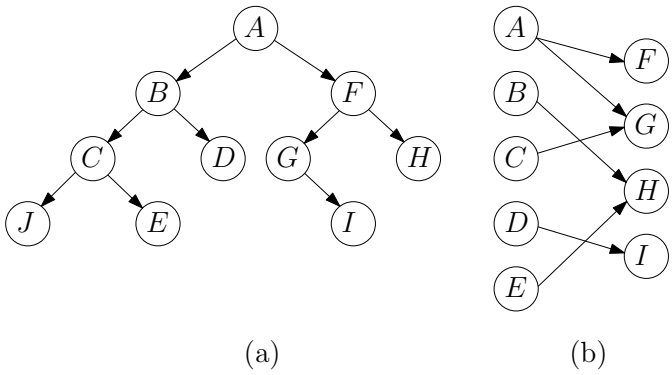
13.2. Tipos de grafos

Los grafos permiten representar y resolver muchos problemas habituales encontrados en muchos ámbitos del día a día, más allá de las ciencias de la computación. Existen multitud de tipos de grafos, que permiten representar de forma muy adecuada a distintos tipos de redes reales. Queda fuera del alcance de este libro la revisión de todos los tipos de grafos existentes, y solo revisaremos algunos de los más utilizados en procesos de análisis y aprendizaje automático (Newman, 2010; Barabási, Pósfai, 2016).

13.2.1. Árbol

Un árbol es un grafo que no tiene caminos cerrados en el que existe exactamente un camino entre cada par de puntos, como se puede ver en el ejemplo de la figura 2 (a). Los árboles son un tipo de grafo que permiten resolver gran cantidad de problemas habituales relacionados con estructuras jerárquicas.

Figura 2. Grafo en forma de árbol (a) y grafo bipartito (b)



Fuente: elaboración propia

Técnicas de aprendizaje automático basadas en la construcción de estructuras en árbol, como Random Forest, pueden representarse adecuadamente utilizando este tipo de grafos.

13.2.2. Grafo acíclico dirigido

Un grafo acíclico dirigido (*directed acyclic graph*, DAG) es un tipo de grafo que ya se ha estudiado en secciones anteriores, debido a su aplicación para la ejecución de tareas en Apache Spark.

La principal característica en un DAG es que dado un vértice v_i, no hay un camino de relaciones que vuelvan a v_i en ningún momento. Es decir, no se producen ciclos. Así, en un DAG existe una dirección definida, con un inicio y un final determinados.

13.2.3. Grafo bipartito

Un grafo bipartito es aquel en el que los nodos pueden separarse en dos grupos, nodos de inicio y nodos finales, tal y como muestra la figura 2 (b).

Si el número de elementos en cada subgrupo es idéntico nos referiremos a él como un grafo bipartito balanceado. Si todos los elementos de cada grupo se relacionan con todos los elementos del segundo grupo, diremos que el grafo es completo.

13.3. Captura de datos en formato de grafos

Los procesos de captura involucrados en datos semiestructurados en formato de grafos presentan una peculiaridad importante respecto a los procesos de captura vistos hasta ahora. En los procesos de captura vistos en los capítulos anteriores, el sistema (ya sea *batch* o *streaming*) captura paquetes de datos que, en general, son independientes los unos de los otros. Es decir, los datos son registros con una serie de atributos, cada uno de los cuales tiene su propia entidad y puede no tener ninguna relación con los demás registros, ya sean antecesores y sucesores en el conjunto de datos.

En el caso de los grafos, la relación entre las entidades (representados por nodos o vértices) es una información de gran importancia. Podríamos llegar a afirmar que las relaciones son igual de importantes que los atributos de los propios registros. Por lo tanto, es especialmente relevante capturar esta información.

13.3.1. Captura a través de API

Un caso paradigmático en la captura de datos en formato de grafos son las redes sociales. Estas presentan información de millones de usuarios, que incluye atributos propios de cada individuo (como por ejemplo, su fecha de nacimiento, aficiones, gustos musicales, etc.) y el conjunto de relaciones que establece con los demás usuarios (por ejemplo, relación de «amistad» o «seguir»). Además, puede contener mucha otra información, como por ejemplo comentarios o *like* sobre información publicada por otros usuarios. Toda esta información suele estar distribuida entre múltiples servidores y la forma de acceder suele ser a través de una API.

Las interfaces de programación de aplicaciones o API (Application Programming Interface),[1] son el conjunto de funciones y procedimientos que ofrece un sistema para ser utilizado por un programa externo con la finalidad de extraer los datos para su posterior análisis y generación de informes. Estas consultas se pueden personalizar con multitud de parámetros, como el idioma de los mensajes, la fecha y región de publicación, la persona que lo escribió, si pertenecen a una categoría, si contienen una etiqueta, etc.

La API de Twitter permite crear aplicaciones con las que seleccionar de forma automática información de los usuarios de Twitter, de sus seguidores o de los usuarios a los que sigue, de los *tweets* que ha publicado, etc.

En este caso, Twitter ofrece un conjunto de funciones y procedimientos que permiten acceder a los datos de usuarios, *tweets* y demás información, desde distintas aplicaciones y lenguajes de programación. Para poder utilizar la API

[1] http://cort.as/-EngK

es necesario tener una cuenta en la plataforma y autenticarse ante los servidores de Twitter.

13.3.2. Formato de los datos

Aunque no existe una forma única para la transferencia de datos a través de API, una de las más empleadas en la actualidad es el formato JSON.

JSON (JavaScript Object Notation)[2] es un estándar abierto basado en texto, diseñado para el intercambio de datos legibles y que permite representar estructuras de datos simples y listas asociativas.

Como hemos visto en los capítulos anteriores, si la velocidad de recogida de los datos puede ser superior a la velocidad de preprocesamiento de estos, es recomendable el uso de almacenamiento temporal, para evitar la pérdida de datos.

En función del formato que utilice la API se pueden emplear sistemas de ficheros o bases de datos NoSQL de tipo documental o clave-valor. Por ejemplo, en el caso de ficheros JSON para la transferencia de datos, la opción de un sistema de ficheros distribuido podría ser una buena opción, aunque el uso de una base de datos NoSQL de tipo documental puede ofrecernos funcionalidades superiores.

13.3.3. Métodos de captura de datos

Generalmente, ninguna API permite acceder a «todos» los datos a partir de una única consulta, y suele ser habitual acceder a los datos con un nivel de granularidad más pequeño.

[2]https://www.json.org/

La API de Twitter permite acceder a los datos a partir de diferentes tipos de preguntas. Por un lado podemos acceder a toda la información de un determinado usuario (datos del perfil, lista de *tweets*, lista de seguidores o *followers*, etc.) y, por otro lado, también podemos acceder a información sobre contenidos (por ejemplo, los *tweets* de un determinado tema o publicaciones en un determinado espacio geográfico).

El esquema general de consulta a API de datos, especialmente en el caso de redes sociales, suele seguir el siguiente esquema:

1. Se escoge un nodo inicial (llamado semilla o *seed*) y se coloca en una lista de nodos pendientes de visitar.

2. Se escoge el primer elemento de la lista de nodos pendientes de visitar y se realiza la consulta de todos sus datos, incluyendo la lista de nodos adyacentes (es decir, nodos a distancia 1).

3. Se almacenan todos los nodos adyacentes en la lista de nodos pendientes de visitar, evitando duplicados.

4. Se ordena la lista de nodos pendientes de visitar. En este sentido, existen multitud de estrategias para ordenar esta lista, y depende del tipo de exploración que se quiere realizar el escoger un u otro método. Algunos de los más básicos son:

    - Cola o lista FIFO (*first in, first out*), que simplemente aplica como criterio el orden en que llegaron los datos.

- Pila o lista LIFO (*last in, first out*), donde el último en llegar es el primero en ser procesado.
- Aplicando alguna métrica que determine la importancia de los nodos en relación a la red.

  Por ejemplo, se podría aplicar el grado de cada nodo para determinar su «importancia» dentro del grafo, con el objetivo de que los nodos más importantes se procesen antes.

- Se repite el proceso a partir del paso 3, hasta que la lista de nodos pendientes de visitar esté vacía.

