Captura, preprocesamiento y almacenamiento de datos masivos

PID_00250688

Francesc Julbe
Jordi Conesa Caralt
Jordi Casas Roma
M.a Elena Rodríguez González

Tiempo mínimo de dedicación recomendado: 5 horas

Universitat Oberta de Catalunya

Índice

Introducción

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

La primera parte de este módulo agrupa los procesos de captura y preprocesamiento de datos, mientras que en la segunda parte nos adentraremos en el almacenaje de datos masivos.

Iniciaremos la primera parte de este módulo con un análisis del proceso de captura de datos, con especial énfasis en la captura de datos dinámicos o en *streaming*. Veremos qué son los datos en *streaming* y los componentes que intervienen en su creación, distribución y consumo. Posteriormente veremos más en detalle cómo se puede realizar la distribución de datos para facilitar la escalabilidad y garantizar la alta disponibilidad.

Continuaremos viendo las tareas asociadas al preprocesamiento de datos y las arquitecturas básicas que dan apoyo a estas tareas. Igual que en el punto anterior, prestaremos especial atención a los datos en *streaming*, que por su casuística requieren de soluciones más complejas y apartadas de los procesos analíticos «tradicionales».

La segunda parte del módulo comprende dos grandes bloques, que son por un lado el almacenamiento mediante sistemas de ficheros distribuidos y, por el otro, el almacenamiento utilizando bases de datos NoSQL.

En relación con los sistemas de ficheros distribuidos, veremos su arquitectura y sus funcionalidades básicas. Aunque existen múltiples sistemas de ficheros distribuidos, en este material nos centraremos en el sistema de ficheros distribuidos de Hadoop (*Hadoop distributed file system*, HDFS).

Finalizaremos el segundo bloque del módulo introduciendo las bases de datos NoSQL. Concretamente, veremos los distintos modelos existentes y presentaremos las principales ventajas e inconvenientes de cada uno de estos modelos.

Objetivos

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

1. Conocer los procesos de captura de datos masivos, así como algunas de las herramientas más representativas en el entorno de datos masivos.
2. Comprender las tareas asociadas al preprocesamiento de datos, así como las principales arquitecturas asociadas.
3. Descubrir los principales sistemas de almacenamiento en entornos de datos masivos.
4. Conocer las principales características de un sistema de ficheros distribuido, así como los mecanismos que permiten la fiabilidad y escalabilidad de los datos.
5. Saber el funcionamiento básico del sistema de ficheros distribuido HDFS, incluyendo los comandos básicos.
6. Aprender cuáles son los principales modelos de bases de datos NoSQL existentes y sus principales ventajas e inconvenientes.
7. Ser capaz de escoger un buen sistema de almacenamiento masivo a partir de las definiciones y los requerimientos de los datos.

1. Captura de datos masivos

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

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

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

- **Datos estáticos**: son datos que ya se encuentran almacenados en algún lugar, ya sea en formato de ficheros (por ejemplo, ficheros de texto, JSON, XML, etc.) o en bases de datos.
- **Datos dinámicos o en *streaming***: son datos que se producen de forma continua, y que deben ser capturados durante un umbral limitado de tiempo, ya que no estarán disponibles pasado un cierto período de tiempo.

Datos estructurados

Los datos estructurados son aquellos que siguen un patrón igual para todos los elementos que, además, es conocido *a priori*. Por ejemplo, los datos de una hoja de cálculo presentan los mismos atributos para cada fila.

Datos semiestructurados

Los datos semiestructurados son una forma de datos que no contiene una estructura fija predefinida *a priori*, pero contiene etiquetas u otros marcadores para separar los elementos semánticos y hacer cumplir jerarquías de registros y campos de los datos. Por ejemplo, las documentos JSON o HTML.

Datos no estructurados

Los datos no estructurados son aquellos que no siguen ningún tipo de patrón conocido *a priori*. Por ejemplo, dos documentos de texto o imágenes.

1.1. Datos estáticos

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

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

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

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

- datos contenidos en bases de datos, bien de tipo relacional (como por ejemplo, MySQL o PostgreSQL) o bien de tipo NoSQL (Cassandra o Neo4J, entre muchos otros).

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

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

Un ejemplo de herramienta completa de ETL puede ser el paquete ofimático (*suite*) Data Integration* de Pentaho, que permite hacer las conexiones a distintas fuentes de datos y crear el *pipeline* de transformación de los datos, en caso de ser necesario.

* http://bit.ly/Zg49Di

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

1.1.1. Apache Sqoop

Apache Sqoop** (SQL-to-Hadoop) es una herramienta de importación y exportación de bases de datos relacionales. Sqoop está diseñado para la transferencia eficiente de datos entre una base de datos relacional, como por ejemplo MySQL u Oracle, hacia un almacén de datos Hadoop, incluidos HDFS, Hive y HBase, y viceversa. Automatiza la mayor parte del proceso de transferencia de datos, leyendo la información del esquema directamente desde el sistema gestor de la base de datos. Sqoop luego usa MapReduce para importar y exportar los datos hacia y desde Hadoop.

** http://sqoop.apache.org/

Sqoop proporciona flexibilidad para mantener los datos en su estado de producción mientras son copiados en Hadoop con tal de que estén disponibles para un análisis posterior, sin modificar la base de datos de producción.

El funcionamiento de Sqoop es simple, solo debemos indicarle el origen de los datos y el destino dentro del sistema de almacenamiento de Hadoop para proceder a la importación de datos en el entorno analítico.

Ejemplo: importación de datos de MySQL a HDFS

En los apartados posteriores de este módulo veremos los detalles de implementación y funcionamiento del sistema de ficheros distribuido HDFS, pero podemos obviar los detalles para presentar un breve ejemplo de importación de datos mediante la herramienta Sqoop.

Supongamos que disponemos de una base de datos MySQL, trabajando en el puerto 3306 y con una base de datos llamada *ejemplo1* que contiene la tabla «sampledata» con la información que nos interesa transferir a Hadoop para su posterior análisis.

El siguiente fragmento de código muestra la orden que ejecuta Sqoop, indicándole la cadena de conexión a la base de datos MySQL, que es el origen de datos. El parámetro opcional `-m 1` indica que este trabajo debe usar una única tarea de Map.

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost:3306/ejemplo1
--username root --table sampledata -m 1

17/09/15 22:47:35 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
17/09/15 22:47:35 INFO manager.MySQLManager: Preparing to use a MySQL
streaming resultset.
17/09/15 22:47:35 INFO tool.CodeGenTool: Beginning code generation
17/09/15 22:47:36 INFO manager.SqlManager: Executing SQL statement:
SELECT t.* FROM `average_price_by_state` AS t LIMIT 1

(output truncated)

17/09/15 22:47:53 INFO mapreduce.ImportJobBase: Transferred 200.4287 KB
in 15.3718 seconds (13.0387 KB/sec)
17/09/15 22:47:53 INFO mapreduce.ImportJobBase: Retrieved 3272 records.
```

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

```
/srv/sqoop$ hadoop fs -head average_price_by_state/part-m-00000

2012,AK,Total Electric Industry,17.88,14.93,16.82,0.00,null,16.33
2012,AL,Total Electric Industry,11.40,10.63,6.22,0.00,null,9.18
2012,AR,Total Electric Industry,9.30,7.71,5.77,11.23,null,7.62
2012,AZ,Total Electric Industry,11.29,9.53,6.53,0.00,null,9.81
2012,CA,Total Electric Industry,15.34,13.41,10.49,7.17,null,13.53
2012,CO,Total Electric Industry,11.46,9.39,6.95,9.69,null,9.39
2012,CT,Total Electric Industry,17.34,14.65,12.67,9.69,null,15.54
2012,DC,Total Electric Industry,12.28,12.02,5.46,9.01,null,11.85
2012,DE,Total Electric Industry,13.58,10.13,8.36,0.00,null,11.06
2012,FL,Total Electric Industry,11.42,9.66,8.04,8.45,null,10.44
```

La instrucción anterior muestra las primeras filas del fichero *part-m-00000*, donde podemos ver las diez primeras filas de los datos importados de la base de datos MySQL.

El proceso de importación a HBase o Hive es similar al ejemplo que acabamos de ver.

1.2. Datos en *streaming*

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

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

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

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

1.2.1. *Streams* de datos

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

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

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

```
(<25 grados, 70 % humedad>, 1483309623)
(<25 grados, 70 % humedad>, 1483309624)
(<26 grados, 60 % humedad>, 1483309625)
(<15 grados, 90 % humedad>, 1483309626)
```

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

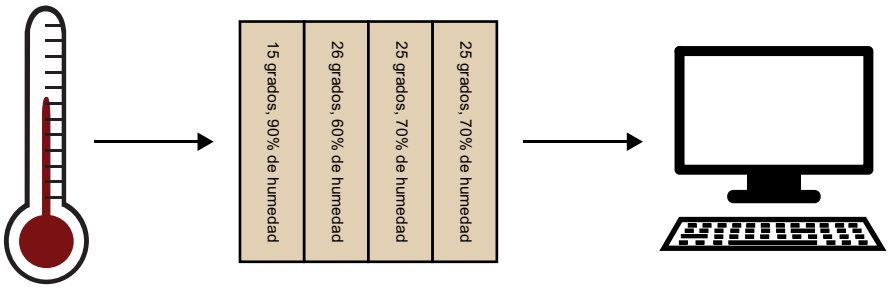
En función de si la *t* viene determinada por un momento en el tiempo (un sensor mide la temperatura cada milisegundo, por ejemplo) o por el orden causal de determinados eventos (un internauta ha eliminado un producto de su cesta de la compra) estaremos hablando de *streams* originados por el **tiempo**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

Tiempo Unix

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

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

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



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

Utilizar sistemas de envío y procesamiento en *streaming* es beneficioso en escenarios donde se generan nuevos datos de forma continua. Son sistemas de interés general, que se encuentran presentes en todo tipo de negocio. Desde un punto de vista analítico, las posibilidades del procesamiento de *streams* son muchas y muy variadas. Las más simples son la recolección y el procesamiento de *logs* (registros), la generación de informes o cuadros de mando y la creación de alarmas en tiempo real. Las más complejas incluyen análisis de datos más complejos, como el uso de algoritmos de aprendizaje automático o de procesamiento de eventos.

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

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

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

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

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

1.2.2. Envío y recepción de datos en *stream*

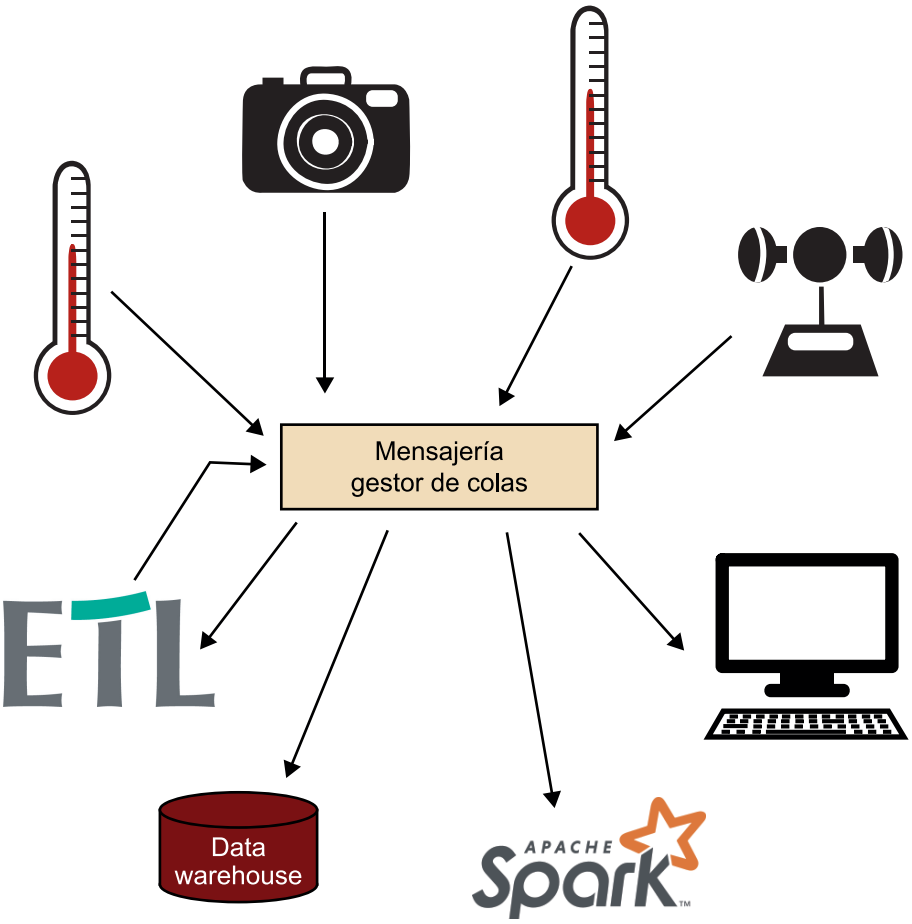
Este subapartado trata sobre los sistemas de mensajería. Estos sistemas permiten el intercambio de datos entre distintos ordenadores.

Cuando queremos compartir datos entre distintos dispositivos podemos tomar distintas alternativas. Quizá la alternativa más sencilla sería utilizar una arquitectura cliente-servidor. En un entorno cliente-servidor los datos se generan en el cliente (el productor) y se envían directamente al servidor (el consumidor). Recordemos que en el caso general este sistema podrá tener múltiples productores y consumidores, por lo tanto tendríamos que implementar un sistema cliente-servidor con tantos clientes como productores haya y tantos servidores como consumidores haya.

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

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

Figura 2. Ejemplo de utilización de un sistema de mensajería



El sistema de la figura 2 se compone de cuatro productores y cuatro consumidores. Los productores son de distinto tipo: dos sensores de temperatura, una cámara fotográfica y un anemómetro (sensor de viento). La información recolectada por estos sistemas se envía al sistema de gestión de colas. Este es el encargado de garantizar que la información enviada (los mensajes) lleguen a sus potenciales consumidores. Estos consumidores podrían ser, por ejemplo, un almacén de datos (*data warehouse*) que almacene la información recolectada hasta el momento, un cuadro de mando, un sistema Spark para hacer analíticas en tiempo real y un proceso de extracción, transformación y carga que agregue y refine los datos. Este sistema ETL podría generar nuevos datos, como por ejemplo estimar la temperatura de determinados puntos geográficos en función de las temperaturas recibidas, y añadir esta nueva información en un nuevo *stream* de datos. En este caso, el proceso ETL se comportaría también como productor para otro *stream* de datos.

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

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

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

- **Permite el envío de datos de forma asíncrona**: los datos recibidos se guardan en una cola y se van procesando a medida que el consumidor los solicita.

- **Permite desacoplar los distintos componentes del sistema distribuido**: se podrán añadir o eliminar creadores o consumidores de información sin necesidad de modificar o ajustar los nodos del sistema distribuido.

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

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

Existen distintos protocolos de mensajería que pueden utilizarse para realizar el envio de mensajes entre productores y consumidores. Los más populares en el contexto de los datos masivos y del internet de las cosas son quizá AMQP* (*advance message queuing protocol*), MQTT** (*message queue telemetry transport*) y STOMP*** (*simple streaming text oriented messaging protocol*).

*** https://www.amqp.org/**
**** http://mqtt.org**
***** https://stomp.github.io**

Estos protocolos son adoptados por distintos sistemas de mensajería que implementan una capa de software intermediario para el envío y la recepción

de mensajes. Algunos de los más populares son: Apache Kafka, Apache ActiveMQ* y RabbitMQ**.

* https://www.amqp.org
** https://www.rabbitmq.com

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

1.2.3. Introducción a Apache Kafka

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

*** https://kafka.apache.org

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

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

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

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

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

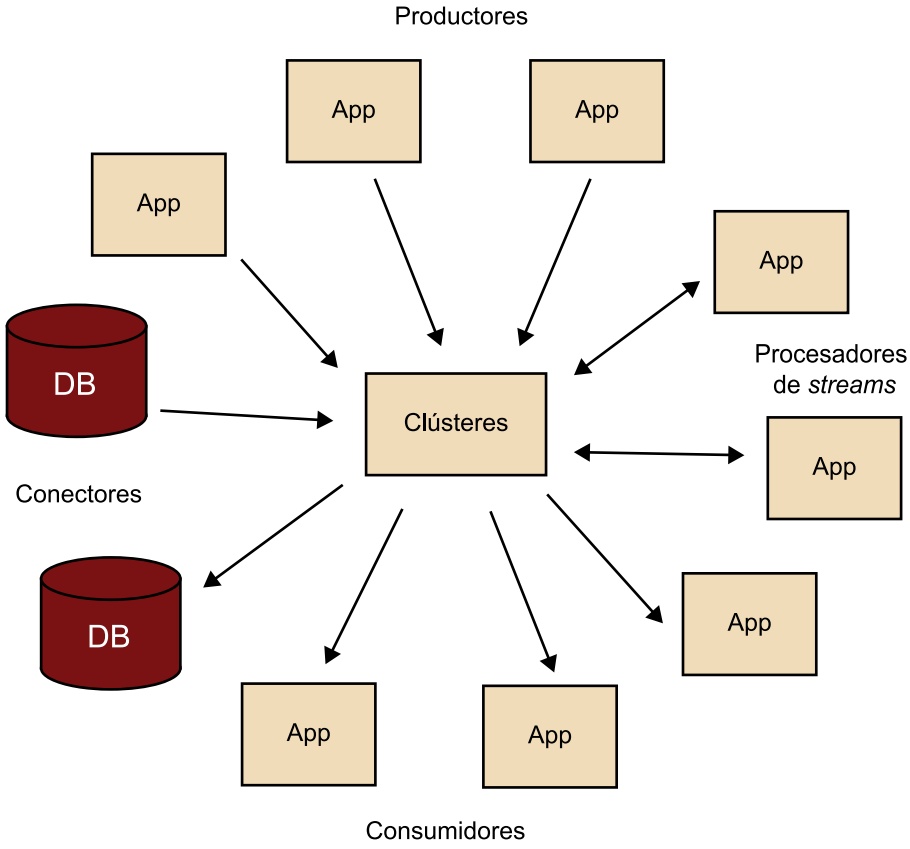
Otra característica interesante de Kafka es que, a diferencia de otros sistemas de mensajería, el consumo de un dato por parte de un productor no elimina el dato del *stream*. Los datos enviados a Kafka tienen fecha de caducidad (se puede configurar el tiempo que deben estar disponibles) y se mantienen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

Arquitectura de Apache Kafka

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

- **Productores**: son los sistemas que envían datos a Kafka. Estos datos se envían en forma de registro y podrán ser publicados en uno o más *topics*.
- **Clústeres**: el sistema encargado de la recepción, fragmentación, distribución, replicación y envío de mensajes. Puede estar compuesto de distintos nodos, cada uno de ellos denominado *broker*.
- **Consumidores**: son los sistemas de destino del *stream* de datos. Se pueden suscribir a uno o más *topics* para obtener los mensajes de los mismos.
- **Conectores**: permiten crear consumidores (o productores) reutilizables que consuman (o publiquen) datos en uno o más *topics* a partir de bases de datos o aplicaciones existentes.
- **Procesadores de *streams***: permiten crear aplicaciones para preprocesar y enriquecer los datos de un *stream* de datos. Estos sistemas actúan como consumidores de uno o más *topics*, realizan operaciones sobre los datos proporcionados y escriben los datos resultantes en uno o más *topics*.

Figura 3. Principales componentes de Kafka

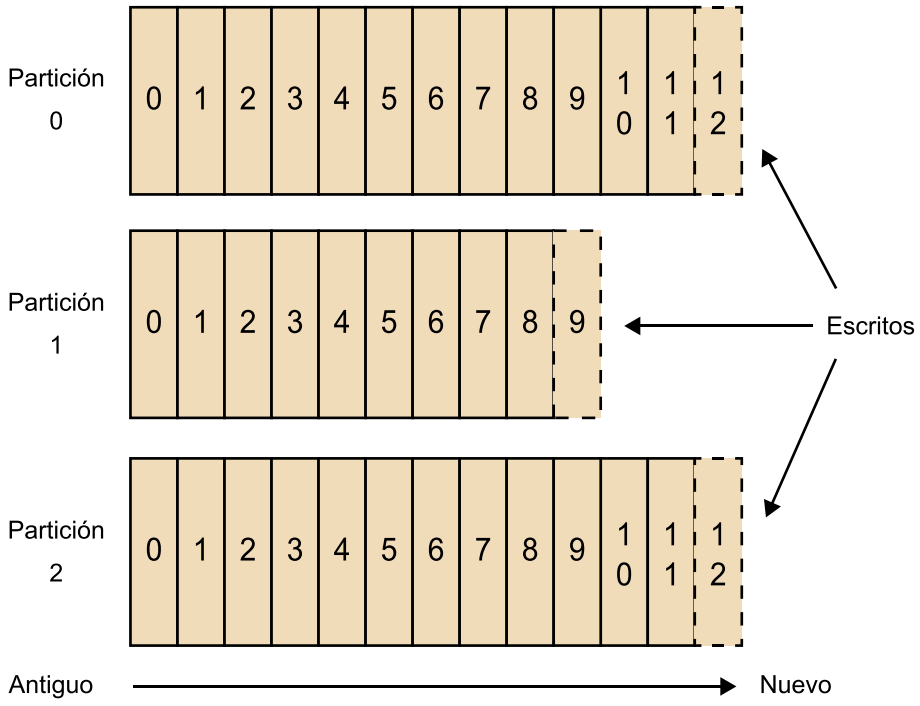