En el caso de una red pequeña, con centenares o miles de usuarios, es posible aplicar este proceso de forma rápida y con un consumo bajo de recursos (tiempo, cálculo y espacio de almacenamiento). Pero en el caso de redes grandes, con cientos o miles de millones de usuarios, este es un proceso terriblemente complejo.

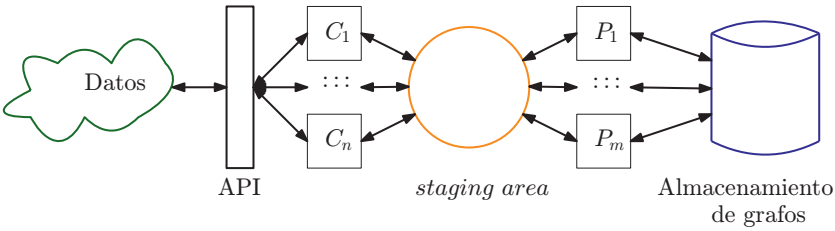
Por un lado, es muy difícil conseguir todos los datos de una red, ya que la lista de usuarios pendientes de visitar se multiplica de forma exponencial cada vez que se visita un usuario (debido a que el grado en las redes sociales suele ser elevado) y es, por lo tanto, muy difícil obtener todos los datos en la práctica. Además, hay que tener en cuenta que la estructura de la red evoluciona con el tiempo, con lo cual lo datos con cierta «antiguedad» dejan de tener validez rápidamente.

13.3.4. Ejemplo de arquitectura para captura de datos de redes

Aunque cada problema tiene ciertas particularidades que lo hacen especial, y no existen soluciones generales para to-

dos los problemas, a continuación presentamos un ejemplo de arquitectura que permite capturar datos de una API, preprocesarlos y almacenarlos de forma distribuida.

Figura 3. Ejemplo de arquitectura para la captura y preprocesamiento de datos en formato de grafos



Fuente: elaboración propia

La figura 3 ejemplifica lo que hemos descrito en este capítulo. En primer lugar, los datos son accedidos a través de llamadas iterativas a una API. Es posible tener un conjunto de procesos independientes realizando esta tarea (etiquetados como $C_1, \ldots, C_n$ en la figura), siempre que de alguna forma, mantengan sincronización sobre la lista de usuarios pendientes de visitar. A continuación la información es depositada en un almacenamiento temporal, donde permanece a la espera para ser leída por los procesos encargados del preprocesamiento de los datos ($P_1, \ldots, P_m$). Estos leen los datos almacenados temporalmente y realizan las acciones necesarias para consolidar los datos en el sistema de almacenamiento final, que los guarda para su posterior análisis.

Capítulo 14

Almacenamiento de grafos

El almacenamiento de grafos es un problema complejo, actualmente aun sin una solución eficiente para grandes volúmenes de datos. El principal problema radica en que los nodos no son independientes entre sí. La solución más básica consiste en almacenar todos los datos en un mismo sistema (escalabilidad vertical). En este caso, las relaciones entre nodos se almacenan todas juntas. Cualquier cambio (por ejemplo, eliminar o crear una nueva arista o arco) se propaga de forma atómica a ambos nodos afectados por el cambio.

Por el contrario, cuando no es posible almacenar todos los datos en un mismo sistema, deberemos dividir los datos entre distintos sistemas de un *cluster* (escalabilidad horizontal). En este caso, el principal problema que aparece está relacionado con el almacenamiento de las relaciones (aristas o arcos) del grafo. Cualquier cambio en la estructura del grafo (por ejemplo, eliminar o crear una nueva arista o arco) se debe enviar a todos los sistemas de almacenamiento que tengan alguno de los nodos involucrados en el cambio, y cerciorarse de que se han recibido y aplicado correctamente. En caso contrario, se

produciría una situación de inconsistencia de datos, donde la información proporcionada por sistemas distintos podrían ser contradictorias.

En este capítulo veremos como podemos representar y almacenar la información contenido en grafos empleando, por un lado, sistemas de ficheros (distribuidos o no), y por otro lado, utilizando bases de datos NoSQL en grafo.

14.1. Almacenamiento en ficheros

El grafo es una herramienta con una potente representación visual para entender relaciones entre nodos, sin embargo las dependencias entre ellos para procesarlos requiere de una notación matemática, que ayude a su procesado desde un punto de vista algorítmico.

En este sentido, existen multitud de formatos para representar los grafos a partir de ficheros de texto. En general, todos ellos están basados en las dos formas principales de representar los grafos de una forma matemática: la matriz y la lista de adyacencia.

14.1.1. Matriz de adyacencia

Dado un grafo G, su matriz de adyacencia $A = (a_{i,j})$ es cuadrada de orden n y viene dada por:

$$a_{i,j} = \begin{cases} p & \text{si } \exists(v_i, v_j) \\ 0 & \text{caso contrario} \end{cases} \tag{14.1}$$

Así, siendo n el número de nodos en un grafo, podemos representarlo con una matriz de n^2 elementos. Si un elemento dado v_i está unido al elemento v_j por una arista de peso

p, el elemento $m_{i,j}$ de la matriz toma el valor p. Para cualquier par de elementos que no están unidos por una arista, su correspondiente valor en la matriz es igual a 0.

La matriz de adyacencia del grafo G, representada en la figura 1 (a), es:

$$A(G) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{14.2}$$

Notese que la matriz de adyacencia de un grafo no dirigido es simétrica, mientras que no tiene por qué serlo en el caso de un grafo dirigido.

Desde un punto de vista computacional la matriz permite:

- Algoritmo de complejidad $O(1)$ para identificar si dos nodos están conectados.
- Para matrices densas, el consumo de memoria es un problema de $O(n^2)$. Las matrices que representan grafos con pocas conexiones entre nodos son llamadas «dispersas» (*spare*). Es decir, la mayoría de elementos de la matriz son 0.

14.1.2. Lista de adyacencia

Una alternativa interesante para representar un grafo es mediante la lista de adyacencia. En esta estructura asocia a cada vértice v_i una lista que contiene todos los vértices adyacentes

$v_j|\{v_i, v_j\} \in E$. En el caso de un grafo dirigido, se asocia a cada vértice v_i una lista que contiene todos los vértices accesibles desde él, es decir, $v_j|(v_i, v_j) \in A$. La principal ventaja de este método de representación es el ahorro de espacio para almacenar la información.

Supongamos que queremos almacenar un grafo no dirigido de 100 vértices y 1000 aristas ($n = 100$ y $m = 1000$). Si utilizamos la matriz de adyacencia necesitaremos una matriz cuadrada de $n \times n$, es decir, 10^4 posiciones. En el caso de emplear la matriz de incidencia deberemos reservar $n \times m = 10^5$ posiciones. Finalmente, utilizando la lista de adyacencia requeriremos una posición para cada vértice del grafo y otras dos para cada arista (debido a que representamos la adyacencia entre ambos vértices), es decir, $n + 2m = 2{,}100$.

Esta representación cuenta con las siguientes propiedades:

- Permite una fácil iteración entre nodos relacionados.
- Las listas tienen longitud variable.
- Requiere menos memoria que la matriz de adyacencia para ser procesada, siendo proporcional a la suma de nodos y aristas.

Desde un punto de vista programático, una lista de adyacencia es una estructura que puede implementarse utilizando *arrays*, ya que se conoce *a priori* el número de elementos y la longitud de cada *array*.

14.2. Bases de datos NoSQL en grafo

El modelo en grafo difiere totalmente de los modelos de agregación. En este modelo los datos no se almacenan mediante agregados, sino mediante grafos.

Como en el resto de modelos de datos NoSQL, no hay un modelo en grafo estándar. En nuestro caso vamos a suponer que el modelo en grafo responde a un grafo de propiedades etiquetado. Bajo esta asunción, los modelos en grafo están compuestos de nodos, aristas, etiquetas y propiedades:

Los **nodos** o **vértices** son elementos que permiten representar conceptos generales u objetos del mundo real. Vendrían a ser el equivalente a las relaciones en el modelo relacional. No obstante, hay una diferencia importante: en el modelo relacional las relaciones permiten representar conceptos («Actor» por ejemplo) y sus filas permiten representar instancias de los mismos, es decir objetos del mundo real («Groucho Marx» por ejemplo). Por lo tanto, los conceptos y los objetos del mundo real se representan mediante construcciones distintas. No obstante, en el modelo en grafo, los conceptos y los objetos del mundo real se pueden representar de la misma forma: mediante nodos. Es posible que no haya una diferenciación semántica de los mismos a nivel de modelo.