Representación interna de los *streams* en Kafka

Tal y como se ha comentado, los datos se distribuyen en *topics* según su categoría. Por tanto, cuando un productor envia datos a Kafka, deberá indicar a qué *topic* corresponden. Por otro lado, los *topics* pueden estar ligados a cero, uno o más consumidores.

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

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



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

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

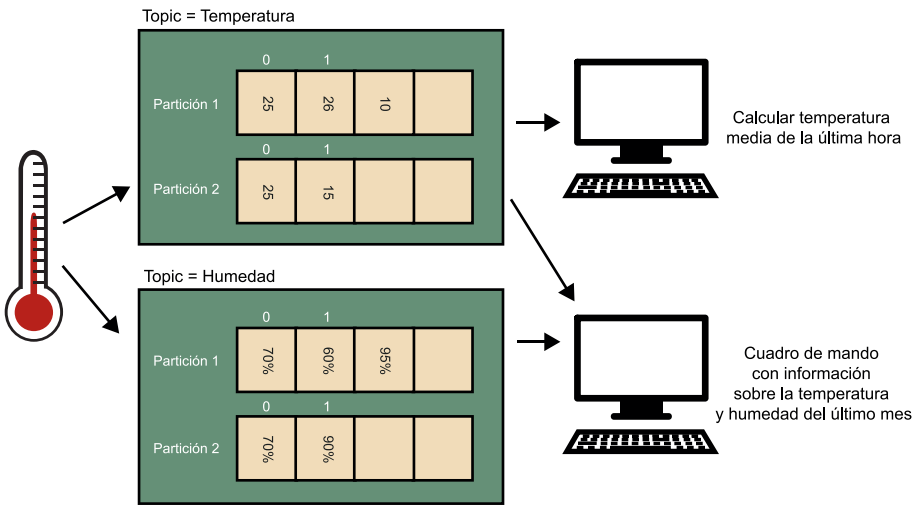
Cuando un consumidor quiere consultar los datos de un *stream* deberá saber el *topic* y la partición en los que se han asignado los datos de interés. Para indicar qué datos se quieren consumir, el consumidor deberá indicar también el *offset* de los registros que desea obtener. Tal y como se ha comentado anteriormente, el consumo de un registro no eliminará el registro del *topic*.

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

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

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

Figura 5. Ejemplo de uso de Apache Kafka



Respecto a los consumidores, el primer consumidor solo necesita datos de temperatura para calcular la temperatura media. Por tanto, solo consultará datos del *topic temperatura*. El segundo consumidor deberá proporcionar información sobre la temperatura y la humedad y, por tanto, deberá consultar

ambos *topics*. Recordad que los consumidores deberán saber el *topic*, la partición y el *offset* de los datos que se van a consultar en cada momento.

Distribución y replicación

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

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

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

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

2. Preprocesamiento de datos masivos

El propósito fundamental de la fase de preprocesamiento o preparación de datos es manipular y transformar los datos brutos (*raw data*) para que el contenido de información sea coherente, homogéneo y consistente. Este proceso es muy importante cuando se recolecta información de múltiples fuentes de datos.

2.1. Tareas de preprocesamiento de datos

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

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

- tareas de limpieza de datos (*data cleansing*)
- tareas de transformación de datos (*data wrangling*)

2.1.1. Limpieza de datos

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

Después de la limpieza, un conjunto de datos debe ser coherente con otros conjuntos de datos similares en el sistema. Las inconsistencias detectadas o eliminadas pueden haber sido causadas originariamente por errores en los datos originales (ya sean errores humanos o errores producidos en sensores u

otros automatismos), por corrupción en la transmisión o el almacenamiento, o por diferentes definiciones de diccionario de datos de las distintas fuentes de datos empleadas.

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

Ejemplo: integración de fechas

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

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

- A partir de una cadena de texto que representa una fecha, en un formato concreto, como puede ser "YYYY-MM-DD hh:mm:ss". En este formato se indica la fecha a partir del año, mes, día, hora, minutos y segundos. Además, hay que añadir el tema de la zona horaria.

- A partir del número de segundos desde la fecha de 1 de enero del año 1970, que es conocido como *Unix timestamp*. Este sistema es muy empleado en los sistemas operativos Unix y derivados (por ejemplo, Linux) e identifica un momento del tiempo concreto a partir de un valor numérico. Por ejemplo, el valor 1502442623 equivale a la fecha 11 de agosto de 2017 a las 9 horas, 10 minutos y 23 segundos.

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

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

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

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

2.1.2. Transformación de datos

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

Esto puede incluir la agregación de datos, así como muchos otros usos potenciales. A partir de los datos extraídos en crudo (*raw data*) del origen de datos, se pueden formatear los datos para posteriormente almacenarlos en estructuras de datos predefinidas o, por ejemplo, utilizar algoritmos simples de minería de datos o técnicas estadísticas para generar agregados o resúmenes que posteriormente serán almacenados en el repositorio analítico.

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

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

2.2. Arquitecturas de preprocesamiento

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

- La tipología del proceso de captura de los datos. En este sentido es importante determinar si el sistema trabajará con datos estáticos o datos dinámicos (en *streaming*).

- La complejidad de las operaciones de limpieza y transformación que se deberán aplicar a los datos antes de almacenarlos en el repositorio analítico.

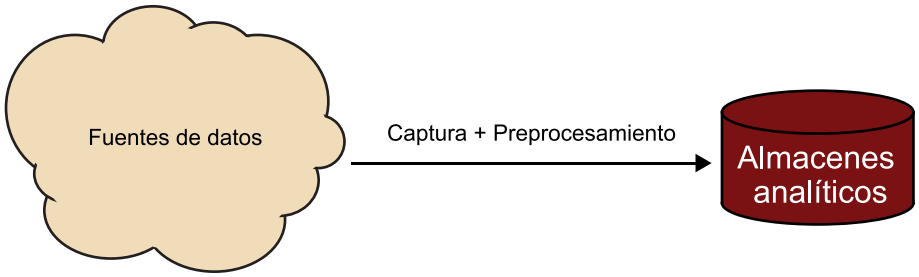
Según estos y otros factores, encontramos dos modelos principales de estructura o arquitectura de preprocesamiento:

- preprocesamiento durante el proceso de captura de los datos
- preprocesamiento independiente del proceso de captura de los datos

En el primer modelo, los datos son limpiados y transformados durante el proceso de captura de datos, y se almacenan una vez han sido limpiados y trans-

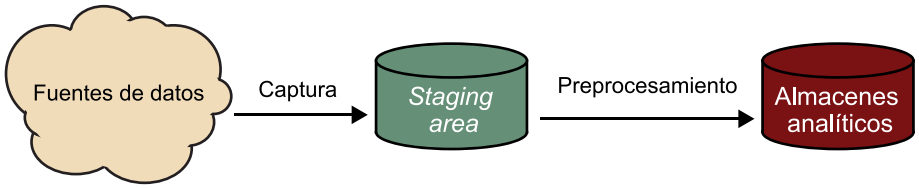
formados. Podemos decir que los procesos de captura y preprocesamiento se realizan en serie, pero sin requerir de un sistema de almacenamiento intermedio.

Figura 6. Esquema de una arquitectura sin *staging area*



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

Figura 7. Esquema de una arquitectura que incorpora *staging area*



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

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

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

3. Almacenamiento de datos masivos

Uno de los aspectos en los que los datos masivos han tenido el impacto más directo es el del almacenamiento de datos. Muchos escenarios que requieren del manejo de grandes volúmenes de datos obligan a las empresas e instituciones a adoptar soluciones de almacenaje capaces de almacenar una gran cantidad de datos a la vez que ofrecen altas prestaciones, escalabilidad y fiabilidad frente a cualquier problema y/o fallo.

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

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

- sistemas de ficheros distribuidos
- bases de datos, principalmente de tipo NoSQL

Los sistemas de ficheros distribuidos se basan en el funcionamiento de los sistemas de ficheros tradicionales, gestionados por el sistema operativo, pero en este caso se almacenan los datos de forma transversal en un conjunto heterogéneo de computadoras.

Ved también

En el apartado 4 veremos los sistemas de ficheros distribuidos, haciendo hincapié en una de los sistemas más conocidos y empleados en la actualidad.