Las **aristas** o **arcos** son relaciones dirigidas que nos permiten relacionar nodos. Las aristas representan relaciones entre objetos del mundo real y serían el equivalente a las claves foráneas en el modelo relacional o a las asociaciones en los diagramas de clases de UML.

Las **etiquetas** son cadenas de texto que se pueden asignar a los nodos y a las aristas para facilitar su lectura y proveer mayor semántica.

Las **propiedades** son parejas <clave, valor> que se asignan tanto a nodos como a aristas. La clave es una cadena de caracteres, mientras que el valor puede responder a un conjunto de tipos de datos predefinidos. Las propiedades serían el equivalente a los atributos en el modelo relacional.

A pesar de que los nodos pueden emplearse para representar conceptos y objetos del mundo real, algunos sistemas gestores de bases de datos NoSQL en grafo, como por ejemplo Neo4J, utilizan las etiquetas para identificar los conceptos a los que pertenecen los objetos del dominio. Así pues, si tenemos el nodo que representa la asignatura de este curso, este nodo puede etiquetarse con la etiqueta «Asignatura» para indicar que el nodo es de tipo asignatura. En estos materiales se utilizará esta estrategia para representar los conceptos de la base de datos.

La característica principal del modelo en grafo es que las relaciones están explícitamente representadas en la base de datos. Eso simplifica la recuperación de elementos relacionados de forma muy eficiente. Esta eficiencia es mayor que en modelos de datos relacionales, donde las relaciones entre datos están representadas de forma implícita (claves foráneas) y requieren ejecutar operaciones de combinación (en inglés *join*) para calcularlas.

Además, la definición de relaciones es más rica en el modelo en grafo, debido a que pueden asignarse propiedades directamente a las aristas. Eso permite que las relaciones entre datos contengan propiedades, proveyendo una representación más natural, clara y eficiente de las mismas.

El modelo en grafo impone pocas restricciones de integridad. Básicamente dos, siendo la segunda consecuencia de la primera:

1. No es posible definir aristas (arcos) sin nodos origen y/o destino.

2. Los nodos solo pueden eliminarse cuando quedan huérfanos. Es decir, cuando no haya ninguna arista que entre o salga de estos.

Al igual que los otros modelos de datos NoSQL vistos, este modelo es *schemaless*. Por lo tanto, no es necesario definir un esquema antes de empezar a trabajar con una base de datos en grafo. No obstante, en este caso, hay parte del esquema implícitamente definido en los grafos. Por ejemplo, los arcos que representen el mismo tipo de relación acostumbran a utilizar una etiqueta común para indicar su tipo y semántica (al igual que los nodos). Como consecuencia de la característica anterior, este modelo permite añadir nuevos tipos de nodos y aristas fácilmente, y sin realizar cambios en el esquema. Aunque es conveniente hacerlo de forma ordenada y planificada, para evitar añadir tipos de relaciones o nodos redundantes o mal etiquetados.

Los sistemas de gestión de bases de datos que implementan un modelo en grafo acostumbran a proporcionar lenguajes de más alto nivel que los basados en modelos de agregación. Por ejemplo, podemos encontrar el uso de Cypher[1] en Neo4j, que es un lenguaje declarativo parecido a SQL, y Gremlin.[2]

Ejemplos de bases de datos NoSQL basadas en este modelo son Neo4j, OrientDB, AllegroGraph y TITAN.

[1]Es un lenguaje de consulta y manipulación de grafos creado soportado por Neo4j. Más información: http://cort.as/-FlrZ

[2]Es un lenguaje de dominio de gestión de grafos y que está soportado por varios sistemas gestores de bases de datos en grafo. Más información: http://cort.as/-Flre

Capítulo 15

Análisis de grafos

Las herramientas y tecnologías para el análisis de *big data* o datos masivos se encuentran en un momento de alta ebullición. Continuamente aparecen y desaparecen herramientas para dar soporte a nuevas necesidades a la vez que las existentes van incorporando nuevas funcionalidades. Por eso, el objetivo principal de este módulo no es enumerar las tecnologías y herramientas disponibles, sino proporcionar las bases teóricas necesarias para entender este complejo escenario y ser capaz de analizar de forma propia las distintas alternativas existentes.

Aun así, sí que veremos con cierto grado de detalle las principales características de algunas de las herramientas más importantes y estables dentro de este complejo escenario, como pueden ser las principales herramientas del ecosistema de Apache Hadoop o las librerías de Apache Spark.

En este módulo revisaremos las principales técnicas relacionadas con el análisis de grafos, así como sus implicaciones cuando trabajamos con grandes volúmenes de datos. El estudio de este tipo de datos requiere de algoritmos específicos, que

permitan explotar la capacidad de expresión de este formato de datos. En esta sección veremos los principales algoritmos de análisis de datos en formato de grafos, que son conocidos como minería de grafos (*graph mining*).

15.1. Procesamiento de grafos

Los grafos son una importante herramienta para la resolución de problemas de procesamiento complejo. Así, al representar un problema en forma de grafo podemos aplicar algunos algoritmos de manejo de estos.

A continuación vamos a presentar algunos de ellos que, en muchas ocasiones, constituyen subrutinas incluidas en tareas de procesamiento más complejas. Los algoritmos presentados es esta sección son los elementos que ayudarán a tareas de procesado más complejas sobre grafos, de forma similar a cómo las tareas de ordenación de series contribuyen a las tareas de procesamiento que manejan largas colecciones de datos.

15.1.1. Algoritmos para el recorrido de grafos

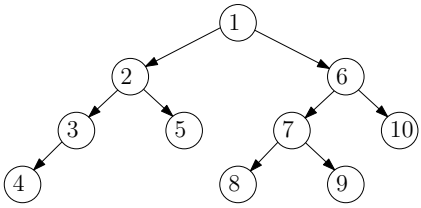
Para examinar la estructura de un grafo, es común tener que realizar un recorrido por todo el grafo. Las dos estrategias más comunes para tal fin son:

- DFS (*depth first search*) o búsqueda en profundidad prioritaria. Este método prioriza la apertura de una única vía de exploración a partir del nodo actual.
- BFS (*breadth first search*) o búsqueda en anchura prioritaria. Este método prioriza la búsqueda en paralelo de todas las alternativas posibles desde el nodo actual.

Búsqueda en profundidad prioritaria

En la búsqueda en profundidad prioritaria (DFS) se recorre un grafo desde cada nodo hasta el último nodo de una rama. Una vez se llega al final, vuelve hacia atrás hasta encontrar otra rama en el siguiente nodo. Es una búsqueda ordenada tal y como muestra la figura 1.

Figura 1. Búsqueda en profundidad. La numeración de los nodos corresponde a su orden en el proceso de búsqueda



Fuente: elaboración propia

Desde un punto de vista de programación, es un patrón utilizado muy habitualmente para recorrer arquitecturas jerárquicas, como podría ser un árbol de directorios. Este tipo de algoritmos tienen un marcado carácter recurrente, lo que facilita la implementación utilizando dichos patrones secuenciales de diseño e ingeniería de *software*. La figura 1 muestra un ejemplo de búsqueda en profundidad prioritaria.

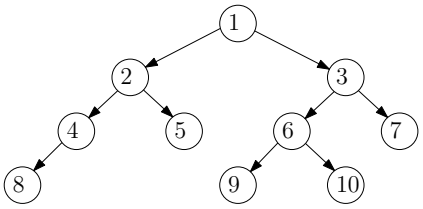
Análisis de paralelismo: DFS tiene una importante componente de ordenación recursiva, recorriendo cada rama del grafo antes de recorrer el siguiente. Esta componente compromete su capacidad de desarrollar algoritmos de procesamiento de grafos en paralelo, ya que el «etiquetado» propio de la ordenación obliga a un sincronismo entre tareas que dificultan su procesado. Para resolver dicha dificultad, existen propuestas de DFS no ordenado.

En este caso, el principal objetivo es visitar todos los nodos y verificar sus relaciones. Sin embargo, si la ordenación no es un requisito de nuestro análisis, podríamos utilizar estructuras de monitorización de nodos visitados y nodos por visitar, de modo que cada tarea paralela puede recorrer aquellas que quedan por visitar.

Búsqueda en anchura prioritaria

Contrariamente a la búsqueda en profundidad prioritaria, en este algoritmo recorremos solo los nodos adyacentes a cada nodo. Es decir, no recorremos todo el camino hasta los vértices terminales, sino que lo recorremos en anchura y profundizando en cada nueva iteración, tal y como muestra la figura 2.

Figura 2. Búsqueda en anchura. La numeración de los nodos corresponde al orden en que el algoritmo los recorre



Fuente: elaboración propia

Análisis de paralelismo: El número de vértices es finito, de modo que a partir de un nodo inicial, vamos a recorrer todos los nodos que se van encontrando por niveles. Siguiendo con el ejemplo de la figura, recorremos el nodo inicial (primer nivel), a continuación los nodos conectados a este (segundo nivel), y así sucesivamente hasta completar el árbol hasta su máxima profundidad.

Este es un proceso iterativo que permite más paralelismo, ya que diferentes procesos se pueden ir ejecutando en paralelo y verificando que cada vértice se visita una vez y se le asigna una profundidad mediante una etiqueta. Desde un punto de vista de diseño de programación, podríamos utilizar hilos de procesamiento (*threads*) que analicen las relaciones y vayan etiquetando/marcando aquellos nodos a medida que se recorren, utilizando estructuras de monitorización y almacenamiento intermedio, tales como colas, para etiquetar los nodos visitados y nodos pendientes.

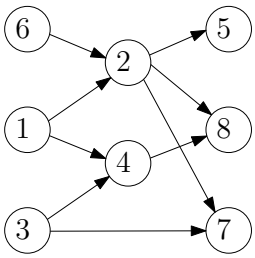
15.1.2. Ordenación topológica

La ordenación topológica de grafos (*topological sort*) es un concepto importante, especialmente en grafos del tipo DAG (directos y sin ciclos) ya que cada nodo viene precedido de otra tarea y es requisito de la siguiente. Mediante la ordenación topológica se identifica la dirección y secuencia del grafo.

Un nodo sin una arista de entrada es un nodo origen, mientras que un nodo sin ninguna arista de salida es un nodo final.

En un grafo puede haber múltiples ordenaciones topológicas válidas. Por ejemplo, en el grafo de la figura 3, las ordenaciones $\{6, 1, 3, 2, 7, 4, 5, 8\}$ y $\{1, 6, 2, 3, 4, 5, 7, 8\}$ son ambas válidas. La complejidad de la ordenación topológica puede llegar a ser alta. Una ordenación topológica hace uso de un algoritmo de recorrido de un árbol, sin embargo, al ser la ordenación un factor importante, en este caso se utilizarían técnicas de DFS, siendo un problema de difícil paralelización.

Figura 3. Grafo DAG sobre el que realizar una ordenación topológica



Fuente: elaboración propia

15.1.3. Árbol recubridor mínimo

El árbol recubridor mínimo (*minimum spanning tree*, MST) es aquel camino mínimo que, utilizando el menor número de vértices (o camino en el que la suma de pesos es menor), recorre todos los nodos del grafo (véase figura 4).

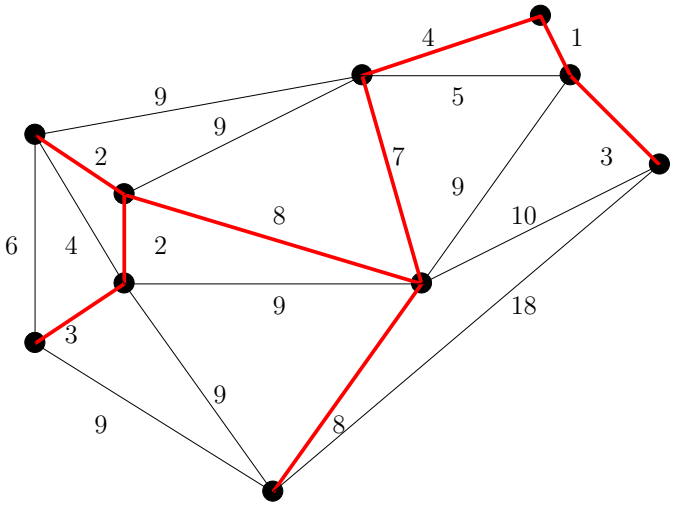
Este tipo de operaciones sobre grafos es de especial importancia en escenarios como la minimización de distancias y longitudes en redes de telecomunicaciones o la optimización de potencias y consumos en redes de energía eléctrica.

Existen varios algoritmos para resolver este tipo de problemas:

- Algoritmo de Kruskal: recorre cada arista a partir de un nodo, las ordena por peso y selecciona las de menor coste con dos premisas: (1) no repetir nodos y (2) no hacer ningún bucle hasta el final.

- Algoritmo de Prim: utiliza la estrategia de buscar las distancias mínimas de cada nodo. Empezando por un vértice inicial arbitrario y va añadiendo vértices al recorrido, tomando aquellos que están a menor distancia.

Figura 4. Ejemplo de MST de un grafo



Fuente: elaboración propia

Cuando ya no quedan más nodos por añadir, entiende que ha finalizado.

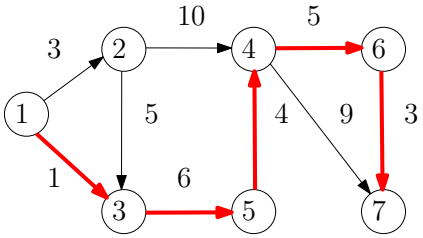
15.1.4. El camino más corto

El algoritmo del camino más corto (*shortest path*, SP) es aquel que encuentra la distancia mínima entre dos vértices.

Existen, una vez más, diferentes algoritmos para resolver este problema, siendo el algoritmo de Dijkstra uno de los más populares.

En este caso, el algoritmo recorre todos los nodos del grafo ordenadamente, añadiendo a una lista aquellos que ya ha visitado y eliminándolos de otra lista en la que se incluyen todos los nodos pendientes de visitar. Además, va añadiendo en una tercera lista solo aquellos que suponen una menor distancia entre dos nodos. A medida que recorre todos los nodos va eli-

Figura 5. Grafo con el camino más corto marcado, en el cual la suma de pesos es mínima para llegar del vértice inicial (1) al vértice final (7)



Fuente: elaboración propia

minando elementos de la lista. Una vez ha completado todos los pasos, devuelve la lista con el o los caminos más cortos entre los vértices inicial y final (véase figura 5).

15.1.5. Importancia de los vértices

Uno de los problemas de gran relevancia dentro del análisis de grafos es determinar la importancia relativa de cada uno de los vértices del grafo. De esta forma se pueden identificar los actores clave en un determinado escenario, proporcionando información muy relevante para ciertas áreas de negocio, como por ejemplo las áreas de *marketing* o publicidad.

Existen multitud de métricas para evaluar la importancia de los vértices (Pérez-Solá, Casas-Roma, 2016), cada una de ellas basadas en alguna característica concreta. Por ejemplo, se puede evaluar a partir del grado del nodo (asumiendo que los nodos con grado superior —llamados *hubs*— son claves para el flujo de la información en el grafo) o del coeficiente de centralidad (calculado a partir del número de caminos más cortos del grafo de los que forma parte cada nodo).

Más allá de este conjunto de métricas, existen algoritmos más complejos para determinar la importancia de los nodos.

Uno de los algoritmos más relevantes es el PageRank,[1] desarrollado por Google para indicar la importancia de una determinada página web dentro de la WWW.