Por otro lado, las bases de datos NoSQL son una opción de almacenamiento masivo y distribuido muy empleado por cualquier empresa o institución que trabaje con datos masivos.

Ved también

Veremos una introducción a las bases de datos NoSQL, y estudiaremos los distintos tipos y sus principales características en el apartado 5, aunque una descripción detallada de todas ellas queda fuera del alcance de estos materiales.

4. Sistemas de ficheros distribuidos

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

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

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

Metadato

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

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

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

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

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

4.1. ¿Qué es HDFS?

El sistema de archivos distribuidos Hadoop (en adelante HDFS, *Hadoop distributed filesystem*) es un subproyecto de Apache Hadoop* y forma parte del ecosistema de herramientas de datos masivos que proporciona Hadoop.

* https://hadoop.apache.org

HDFS es un sistema de archivos altamente tolerante a fallos diseñado para funcionar con el llamado «hardware de bajo coste» (*commodity* hardware), lo que permite reducir los costes de almacenamiento significativamente. Proporciona acceso de alto rendimiento a grandes volúmenes de datos y es adecuado para aplicaciones de procesamiento distribuido, tal y como su nombre indica.

***Commodity* hardware**

La computación de «bajo coste» implica el uso de un gran número de componentes de computación ya disponibles para la computación paralela, con el objetivo de obtener la mayor cantidad de computación útil a bajo coste.

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

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

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

Y entre las principales características de HDFS podemos citar:

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

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

- HDFS ha sido diseñado para ser fácilmente portable de una plataforma a otra, lo que lo hace compatible en entornos de hardware heterogéneo (*commodity* hardware) y diferentes sistemas operativos. HDFS está implementado con lenguaje Java, de modo que cualquier máquina que admita la programación Java puede ejecutar HDFS. Esto facilita la adopción generalizada de HDFS como una plataforma de elección para un gran conjunto de aplicaciones. Además, este factor incrementa su escalabilidad, ya que la integración de un nuevo nodo en un clúster HDFS impone pocas restricciones.

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

- Proporciona diversas interfaces para que las aplicaciones trabajen sobre los datos almacenados, así que maximiza la facilidad de uso y el acceso.

4.2. Arquitectura

HDFS no es un sistema de ficheros regular como podría ser ext3* o NTFS**, que están construidos sobre el *kernel* del sistema operativo, sino que es una abstracción sobre el sistema de ficheros regular. HDFS es un servicio que se ejecuta en un entorno distribuido de ordenadores o clústeres siguiendo una arquitectura de maestro (al que llamaremos *NameNode* en adelante) y esclavo (*DataNode* en adelante). En un clúster suele haber un solo *NameNode* y varios *DataNodes*, uno por ordenador en el clúster.

*** http://bit.ly/2Cvkeuz**
**** http://bit.ly/2EkhQY1**

4.2.1. *NameNode* y *DataNode*

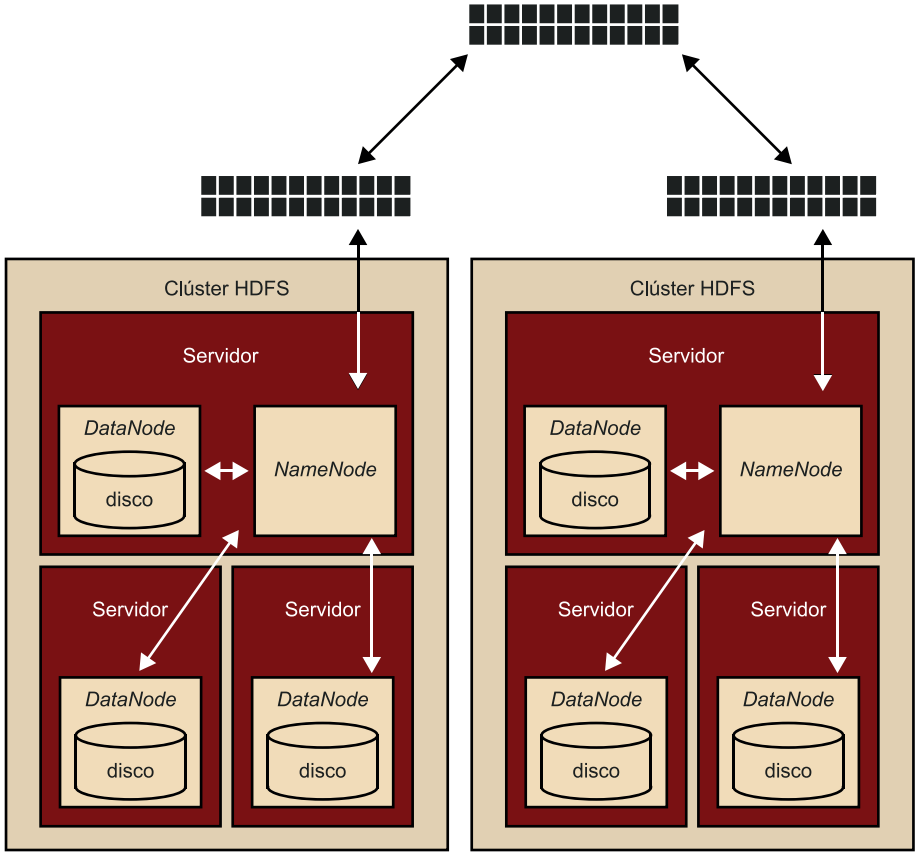
El *NameNode* gestiona las operaciones del *Namespace* del sistema de archivos como abrir, cerrar y renombrar archivos y directorios, y regula el acceso del

cliente a los archivos. También es responsable de la asignación de los datos a su correspondiente *DataNode*.

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

Los *NameNodes* y los *DataNodes* son componentes de software diseñados para ejecutarse de una manera desacoplada entre ellos en los nodos del clúster. Tal y como muestra la figura 8, la arquitectura típica consta de un nodo en el que se ejecuta el servicio *NameNode* y posiblemente el servicio *DataNode*, mientras que en cada una de las otras máquinas del clúster se ejecuta el *DataNode*.

Figura 8. Arquitectura típica de un clúster HDFS



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

Todos los procesos de comunicación HDFS se basan en el protocolo TCP/IP. Los clientes HDFS se conectan a un puerto TCP (protocolo de control de transferencia) abierto en el *NameNode* y se comunican con él utilizando un

protocolo basado en RPC (*remote procedure call*). A su vez, los *DataNodes* se comunican con el *NameNode* usando un protocolo propietario.

RPC

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

4.2.2. Espacio de nombres (*namespace*)

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

Organización de datos

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

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

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

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

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

Los archivos de datos se dividen en bloques y se distribuyen mediante el clúster en el momento de ser cargados. Cada fichero se divide en bloques (del tamaño de bloque definido) que se distribuyen con el clúster.

Al distribuir los datos entre diferentes máquinas del clúster se elimina el *single point of failure* (SPOF) o único punto de fallo. En el caso de fallo de uno de los nodos no se pierde un fichero, ya que se puede recomponer a partir de sus réplicas. Esta característica también facilita el procesado en paralelo, porque diferentes nodos pueden procesar bloques de datos individuales de forma concurrente.

Ejemplo: proceso de creación de un fichero en HDFS

La figura 9 muestra a modo de ejemplo cómo un fichero dado se divide en cuatro bloques (B1, B2, B3 y B4), y en el momento de ser cargado en el HDFS se reparte entre los diferentes nodos. Se almacenan hasta tres copias de cada bloque debido a la replicación de datos (que se describirá en apartados posteriores).

Figura 9. Distribución de bloques de datos en un clúster HDFS entre los diferentes *DataNodes*

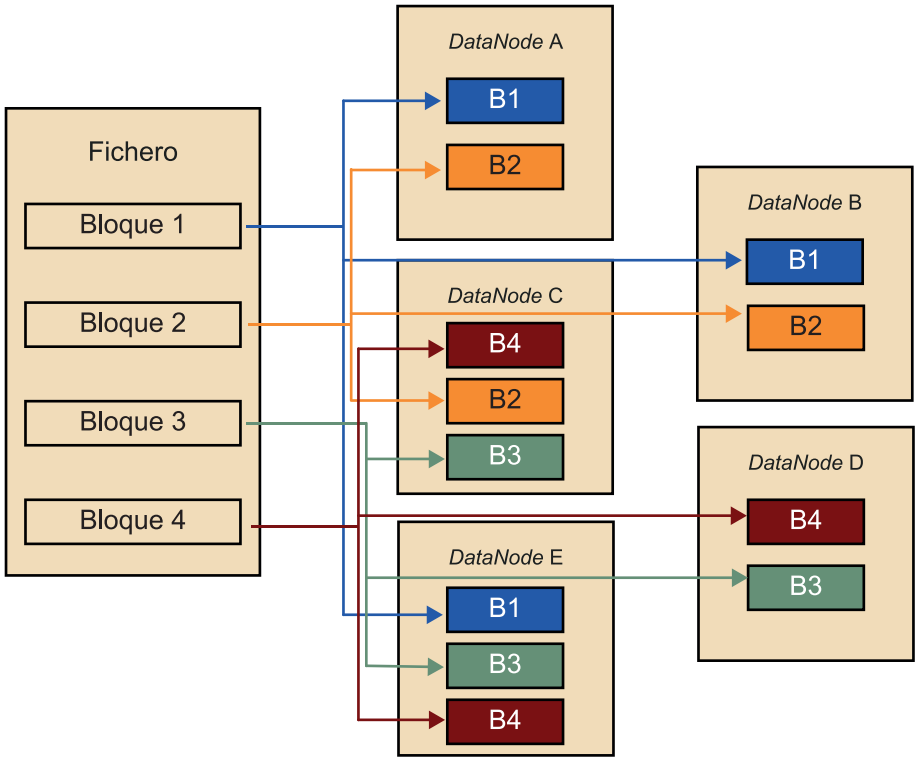


Figura 9

Nótese que en la figura se crean tantas copiar como réplicas tenga definidas el sistema.

Como ya se ha introducido anteriormente, el tamaño del bloque suele ser de 64 MB o 128 MB, aunque es altamente configurable por medio de los parámetros de configuración, que se encuentran en el fichero *hdfs-site.xml*. A continuación, se muestra un ejemplo de configuración de dicho valor:

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

Así, trabajar con ficheros grandes comporta varias ventajas:

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

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

Replicación de datos

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

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

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

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