Algoritmo PageRank

En el siguiente ejemplo, consideraremos 4 páginas que se referencian entre sí (página A, B, C y D). Cada una contribuye a las otras con su peso, según los valores presentados en la figura 6, dividido por el número de referencias a las otras. Por ejemplo, si una página tiene un peso de 1,0 y dos referencias a otras páginas, entonces contribuye con 0,5 a cada una de ellas.

Las contribuciones se ajustan por el factor de *damping*,[2] un ajuste estadístico que tiene un valor aproximadamente de 0,85.

Así, en cada iteración i, el peso de cada página se actualiza como su peso P_i, menos el factor de *damping* d, más la suma de los pesos del resto de páginas que la referencian, aplicándole el factor d. Para calcular el peso de la página A en la iteración $i+1$ utilizamos la siguiente ecuación:

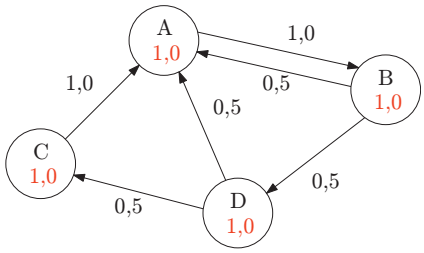
$$P_{i+1}(A) = (P_i(A) - d) + d(P_i(B) + P_i(C) + P_i(D))$$

La solución converge cuando el incremento en cada iteración es muy pequeño. El número de iteraciones depende de la precisión que deseamos obtener: cuantas más iteraciones, mayor tiempo de cálculo y mayor precisión.

[1]PageRank es un algoritmo utilizado por Google para clasificar páginas web en base a las referencias cruzadas entre ellas. La idea que subyace es que las páginas web más importantes tienen más referencias de otras páginas web.

[2]Probabilidad de que un usuario vaya a dejar de «navegar» por las páginas referenciadas entre ellas.

Figura 6. Representación gráfica del ejemplo PageRank



Fuente: elaboración propia

15.1.6. Detección de comunidades

En redes sociales y, generalmente, en cualquier tipo de red, se define informalmente una comunidad como un conjunto de nodos que se encuentran densamente conectados entre ellos y poco conectados al resto de la red. Los nodos de una misma comunidad, por su parte, acostumbran a tener alguna propiedad en común que los une y/o juegan papeles similares dentro de la red.

La detección de comunidades es un problema clásico no supervisado de agrupación (*clusteing*). Cuando se afronta un problema de detección de comunidades, el número de comunidades a obtener es normalmente un valor desconocido. Además, no suelen tener un tamaño constante y es habitual encontrar divisiones que presentan comunidades de tamaños significativamente distintos. Del mismo modo, también es común que la densidad que presenta cada comunidad como subgrupo sea distinta.

Una gran cantidad de algoritmos de detección de comunidades se basan en el agrupamiento jerárquico. Estos algoritmos intentan construir una jerarquía de grupos asignando los nodos a comunidades pequeñas, que a su vez se asignan a co-

munidades de mayor tamaño. El resultado de estos algoritmos es una agrupación multinivel que permite analizar las comunidades de un grafo con distintas granularidades. Otro conjunto de algoritmos de agrupamiento se centran en el uso de los valores y vectores propios de ciertas matrices relacionadas con los grafos para detectar las comunidades. Estos algoritmos se conocen como algoritmos de agrupamiento espectrales. La matriz laplaciana es la matriz más usada en este tipo de algoritmos. Finalmente, existen también algoritmos dinámicos basados en recorrer el grafo para extraer su estructura de comunidades. Por ejemplo, algunos algoritmos realizan *paseos* aleatorios (en inglés, *random walks*) con la idea de que pasará mucho tiempo dentro de una comunidad densa debido al gran número de caminos posibles que esta contiene.

Existen multitud de algoritmos de detección de comunidades, aunque muchos de ellos no son aplicables en casos de *big data*, dado su elevado coste computacional. Este sería el caso, por ejemplo, del algoritmo de Girvan-Newman, que solo puede ser aplicado a grafos de tamaño pequeño o mediano, ya que se basa en el cálculo de las centralidades de todas las aristas del grafo.

Un caso interesante cuando lidiamos con datos masivos es el algoritmo de propagación de etiquetas o Raghavan. El algoritmo de propagación de etiquetas (*label propagation algorithm* o LPA) es un algoritmo jerárquico. El algoritmo se inicializa asignando una etiqueta única a cada nodo del grafo e iterativamente actualiza las etiquetas de los nodos en función de las etiquetas de sus vecinos, de manera que la nueva etiqueta de un nodo es aquella que más aparece entre sus vecinos. Cuando el algoritmo converge, las etiquetas restantes identifican las diferentes comunidades detectadas.

El algoritmo de propagación de etiquetas es posiblemente el más común a la hora de detectar comunidades en grafos, ya que tiene un coste computacional bajo. No obstante, tiene dos inconvenientes relevantes:

1. La convergencia hacia una solución aceptable no está garantizada.
2. Puede converger hacia una solución trivial (cada nodo forma parte de su propia comunidad individual).

15.2. Visualización de grafos

La visualización de grafos es un área situada a caballo entre las matemáticas y las ciencias de la computación, que combina los métodos de la teoría de grafos y la visualización de la información para derivar representaciones bidimensionales de grafos. La visualización de un grafo es una representación pictórica de los vértices y las aristas que forman el grafo. Este gráfico o representación no debe confundirse con el grafo en sí mismo; configuraciones gráficas muy distintas pueden corresponder al mismo grafo. Durante los procesos de manipulación de los grafos, lo que importa es qué pares de vértices están conectados entre sí mediante aristas. Sin embargo, para la visualización de los grafos, la disposición de estos vértices y aristas dentro del gráfico afecta su comprensibilidad, su facilidad de uso y su estética.

15.2.1. Estrategias de visualización

En esta sección veremos algunas de las principales estrategias de visualización de grafos enfocadas a la distribución de

los nodos en un espacio bidimensional. Es decir, estas estrategias se centran en asignar posiciones (o coordenadas) a los nodos con la finalidad de facilitar la visualización del grafo, pero no consideran aspectos como el color o el tamaño de los nodos.

Se han definido múltiples métricas para evaluar de forma objetiva la calidad de una estrategia de visualización de grafos. Algunas de estas métricas, además, son utilizadas por las estrategias de disposición como una función objetivo que se trata de optimizar durante la colocación de los vértices, para mejorar la visibilidad de la información. Algunas de las métricas más relevantes son las siguientes:

- El número cruces se refiere al número de pares de aristas que se cruzan entre sí. Generalmente no es posible crear una disposición sin cruces de aristas, pero si se intenta minimizar el número de cruces se suelen obtener representaciones más sencillas, y por lo tanto, más fáciles de interpretar de forma visual.

- El área de representación del grafo es el tamaño del marco delimitador más pequeño. Las representaciones con un área más pequeña son, en general, preferibles a aquellos con un área mayor, ya que permiten que las características del grafo sean más legibles.

- La simetría permite hallar grupos con cierta simetría dentro del grafo dado y mostrar estas simetrías en la representación del grafo, de forma que se puedan identificar rápidamente en una observación visual.

- Es importante representar las aristas y los vértices utilizando formas simples, para facilitar la identificación

visual. Generalmente se aconseja representar las aristas utilizando formas rectas y simples, evitando las curvas innecesarias.

- La longitud de las aristas también influye en la representación de un grafo. Es deseable minimizar la longitud de las aristas y, además, mantener longitudes uniformes para todas las aristas que se representen.
- La resolución angular mide los ángulos entre las aristas que salen (o llegan) a un mismo nodo. Si un grafo tiene vértices con un grado muy grande (*hubs*), entonces la resolución angular será pequeña, pero en caso contrario se debe mantener una proporcionalidad que ayuda a la estética y legibilidad del grafo.

Existen múltiples estrategias de visualización de grafos, y de forma similar a la mayoría de escenarios, cada una posee ciertos puntos fuertes y débiles. Consecuentemente, es importante conocer estas estrategias básicas para escoger en cada momento la que mejor se adapte al planteamiento y los datos con los que se está trabajando.

1. La **disposición aleatoria** (*random layout*) es la estrategia más simple de visualización. Consiste en distribuir los vértices de forma aleatoria en el espacio bidimensional y añadir las aristas correspondientes entre los pares de vértices.
2. La **disposición circular** (*circular layout*) coloca los vértices del grafo en un círculo, eligiendo el orden de los vértices alrededor del círculo intentando reducir los cruces de aristas y colocando los vértices adyacentes cerca el uno del otro.

3. Los **diagramas de arcos** (*arc diagrams*) sitúan los vértices sobre una linea imaginaria. A continuación, las aristas se pueden dibujar como semicírculos por encima o por debajo de la línea.

4. Los sistemas de **disposición basados en fuerzas** (*force-based layout*) modifican de forma continua una posición inicial de cada vértice de acuerdo a un sistema de fuerzas basado en la atracción y repulsión entre vértices. Típicamente, estos sistemas combinan fuerzas de atracción entre los vértices adyacentes y fuerzas de repulsión entre todos los pares de vértices, con el fin de buscar una disposición en la que las longitudes de las aristas sean pequeñas y los vértices estén bien separados para facilitar la visualización.

5. La **disposición basada en el espectro** del grafo (*spectral layout*) utiliza los vectores propios de la matriz de Laplace,[3] u otras matrices relacionadas con la matriz de adyacencia del grafo, como coordenadas de los vértices en el espacio bidimensional.

6. Los métodos de **disposición ortogonal** (*orthogonal layout*) fueron diseñados originalmente para problemas de diseño de VLSI.[4] Típicamente emplean un enfoque de múltiples fases, en las que se intenta reducir el número de cruces entre aristas, trazar aristas que sean lo más

[3]Es una matriz cuadrada de orden $n \times n$ tal que $L = D - A$, donde D es la matriz diagonal que contiene la suma de filas a lo largo de la diagonal y 0 en las demás posiciones, y A es la matriz de adyacencia.

[4]Integración a escala muy grande (*very large scale integration*) de sistemas de circuitos basados en transistores en circuitos integrados.

rectas posibles y reducir el espacio de representación del grafo.

7. La **disposición basada en capas** (*layered-based layout*), a menudo también llamados de estilo Sugiyama, son los más adecuados para grafos dirigidos acíclicos o grafos casi acíclicos. En estos métodos, los vértices del grafo están dispuestos en capas horizontales, de tal manera que la mayoría de aristas se encuentran en posición vertical desde una capa a la siguiente.

Representaciones de grafos

La figura 7 muestra el resultado de los cuatro algoritmos descritos anteriormente. En primer lugar se puede ver el algoritmo aleatorio, que muestra los vértices sin ningún tipo de agrupamiento, dificultando su visibilidad. En segundo lugar vemos la disposición en círculo de los vértices. Esta disposición puede ser útil en algunos casos donde existen pocas aristas y el grafo no presenta ningún tipo de estructura interna. En tercer y cuarto lugar, podemos ver los algoritmos de escalado multidimensional[5] y basado en fuerzas.[6] La disposición de los vértices en ambos casos ayuda a una correcta visualización del grafo.

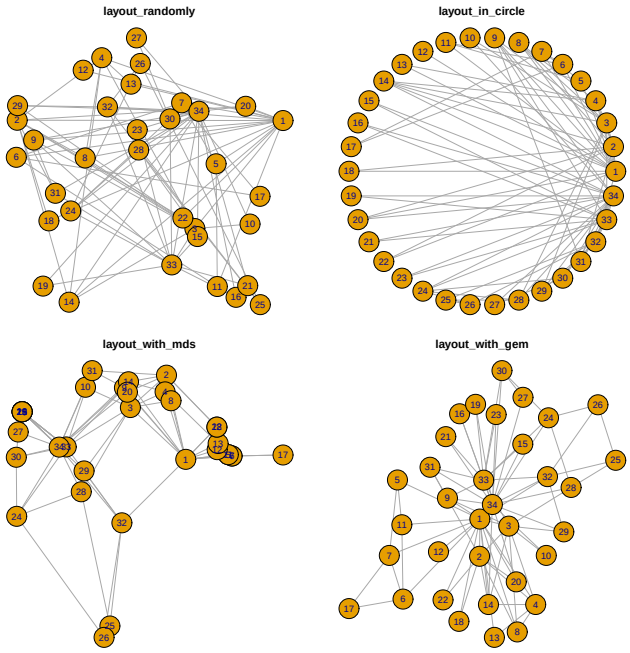
15.2.2. La problemática del orden

La representación de datos mediante gráficos presenta algunos problemas, especialmente cuando queremos representar

[5]El método de escalado dimensional sitúa los puntos de un espacio dimensional grande (≥ 3) en un espacio bidimensional, de manera que se intenta mantener la distancia entre los puntos en el espacio original.

[6]Este algoritmo sitúa los vértices en el plano utilizando la disposición basada en fuerzas del método GEM(Frick *et al.*, 1995).

Figura 7. Ejemplos de representación de un grafo



Fuente: elaboración propia

grandes volúmenes de datos. La visualización de grafos suele ser muy efectiva cuando se manejan conjuntos de datos de un tamaño razonable, pero la complejidad aumenta mucho cuando se pretenden representar grandes conjuntos de datos, es decir, grafos con un orden (número de vértices) elevado.

Existen dos metodologías básicas para tratar con grafos de orden elevado:

- Una primera alternativa consiste en representar, no el conjunto entero de todos los vértices, sino un subconjunto reducido de valores agregados a partir de los datos

originales. De esta forma se pretende reducir el tamaño de los datos y facilitar su visualización.

- Un segundo enfoque consiste en utilizar una estrategia de visualización dinámica, que permita un cierto granulado de la información. Por ejemplo, se puede visualizar el grafo en un primer momento mostrando solo los vértices más importantes (por ejemplo, los vértices con un grado elevado), y permitir al usuario focalizar la visualización en un punto del grafo concreto, mostrando los vértices de menor grado en este punto. Es decir, se trata de hacer un tipo de *zoom* de las distintas zona que permita ver los «detalles» que puedan quedar ocultos al mostrar solo los vértices más importantes.

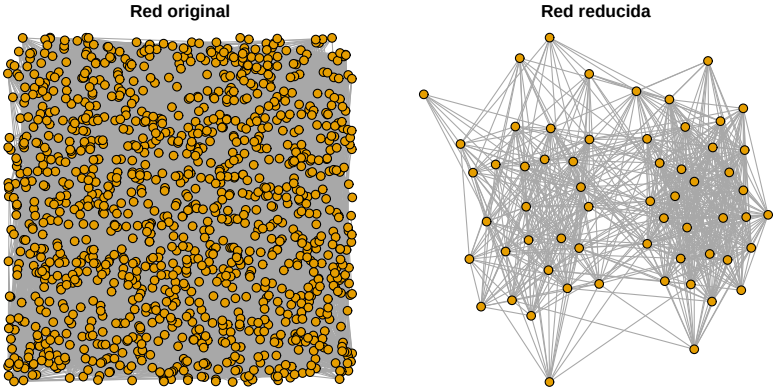
Problema de visualización de grafos grandes

La figura 8 muestra la estructura de dos grafos. Como se puede observar, la figura de la izquierda muestra un grafo de tamaño medio, con 1.224 usuarios y 16.715 relaciones entre ellos. Como se puede ver en el gráfico, es muy difícil, sino imposible, obtener información alguna de los datos presentados en esta figura. El número de nodos y aristas es demasiado grande, resultando imposible identificar la estructura del grafo, las comunidades que la componen o los nodos principales con un simple análisis visual.

Por otro lado, la figura de la derecha muestra un resumen de esta misma red social, donde solo se muestran los vértices más importantes. En este caso se ha definido la importancia de un vértice utilizando una métrica basada en el grado. Solo los vértices con grado superior a 100 son mostrados en este grafo. Obtenemos, por lo tanto, un resumen de 60 vértices y 702 relaciones entre ellos. Dado el nuevo

volumen de datos mostrados, es mucho más fácil identificar claramente la estructura del grafo; identificando las distintas comunidades que la forman y los nodos principales, es decir, los individuos que están conectados a un gran número de usuarios y que actúan como concentradores de información (*hubs*).