```
<property>
    <name>dfs.replication</name>
    <value>3</value>
</property>
```

Ejemplo: almacenamiento de un fichero

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

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

Correspondencia entre fichero y bloques:

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

Correspondencia entre bloques y nodos:

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

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

Ejemplo: lectura de un fichero

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

En este caso:

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

El *NameNode* devuelve al cliente una lista con los nodos ordenados por proximidad a él, siguiendo el criterio que presentamos a continuación:

1) está en la misma máquina,

2) está en el mismo *rack* (*rack awarenes*),

3) se encuentra en alguno de los nodos restantes.

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

4.3. Fiabilidad

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

Disponibilidad (*secondary NameNode*)

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

- modo de alta disponibilidad
- modo clásico

En el **modo de alta disponibilidad** hay un *NameNode* activo, llamado principal o primario, y un segundo *NameNode* en modo *stand-by*, llamado *mirror*, el cual toma el relevo del *NameNode* principal en caso de fallo, en un procedimiento llamado *hot swap*.

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

- Mantiene una imagen del mapeo de ficheros con sus respectivos bloques de datos y su ubicación en los *DataNodes* (llamado *FsImage*).

- Los cambios en el sistema de archivos no se integran inmediatamente al archivo de imagen; en su lugar, el archivo de imagen principal se actualiza en modo *lazy* (es decir, solo bajo demanda).

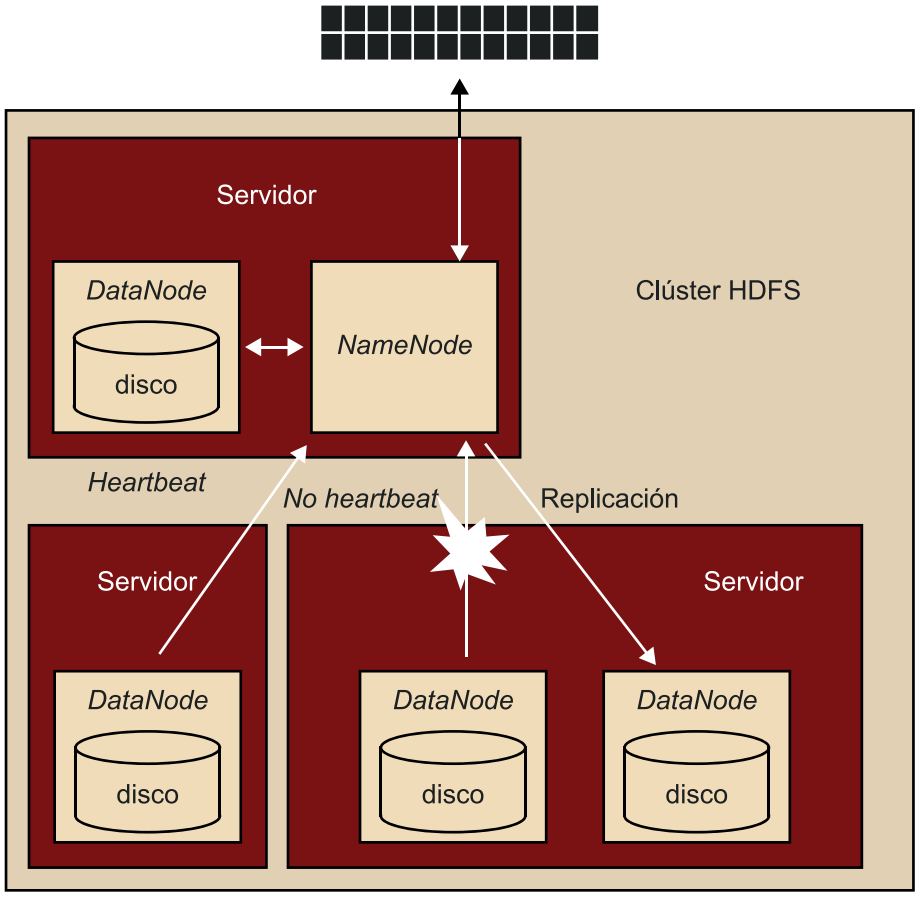
- Los cambios en el sistema de archivos se graban en unos archivos separados conocidos como *EditLogs*, análogos a un punto de control o *breakpoints* en un registro de transacciones de una base de datos relacional. En el *EditLog* se registra de forma persistente cada transacción que se aplica a los metadatos del sistema de archivos HDFS. Si los archivos *EditLog* o *FsImage* se dañan, la instancia de HDFS a la que pertenecen deja de funcionar. Por lo tanto, un *NameNode* admite múltiples copias de los archivos *FsImage* y *EditLog*, y cualquier cambio en cualquiera de estos archivos se propaga a todas las copias. Cuando un *NameNode* se reinicia, utiliza la última versión consistente de *FsImage* y *EditLog* para inicializarse. De forma periódica se fusionan los *EditLog* en el archivo de imagen principal *FsImage*, típicamente una vez por hora, lo que acelera el tiempo de recuperación si falla el *NameNode*. Cuando se inicializa un *NameNode*, lee el archivo *FsImage* junto con los *EditLogs* y aplica las transacciones y la información de estado que se encuentran registradas en dichos archivos.

A pesar de todo esto, el *NameNode* secundario no es un nodo de *backup* que reemplaza al *NameNode* en caso de fallo, de modo que en escenarios de alta disponibilidad no se suele usar.

4.3.1. HDFS *heartbeats*

Existen multitud de situaciones que pueden causar pérdida de conectividad entre el *NameNode* y los *DataNodes*. Así pues, para detectar dichos problemas HDFS utiliza los *heartbeats* (pulsaciones) que verifican la conectividad entre ambos. Cada *DataNode* envía mensajes de forma periódica (los *heartbeats*) a su *NameNode* y este puede detectar la pérdida de conectividad si deja de recibirlos. El *NameNode* marca como *DataNodes* caídos a aquellos que no responden a los *heartbeats*, y para de enviarles más solicitudes. Los datos almacenados en un nodo caído ya no están disponibles para un cliente HDFS desde ese nodo, que se elimina efectivamente del sistema. Si la caída de un nodo hace que el factor de replicación de los bloques de datos caiga por debajo de su valor mínimo, el *NameNode* inicia el proceso de creación de una réplica adicional para devolver el factor de replicación a un estado normal. Este proceso se ilustra en la figura 10.

Figura 10. Diagrama de replicación debido a un fallo en el *heartbeat*



4.3.2. Reequilibrado de bloques de datos

Los bloques de datos HDFS no siempre se pueden colocar uniformemente entre los *DataNodes*, lo que significa que el espacio utilizado para uno de dichos

nodos o más puede estar infrautilizado. Para evitar que esto ocurra, HDFS realiza tareas de equilibrado de bloques entre nodos utilizando varios modelos:

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

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

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

- políticas de replicación, descritas anteriormente,
- uniformidad en la distribución de datos entre nodos,
- reducción de utilización de ancho de banda por el tráfico de datos en la red interna.

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

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

Checksum

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

4.3.3. Permisos de usuarios, ficheros y directorios

HDFS implementa un modelo de permisos para archivos y directorios que tiene mucho en común con el modelo POSIX (*portable operating system interface*).

Por ejemplo, cada archivo y directorio está asociado a un propietario y a un grupo, de forma que admite permisos de lectura (r – *read*), escritura (w – *write*) y ejecución (x – *execution*). Debido a que no hay un concepto de ejecución de archivos dentro de HDFS, la *x* de ejecución tiene en un significado diferente. En este caso, el atributo *x* indica el permiso para acceder a un directorio hijo de un directorio padre determinado. El propietario de un archivo o directorio es la identidad del proceso cliente que lo creó y el grupo es el grupo del directorio padre.

4.4. Interfaz HDFS

Para acceder al servicio HDFS disponemos de varias alternativas. A continuación, presentamos las más utilizadas.

4.4.1. Línea de comandos

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

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