Figura 8. Gráfico de una red social de 1.224 vértices (izquierda) y una versión reducida que incluye los 60 vértices más importantes en función del grado (derecha)



Fuente: elaboración propia

15.3. Herramientas para datos masivos

Actualmente existen dos grandes implementaciones en el entorno *big data* para lidiar con grafos. Por un lado, encontramos Apache Giraph, que es parte del ecosistema de Apache Hadoop. Y por otro lado, tenemos GraphX, que es una librería de Apache Spark especializada en la representación y análisis

de grafos. Veremos con cierto detalle qué tipos de grafos y algoritmos soportan ambas librerías.

15.3.1. Hadoop Giraph

Apache Giraph[7] es un sistema iterativo de procesamiento de grafos creado para entornos altamente escalables. Giraph se originó como la equivalente de código abierto de Pregel (Malewicz *et al.*, 2010), la arquitectura de procesamiento de grafos desarrollada en Google. Giraph utiliza la implementación MapReduce de Apache Hadoop para procesar los grafos. Actualmente es utilizado, entre muchos otros, por Facebook para analizar el grafo social formado por los usuarios y sus conexiones.

Giraph es un motor de procesado de grafos cuyos cálculos se realizan sobre una máquina *bulk synchronous parallel* (BSP) a gran escala que se ejecuta enteramente sobre Hadoop. El modelo BSP para diseñar algoritmos paralelos no garantiza la comunicación y sincronización entre nodos. Dicho modelo consiste en:

1. Componentes con capacidad de proceso y realizado de transacciones en la memoria local (por ejemplo, una CPU).

2. Una red que encamina mensajes entre pares de los mencionados componentes de procesado.

3. Un entorno *hardware* que permite la sincronización de todos o un subconjunto de los componentes de procesado.

[7]http://giraph.apache.org/

Una máquina BSP suele estar implementada como un conjunto de procesadores que ejecutan diferentes hilos de computación, equipados con una memoria local rápida e interconectados por una red de comunicación. La comunicación es un componente esencial en un algoritmo BSP, donde cada proceso que se lleva a cabo en una serie de *superpasos* que se repiten, y están compuestos por tres subfases:

- **Computación concurrente**: cada procesador realiza cálculos de manera local y asíncrona accediendo solo a los valores guardados en la memoria local.
- **Comunicación**: los procesos intercambian datos entre ellos para facilitar la persistencia remota de datos.
- **Sincronización de barrera**: cuando un proceso alcanza un punto de barrera, espera hasta que los demás procesos hayan alcanzado dicha barrera.

La computación y la comunicación no tienen por qué estar ordenadas de forma temporal. La comunicación típicamente es unidireccional, en la que el proceso pide y envía datos a una memoria remota (nunca recibe datos si no los pide). La posterior sincronización de barrera asegura que todas las comunicaciones unidireccionales de los procesos han finalizado satisfactoriamente.

Giraph define los sistemas de procesado como un grafo compuesto por vértices unidos por aristas dirigidas. Por ejemplo, los vértices podrían representar una persona y las aristas las peticiones de amistad en una red social. La computación se lleva a cabo mediante una serie de *superpasos* anteriormente descritos.

A la hora de implementar vértices, el usuario puede decidir las características, memoria, y velocidad de cada uno de ellos. Esto se puede conseguir implementando varias de las superclases abstractas de Vertex que Giraph provee. Mientras los usuarios expertos pueden implementar sus propios vértices a bajo nivel, el equipo de Giraph recomienda a los usuarios noveles/intermedios implementar subclases de Vertex y adaptar los ejemplos que ellos mismos proveen.[8]

Tipos de grafos soportados por Giraph

El tipo de grafo por defecto en Giraph son grafos dirigidos, en los que todas los arcos se consideran enlaces de salida que salen del vértice en el cual se han definido. A partir de ahí, existe también la posibilidad de definir otros tipos de grafos:

- Grafos no dirigidos.
- Grafos no etiquetados.
- Grafos etiquetados.
- Multigrafos.

Agregadores

Los agregadores permiten realizar computaciones de manera global, sobre todos los nodos. Se pueden usar para comprobar que una condición global se cumple, o para calcular algunas estadísticas (por ejemplo, contar elementos). Durante un *superpaso*, los vértices asignan valores a los agregadores, que luego son agregados (valga la redundancia) por el sistema, haciendo que los resultados estén disponibles para todo el

[8]http://cort.as/-Enq9

sistema en el siguiente *superpaso*. Por ello, la operación realizada por un agregador debe cumplir la propiedad conmutativa y asociativa (por ejemplo, una suma, un AND o OR booleano, etc.).

El usuario puede crear sus propios agregadores, pero Giraph provee algunos agregadores básicos: encontrar el máximo o mínimo entre dos números, sumatorios, etc.

Giraph es un motor de computación de grafos, y no una biblioteca de algoritmos, por lo que los algoritmos para tratar grafos son presentados como «ejemplos» en el paquete *org.apache.giraph.examples*, entre los cuales se encuentran:

- **BrachaTouegDeadlockComputation**,[9] para detectar situaciones de *deadlock*, en las que un vértice A necesita para continuar un recurso proporcionado por un vértice B, pero B no puede proporcionarlo porque necesita un recurso que solo A le puede proporcionar.
- **ConnectedComponentsComputation**,[10] que detecta componentes que conectan varios vértices y asigna a cada vértice un identificador del componente al que pertenecen.
- **MaxComputation**, que detecta el valor máximo de un grafo.
- **PageRankComputation**, que implementa el algoritmo PageRank de Google para puntuar la importancia de los vértices de un grafo en base a, entre otros criterios, cuántos vértices le enlazan y qué características tienen.

[9] https://bit.ly/2MqObA1
[10] https://bit.ly/2CsedOZ

- **RandomWalkComputation**, que ejecuta un «paseo aleatorio» por los vértices de un grafo.
- **SimpleShortestPathsComputation**, para hallar el camino más corto entre dos vértices de un grafo en base al número de aristas y el peso asignado a estas.

Además, es fácil encontrar librerías de terceros que agrupan algoritmos para Giraph, por ejemplo las del grupo de investigación en bases de datos de la Universidad de Leipzig,[11] tales como:

- **BTG**: extrae grafos de transacciones de negocio (*business transactions graphs*, BTG) (Petermann *et al.*, 2014).
- **Propagación de etiquetas**: encuentra comunidades dentro de grafos mediante la propagación a través de los vértices de etiquetas identificativas de la comunidad.
- **Repartición adaptativa**: particiona un grafo utilizando propagación de etiquetas (Vaquero *et al.*, 2014).

15.3.2. Spark GraphX

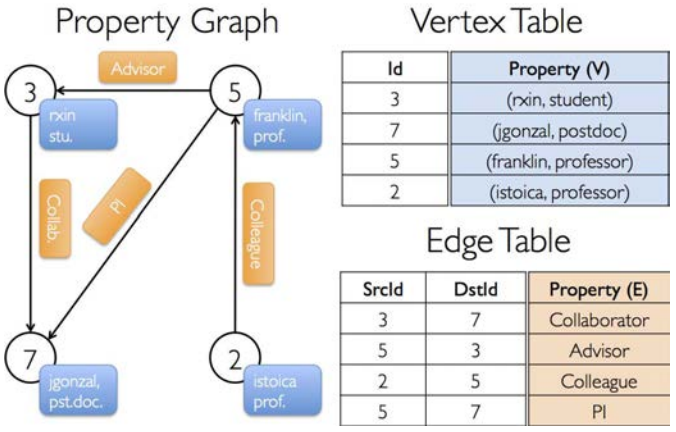
GraphX[12] es la API de Spark para el procesado y computación paralela de grafos. Tal y como hemos visto con las otras APIs de Spark, en GraphX la capa de abstracción RDD es la llamada *resilient distributed property graph* y expone una serie de operadores y algoritmos específicos para simplificar el trabajo con grafos.

[11] https://github.com/dbs-leipzig/giraph-algorithms
[12] http://spark.apache.org/graphx/

GraphX es la API de Spark para extender sus funcionalidades para trabajar con grafos, que a su vez son representables como tablas, unificando el procesado de datos y el de grafos en paralelo. En ocasiones el usuario va a preferir representar los datos como tablas en vez de vértices y ejes, de modo que GraphX permite trabajar con RDD que pueden visualizarse a la vez como componentes de un grafo (vértices —VertexRDD; aristas— EdgeRDD) y sus características como colecciones (RDD propiamente dichos).

Por tanto, un grafo en GraphX es solo una vista de los mismos datos, y cada vista tiene sus operadores específicos. Como muestra la figura 9, un mismo grafo puede almacenarse utilizando el formato de vértices y aristas, o bien como una colección de datos tabulares.

Figura 9. Representaciones de un grafo en GraphX y sus posibles vistas



Fuente: http://spark.apache.org/graphx/

Apache GraphX, además de las API para escribir programas para analíticas de grafos, distribuye implementaciones de algunos de los algoritmos esenciales para Grafos. En su versión 1.6.1, GraphX soporta los siguientes algoritmos:

- **PageRank**: mide la importancia de cada vértice en un grafo, asumiendo que una arista de un vértice A a un vértice B supone una validación de la importancia de B para A. Por ejemplo, si una página web es enlazada por muchas otras páginas, esta será catalogada como página importante (motivo para que aparezca en las primeras posiciones de un buscador).

  GraphX implementa versiones estáticas y dinámicas de PageRank. Las versiones estáticas ejecutan un número fijo de iteraciones, mientras que las dinámicas se ejecutan hasta que los *rankings* convergen hasta llegar a una solución aceptable.

- **Componentes conectados**: conocido como *connected components* en inglés, es un algoritmo que etiqueta a cada componente (grupo de vértices) conectado del grafo.

- **Conteo de triángulos**: un vértice forma parte de un triángulo cuando tiene dos vértices adyacentes, también unidos por una arista entre sí. El algoritmo TriangleCount determina el número de triángulos de los que forma parte un vértice, y se usa para detectar agrupaciones o *clusters*.

- **Propagación de etiquetas**: encuentra comunidades dentro de grafos mediante la propagación a través de los vértices de etiquetas identificativas de la comunidad.

Bibliografía

Alon, N.; Matias, Y.; Szegedy, M. (1996). «The Space Complexity of Approximating the Frequency Moments». En: *ACM Symposium on Theory of Computing*.

Barabási, A.-L.; Pósfai, M. (2016). *Network Science*. Cambridge: Cambridge University Press.

Barlas, G. (2015). *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann Publishers Inc.

Bengfort, B.; Kim, J. (2016). *Data Analytics with Hadoop*. O'Reilly Media.

Biery, R. (2017). *Introduction to GPUs for Data Analytics. Advances and Applications for Accelerated Computing*. Packt Publishing.

Bloom, B. (1970). «Space/Time Trade-Offs in Hash Coding with Allowable Errors». *Communications of the ACM* (vol. 13, n.º 7, págs. 422-426).

Broder, A. Z.; Mitzenmacher, M. (2003). «Network Applications of Bloom Filters: A Survey». *Internet Mathematics* (vol. 1, n.º 4, págs. 485-509).

Cormode, G.; Muthukrishnan, S. (2005). «An Improved Data Stream Summary: The Count-Min Sketch and its Applications». *Journal of Algorithms* (vol. 55, n.º 1, págs. 58-75).

Dunning, T.; Friedman, E. (2015). *Real-World Hadoop.* O'Reilly Media.

Frick, A.; Ludwig, A.; Mehldau, H. (1995). *A Fast Adaptive Layout Algorithm for Undirected Graphs.* Graph Drawing (págs. 388-403). Berlin/Heidelberg: Springer.

Garofalakis, M.; Gehrke, J.; Rastogi, R. (2016). *Data Stream Management: Processing High-Speed Data Streams.* Springer.

Kleppmann, M. (2016). *Making Sense of Stream Processing: The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms.* O'Reilly Media.

Kumar, M.; Singh, C. (2017). *Building Data Streaming Applications with Apache Kafka.* Packt Publishing.

Laney, D. (2001). *3D Data Management: Controlling Data Volume, Velocity, and Variety.* [documento en línea]. http://cort.as/-FUOc

Malewicz, G.; Austern, M. H.; Bik, A. J.; Dehnert, J. C.; Horn, I.; Leiser, N.; Czajkowski, G. (2010). «Pregel: A System for Large-Scale Graph Processing». En: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10* (págs. 135-146), Nueva York: ACM.

Newman, M. (2010). *Networks: An Introduction*. Nueva York: Oxford University Press, Inc.

Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E. (2014). «Biiig: Enabling Business Intelligence with Integrated Instance Graphs». En: *2014 IEEE 30th International Conference on Data Engineering Workshops* (págs. 4-11).

Pérez-Solá, C.; Casas-Roma, J. (2016). *Análisis de datos de redes sociales*. Barcelona: Editorial UOC.

Vaquero, L. M.; Cuadrado, F.; Logothetis, D.; Martella, C. (2014). «Adaptive Partitioning for Large-Scale Dynamic Graphs». En: *2014 IEEE 34th International Conference on Distributed Computing Systems* (págs. 144-153).

Viktor Mayer-Schönberger, K. C. (2013). *Big Data: A Revolution That Will Transform How We Live, Work and Think*. John Murray.

White, T. (2015). *Hadoop: The Definitive Guide. Storage and Analysis at Internet Scale (4th Edition)*. O'Reilly Media.

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

Zaharia, M.; Karau, H.; Konwinski, A.; Wendell, P. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media.

JORDI CASAS ROMA
JORDI NIN GUERRERO
FRANCESC JULBE LÓPEZ

TECNOLOGÍA

BIG DATA

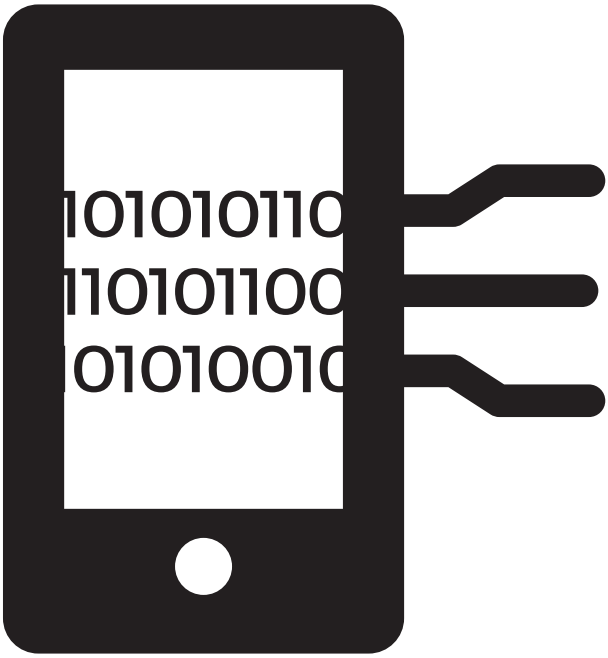
ANÁLISIS DE DATOS EN ENTORNOS MASIVOS

En este libro se introducen los conceptos fundamentales del análisis de datos en entornos *big data*. Se ofrece una revisión completa de las técnicas avanzadas más usadas en estos campos, con un enfoque claramente descriptivo, para que el lector entienda los conceptos e ideas básicos de cada modelo o técnica.

Los diferentes capítulos de esta obra comprenden la definición de problemas que requieran el uso de técnicas de *big data*, la tecnología básica empleada en este tipo de proyectos, así como una descripción de los principales escenarios de procesamiento de datos masivos: procesamiento de datos por lotes (*bacth*), procesamiento de datos continuos o en flujo (*streaming*) y procesamiento de datos en formato de grafos. Para cada uno de estos escenarios, se describirán las especificidades y herramientas principales relacionadas con las etapas de captura y preprocesamiento, almacenamiento y análisis.

Con este libro aprenderás sobre:

✓ *big data*; ✓ datos masivos; ✓ computación distribuida; ✓ inteligencia artificial; ✓ minería de datos; ✓ aprendizaje automático; ✓ ciencia de datos; ✓ *data science*



EDITORIAL UOC