```
> hdfs dfs -help
```

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

```
> hdfs dfs -ls
```

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

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

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

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

Para crear un directorio, usaremos el comando:

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

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

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

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

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

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

```
> hdfs dfs –chmod rwx /directorio,
```

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

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

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

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

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

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

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

Comandos avanzados

Es posible combinar y descargar varios ficheros del sistema HDFS al sistema local en un solo fichero mediante el comando `getmerge`.

```
> hdfs dfs -getmerge <origen-hdfs-1> <origen-hdfs-2>
        ... <fichero-destino-local>
```

Este comando creará un solo fichero con una combinación de los ficheros de origen del HDFS.

4.4.2. Proyectos del ecosistema

Trabajar sobre el sistema HDFS mediante la línea de comandos es muy habitual por su similitud al funcionamiento de la línea de comandos de Unix. Sin embargo, el ecosistema Hadoop ofrece también un conjunto de herramientas que de algún modo permiten trabajar sobre el sistema de ficheros HDFS y que ofrecen funcionalidades diversas.

Flume

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

* https://flume.apache.org

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

La arquitectura de Flume la podemos dividir en los componentes siguientes:

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

- **Agente (*agent*)**. Es un proceso Java que se encarga de recoger eventos desde la fuente externa en un formato reconocible por Flume y pasárselos transaccionalmente al canal. Si una tarea no se ha completado, debido a

su carácter transaccional, esta se reintenta por completo y se eliminan resultados parciales.

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

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

* http://bit.ly/2pZH2Qc
** http://www.json.org
*** https://avro.apache.org

Sqoop

Apache Sqoop**** (SQL To Hadoop) es una herramienta cuyo propósito es el intercambio de datos entre bases de datos relacionales (RBDMS) y HDFS. Hace uso del modelo MapReduce para realizar la copia de datos desde una base de datos relacional, pero usa solo tareas *map* donde el número de *mappers* para realizar la conversión es configurable. Sqoop analiza el número de filas que se van a importar y divide la tarea entre los diferentes *mappers*, generando código Java (que se puede conservar para usos posteriores) para cada tabla que vamos a importar.

**** http://sqoop.apache.org

Del mismo modo que los datos pueden importarse al HDFS, estos pueden exportarse a una base de datos desde un repositorio HDFS.

La herramienta dispone de una consola para realizar las tareas de importación y exportación del tipo:

```
> sqoop import –connect jdbc:mysql:<params>
```

Ofrece compatibilidad con casi cualquier tipo de base de datos mediante una conexión JDBC genérica. No obstante, también dispone del llamado modo directo, en el cual el rendimiento es mucho mejor gracias al uso de utilidades propias de cada servidor de base de datos.

Aunque Sqoop puede ejecutarse en paralelo, cuanto mayor sea el paralelismo mayor será la carga sobre la base de datos, así que dicho paralelismo debe dimensionarse adecuadamente.

Hasta ahora Sqoop era una herramienta «cliente», es decir, el propio cliente se encargaba de conectarse a la base de datos (con usuario y contraseña) y al HDFS a la vez para realizar la transferencia. Así, apareció una mejora con Sqoop2, en la cual se define una nueva arquitectura cliente-servidor. El cliente, esta vez, solo necesita acceso al servidor, de modo que no tiene acceso directo a la base de datos y permite al administrador del sistema configurar Sqoop, limitando los recursos que se pueden utilizar, estableciendo una política de seguridad, etc.

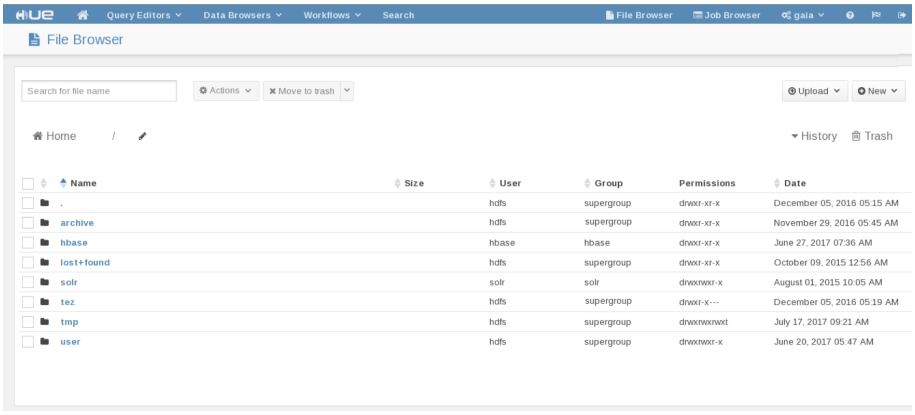
Hue

Hue (acrónimo de Hadoop User Experience*) es una aplicación web que permite ver y administrar los directorios y archivos en el sistema HDFS. Mediante HUE se pueden crear, modificar (solo directorios, no ficheros), mover, ver, subir, descargar y eliminar ficheros y directorios.

* http://gethue.com

Más allá de las funcionalidades de exploración del sistema HDFS, además permite gestionar el *job scheduler* y *job manager* (servicio YARN, que forma parte del ecosistema Hadoop), ya que observa cuáles son las tareas en ejecución en el clúster. También se integra con otras herramientas de consulta sobre el sistema de ficheros tales como Impala y Hive. Se puede ver el panel principal de HUE en la figura 11.

Figura 11. Panel principal de la aplicación HUE (en el menú superior pueden verse todas sus funcionalidades)



Otras herramientas

Podemos distinguir otras herramientas:

- El sistema HDFS puede ser montado como cualquier otro sistema de ficheros mediante la herramienta FUSE**, lo que una vez montado permite al usuario navegar por el sistema de ficheros con los comandos habituales `ls`, `cd`, etc.

** http://bit.ly/2Cha3wu

- HttpFS* es un servidor que proporciona una puerta de enlace HTTP REST que soporta todas las operaciones del sistema de archivos HDFS (lectura y escritura).

  HttpFS se puede utilizar para transferir datos entre clústeres que ejecutan versiones diferentes de Hadoop (superar problemas de versiones de RPC) o para acceder a datos en HDFS en un clúster detrás de un servidor de seguridad (el servidor HttpFS actúa como una puerta de enlace y es el único sistema que se permite cruzar el cortafuegos en el clúster).

  HttpFS se puede utilizar para acceder a datos en HDFS utilizando las utilidades HTTP (como `curl` y `wget`) y las bibliotecas HTTP de otros lenguajes.

*** http://bit.ly/2C2EGBR**

5. Bases de datos NoSQL

Las bases de datos relacionales son ampliamente usadas en todo tipo de sistemas y organizaciones desde la década de los setenta. Los sistemas gestores de bases de datos relacionales son altamente sofisticados y distan mucho de los primeros sistemas implementados en los albores del modelo relacional. Actualmente los sistemas relacionales están ampliamente extendidos, utilizan lenguajes de acceso y manipulación de datos estándares y están muy optimizados. Las ventajas de los sistemas relacionales son muchas y muy variadas, y algunas de ellas son las siguientes:

- proporcionan un modelo de datos formal;
- proporcionan un lenguaje de consulta y una manipulación estándar, cuya última versión fecha del 2016*;
- proporcionan interfaces de acceso comunes e interoperables: ODBC, JDBC, etc.;
- proporcionan un sistema transaccional que garantiza un alta consistencia de los datos;
- los datos almacenados en las bases de datos relacionales son altamente interoperables, ya que siguen el mismo modelo de datos;
- la curva de aprendizaje necesaria para aprender a utilizar una base de datos relacional es mínima, ya que los distintos sistemas gestores de bases de datos utilizan el mismo lenguaje para acceder y manipular los datos (SQL). Eso garantiza también que haya una gran cantidad de profesionales habilitados para trabajar con bases de datos relacionales de forma eficiente;
- proporciona herramientas de optimización muy potentes y facilita la personalización del modelo físico de los datos para optimizar aún más las operaciones de datos en función de las características de los problemas que se quieren tratar;
- existen diferentes extensiones al sistema relacional para suplir algunas de sus deficiencias en según que tipo de contextos: extensión geográfica, extensión XML, etc.

*** ISO/IEC 9075-1:2016: Octava revisión del estándar de SQL (véase http://bit.ly/2lx9nYW)**

Teniendo en cuenta la utilidad de los sistemas relacionales, la pregunta que se nos plantea es, ¿son las bases de datos relacionales la solución definitiva? ¿Debemos utilizarlas para resolver todo tipo de problemas? La respuesta evidentemente es no.

Con la explosión de internet y de los teléfonos inteligentes, entre otros, las bases de datos relacionales no siempre pueden dar respuesta a proyectos que requieren el almacenamiento de un gran volumen de datos (por lo general distribuidos) y su tratamiento en tiempo real. Es por ello que en la última década han aparecido otros tipos de base de datos, las llamadas bases de datos NoSQL y NewSQL, que pretenden suplir a las bases de datos relacionales en aquellos entornos donde su uso no es adecuado.

En este apartado nos centraremos en las bases de datos NoSQL. En particular, empezaremos describiendo el concepto de NoSQL y hablando un poco de sus orígenes y características. Más adelante identificaremos los modelos de datos que soporta, para luego describir en más detalle cada uno de ellos y mostrar, mediante un ejemplo, cómo representar los datos en los distintos modelos de datos.

5.1. ¿Qué es NoSQL?

El término NoSQL aparece por primera vez en 1998 para denominar una base de datos creada por Carlo Strozzi que no usaba el lenguaje SQL, y que tiene poco en común con las bases de datos NoSQL actuales. A partir de 2009 se popularizó esta denominación para referirse a una nueva generación de bases de datos no relacionales y altamente distribuidas que aparecieron durante la primera década del 2000.

Aunque muchas de las organizaciones que desarrollaron y popularizaron esta nueva tecnología son ahora grandes corporaciones (por ejemplo Google, Facebook o Amazon), la tecnología NoSQL ha estado siempre ligada al auge de las *startup* y al movimiento del código abierto. De hecho, la mayoría de bases de datos NoSQL son de código abierto.

Es importante destacar que la denominación *NoSQL* es polémica. Si revisamos los hechos, y con ello los problemas que llevaron al desarrollo de las bases de datos NoSQL, estos no están directamente relacionados con el lenguaje SQL, tal y como el nombre NoSQL puede sugerir. El hecho de que estos nuevos modelos de bases de datos se llamen NoSQL es accidental. Por tanto, siendo rigurosos, debemos tomar el término *NoSQL* simplemente como una etiqueta que identifica un conjunto de bases de datos de nueva aparición que no siguen el modelo de datos relacional, y no como un acrónimo que signifique «No se usa SQL», «No SQL», «No solo SQL», ni nada parecido.

Durante los últimos años, el auge de internet ha propiciado la aparición de aplicaciones web (o servicios web) que deben garantizar, idealmente, dos propiedades ante los fallos en la red o en los servidores de la aplicación: la garantía de que los datos estén disponibles aún cuando la red caiga o haya servidores del sistema inoperativos y la garantía de que los datos de la aplicación continuarán siendo consistentes (correctos) en caso de caída del sistema. En este contexto en el año 2000 se presentó la conjetura de Brewer, que más tarde en el 2002 se demostró y paso a ser el teorema CAP. Este teorema enumera tres propiedades deseables en un sistema distribuido: la consistencia, la disponibilidad y la tolerancia a particiones (sus acrónimos forman el término CAP). El teorema demuestra que, en un sistema distribuido, solo dos de estas propiedades pueden ser garantizadas de forma simultánea. Es decir, es imposible garantizar las tres a la vez. Esto ha dado pie a la creación de distintos sistemas gestores de bases de datos que permiten satisfacer diferentes combinaciones de dichas tres propiedades y así poder dar solución a distintos problemas de datos de forma eficiente.

En resumen, la necesidad de almacenar y gestionar grandes volúmenes de datos, altamente relacionados entre sí, en entornos distribuidos y en un ecosistema donde las aplicaciones tienen que responder rápidamente, de forma continuada y para todo el mundo, ha causado que nazcan nuevas generaciones de bases de datos.

Las primeras bases de datos NoSQL que se conocen son BigTable de Google en 2003 y Marklogic en 2005, aunque ya en 1989 existían bases de datos con características parecidas. No obstante, solo se utiliza el término NoSQL para bases de datos creadas a partir del año 2000. Desde entonces ha habido una gran proliferación de bases de datos NoSQL. Hoy en día existen cerca de doscientas bases de datos NoSQL con características muy variadas y con altos niveles de sofisticación.

Enlace de interés

Podéis ver una lista de los sistemas gestores de bases de datos NoSQL (y relacionales) ordenados por popularidad en el sitio web de dbengines: https://db-engines.com.

5.1.1. Características de las bases de datos NoSQL

Las principales características de las bases de datos NoSQL son:

- No ofrecen SQL como lenguaje estándar: las bases de datos NoSQL engloban varios tipos de bases de datos, cada uno con su lenguaje de consulta específico. En algunos casos el lenguaje utilizado puede ser parecido a SQL, incluyendo algunas extensiones específicas para NoSQL. Eso facilita el acceso a estas bases de datos a gente que sepa utilizar SQL. La gran mayoría de las bases de datos NoSQL se pueden utilizar mediante una API particular de programación, siendo este el modo de uso generalizado.

- El esquema de datos es flexible o no tienen un esquema predefinido (*schemaless*): se puede empezar a añadir datos sin definir previamente el es-

quema de los mismos, como se hace en el modelo relacional. Eso permite mucha más flexibilidad para tratar datos heterogéneos pero dificulta su programación, haciendo que la gestión del esquema (interpretación de los datos) se haga explícitamente en el código de los programas que acceden a la base de datos. El término *schemaless* es un poco abusivo en este contexto, porque aunque es cierto que no existe un esquema explícito, siempre hay un esquema implícito que se usa y que condiciona cómo se distribuirán los datos en las bases de datos NoSQL. Uno de los riesgos de que el esquema quede implícito es que se compromete la independencia de los datos, uno de los pilares sobre los que se fundamenta el desarrollo de las bases de datos relacionales.

- Las propiedades ACID de las transacciones (atomicidad, consistencia, aislamiento y definitividad) no siempre se garantizan por completo: esto se hace con el objetivo de mejorar el rendimiento y aumentar la disponibilidad. De hecho, las bases de datos NoSQL que promueven la alta disponibilidad acostumbran a seguir otro modelo transaccional denominado BASE.

- Permiten reducir, en parte, los problemas de falta de concordancia (*impedance mismatch*) entre las estructuras de datos usadas en los programas y las bases de datos. Es decir, el formato en que se guarda la información en las bases de datos es cercano al formato utilizado en los programas que acceden a ellas.

- La mayoría de ellas están especialmente diseñadas para crecer, generalmente de forma horizontal (mediante la fragmentación de los datos/esquema y la existencia de copias idénticas de los datos en múltiples servidores).

- Son generalmente distribuidas –con diferentes modelos de distribución (*peer-to-peer*, *master-slave*, etc.)– y de código abierto (con licencias dobles o basadas en soporte).

5.2. Modelos de datos NoSQL

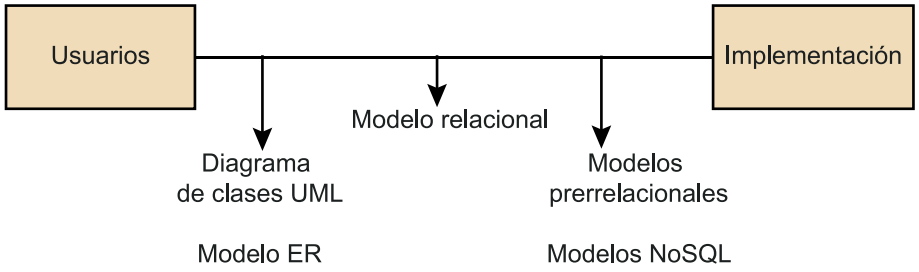
Un modelo es un esquema que nos permite analizar, conceptualizar y representar una parte de la realidad que nos interesa comprender con el objetivo de extraer conocimiento. Los modelos se utilizan en todas las disciplinas científicas y en ingeniería y, por lo tanto, también en informática.

Desde una perspectiva de ingeniería del software, un modelo conceptual de datos (o simplemente, un modelo conceptual) constituye una descripción de una parte del mundo real (o dominio de aplicación) que nos interesa analizar y conceptualizar.

Para construir modelos conceptuales se suelen usar modelos que se basan en lenguajes gráficos, dado que se orientan a personas. Su construcción surge

como consecuencia de un proceso de abstracción a partir de la observación del mundo real que nos interesa modelar. Como ejemplos de modelos que nos permiten realizar esta tarea, tenemos los diagramas de clases de UML y el modelo *entity-relationship* (o modelo entidad-interrelación, también conocido simplemente como modelo ER). UML se basa en el paradigma de orientación a objetos y es más expresivo que el modelo ER. Con los diagramas de clases de UML podemos representar clases de objetos del mundo real que tienen propiedades comunes y diferentes tipos de relaciones entre clases (por ejemplo, relaciones de herencia, asociaciones y diversos tipos de relaciones parte-todo). En la figura 12 podemos observar distintos modelos de datos, clasificados según su nivel de abstracción y con la indicación, para cada uno de ellos, de si está más orientado al usuario o a la implementación final.

Figura 12. Distintos modelos de datos clasificados según su nivel de abstracción



Una característica fundamental de los modelos conceptuales es que se aíslan de posibles tecnologías de representación. A pesar de ello, un modelo conceptual tiene que ser finalmente implementado, con el objetivo de que pueda ser utilizado por nuestros programas de aplicación. Cuando la tecnología de implementación es una base de datos, el modelo conceptual se tiene que transformar de acuerdo al modelo de datos en el que se basa la base de datos elegida.

Desde la perspectiva de bases de datos, un modelo de datos constituye el conjunto de elementos que nos proporciona el sistema gestor de la base de datos para implementar (o representar) nuestro modelo conceptual, e incluye (al menos desde una perspectiva clásica) los elementos siguientes:

1) estructuras de datos,

2) operaciones para consultar y modificar los datos,

3) mecanismos para definir restricciones de integridad que los datos deben cumplir.

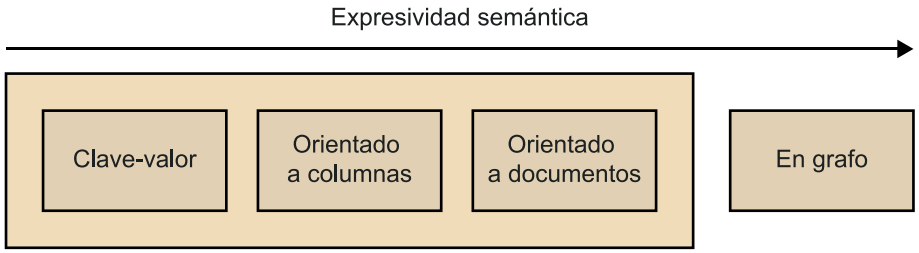
A modo de ejemplo, el modelo relacional incluye la relación como elemento de estructuración, lenguajes como el álgebra relacional y SQL para consultar y

modificar los datos, y la posibilidad de definir restricciones de integridad (por ejemplo, claves primarias y foráneas).

El modelo relacional no es el único modelo de datos, también existen los modelos prerrelacionales (basados en los modelos jerárquicos y en red) y los modelos de datos NoSQL. Una característica común a todos estos modelos es que están más próximos (con respecto al modelo relacional) al mundo de las representaciones informáticas.

Bajo el paraguas NoSQL subyacen principalmente dos familias de modelos de datos, el modelo de grafos y los modelos de agregación (figura 13). El modelo de grafos se orienta a representar dominios de aplicación donde se establecen múltiples y complejas interrelaciones entre los objetos del mundo real que deseamos representar. Por su parte, los modelos de agregación incluyen tres modelos: el clave-valor, el documental, y el orientado a columnas. En consecuencia, tenemos cuatro modelos de datos que se acostumbran a usar para clasificar las diferentes implementaciones de bases de datos NoSQL disponibles en el mercado.

Figura 13. Modelos de datos NoSQL más comunes



5.2.1. Modelos de datos NoSQL de agregación

Los modelos de agregación se basan en el concepto de agregado. Un agregado es una colección de objetos del mundo real relacionados que deseamos tratar como una unidad independiente (o atómica) desde un punto de vista de significado a efectos de:

1) Acceso y manipulación: la unidad mínima de intercambio de datos entre nuestros programas y el sistema gestor de la base de datos es el agregado.

2) Consistencia y control de concurrencia: asegurar que la definitividad de los cambios sobre el agregado y el mantenimiento de su integridad se circunscriben al ámbito del agregado. Si es necesario mantener restricciones de integridad entre diferentes agregados, la responsabilidad de asegurar la consistencia será de los programas y no de la base de datos.

3) Como unidad de distribución y replicación: si la base de datos está distribuida, el agregado se usa como unidad de distribución y replicación de los datos.

Si consideramos los tres elementos que desde un punto de vista teórico todo modelo de datos tiene que proveer, es importante destacar que cada modelo de agregación ofrece estructuras (en algunos casos, mínimas) para almacenar los datos. En relación con las operaciones para manipular y consultar los datos, cada fabricante ofrece, en general, su propio lenguaje (recordemos la falta de estandarización de las tecnologías NoSQL). Finalmente, los modelos de agregación delegan el mantenimiento de restricciones de integridad principalmente en los programas que acceden a la base de datos.

A continuación vamos a entrar en más detalle en cada uno de los modelos de agregación.

Modelos de agregación en clave-valor

El modelo clave-valor tiene su origen en la base de datos BerkeleyDB, que a día de hoy es propiedad de Oracle. Otros ejemplos de bases de datos NoSQL basadas en este modelo son DynamoDB, Redis y Riak.

Se trata del modelo de agregación más simple, dado que el agregado constituye un par (clave, valor), donde el valor contiene los datos del agregado y la clave permite identificarlo unívocamente; por tanto, es la que se utiliza para identificar y consultar los agregados. El sistema gestor de la base de datos desconoce la estructura interna asociada al agregado (el elemento valor). Esto no significa necesariamente que el agregado no tenga estructura, sino que esta solo será comprendida por los programas que manipulan los agregados.

Como el agregado es opaco para el SGBD, las consultas deberán realizarse por clave. Es conveniente que la clave de cada agregado tenga un significado dentro del dominio de interés que se va a modelar. Este sería el caso, por ejemplo, de claves como nombres de usuario, direcciones de correo electrónico, coordenadas cartesianas para el almacenamiento de datos de geolocalización, etc. Asignar claves aleatorias o que puedan olvidarse fácilmente puede ser problemático, y dificulta enormemente el acceso a datos previamente almacenados.

Este conjunto de características base se encuentra representada con distinta profundidad en las diferentes bases de datos NoSQL de tipo clave-valor. Por ejemplo, Riak y Redis permiten definir una estructuración mínima del agregado y representar explícitamente relaciones entre agregados. Recordemos que, en NoSQL, las bases de datos suelen ser aproximaciones a los modelos de datos. Esto significa que pueden tanto no implementarlos en su totalidad, como añadir funcionalidades extra. Este hecho puede provocar que algunas bases de datos NoSQL se puedan clasificar de diferentes maneras, según la fuente de información consultada.

Modelos de agregación orientado a documentos

Este modelo se considera un caso particular del modelo clave-valor, pero a diferencia del modelo clave-valor, en el modelo documental los agregados (que reciben el nombre de documento) tienen una estructura interna. Esta estructura interna simplifica el desarrollo de aplicaciones, pero reduce la flexibilidad del modelo clave-valor.

La estructuración interna puede ser aprovechada por el sistema gestor de la base de datos y por los lenguajes que ofrecen estas bases de datos. Así, por ejemplo, los documentos se pueden recuperar mediante su clave o mediante el valor que toman sus atributos. También es posible acceder a partes del documento y crear índices que ayuden a recuperar eficientemente los documentos almacenados en la base de datos. Los documentos se pueden agrupar en colecciones.

Aunque los documentos puedan tener una estructura interna, a diferencia del modelo relacional, no es necesario definirla de antemano, sino que será implícita y dependerá de cómo están estructurados los datos en los documentos. En consecuencia, distintos documentos que representen el mismo concepto del mundo real (por ejemplo, los datos de dos personas) pueden tener estructuras totalmente distintas o variaciones de la misma estructura.

La mayoría de bases de datos basadas en el modelo documental se caracterizan por almacenar documentos en formato JSON, pero también se admiten otros formatos, como sería el caso de XML.

Algunas de las bases de datos NoSQL más conocidas que implementan el modelo documental son MongoDB, CouchDB, Marklogic y RethinkDB.

Modelos de agregación orientado a columnas

Los modelos de agregación orientados a columnas están basados en el modelo de datos BigTable de Google. A pesar de ello, hoy en día, las bases de datos adheridas a este modelo han evolucionado y pueden presentar diferencias significativas con el modelo de datos motivador. Las bases de datos que siguen este modelo también reciben el nombre de almacenes para grandes datos (*big data stores*) y almacenes de registros extensibles (*extensible record stores*).

Conceptualmente, podemos ver este modelo como bidimensional (una matriz), donde cada fila de la tabla representa un agregado y es accesible a partir de una clave. Hasta ahora no hay ninguna novedad respecto a los modelos de agregación vistos anteriormente. No obstante, en este modelo, los datos de los agregados (es decir, de cada una de las filas de la tabla) se organizan en columnas.

Por lo tanto, un agregado es un conjunto de columnas, donde cada columna está formada por una tripleta compuesta por el nombre de la columna, el valor de la columna y una marca de tiempo (*timestamp*) que indica cuándo se añadió la columna en la base de datos.

Distintas columnas pueden agruparse en una nueva estructura, llamada normalmente familia de columnas. Una familia de columnas tiene un nombre y una semántica muy definida. Habitualmente, dentro de una agregación, una familia de columnas representa un concepto de la agregación. Por ejemplo, entendiendo un estudiante como un agregado, tres de sus familias de columnas podrían ser, respectivamente, sus datos personales (sexo, fecha de nacimiento, nombre), su domicilio (calle, código postal, municipio, provincia y país) y los estudios previos realizados por el estudiante (asignaturas cursadas previamente, resultados, año de superación, universidad donde se cursaron, etc.).

Una característica de este modelo es la facilidad que provee para acceder a un subconjunto de los atributos de un agregado (recordemos que los datos están organizados por columnas). Esto permite, por ejemplo, recuperar datos de manera eficiente, solo con los atributos relevantes en cada momento. Otra característica de este modelo es que, en comparación con el modelo relacional, no todas la filas de una misma tabla deben tener el mismo conjunto de columnas. El modelo por columnas puede ser un poco confuso al principio, pero es un modelo muy versátil y con muchas posibilidades.

Ejemplos de bases de datos que siguen el modelo de agregación orientado a columnas son Cassandra, HBase y Amazon SimpleDB.

5.2.2. Modelos de datos NoSQL en grafo

El modelo en grafo difiere totalmente de los modelos de agregación. En este modelo los datos no se almacenan mediante agregados, sino mediante grafos.

Como en el resto de modelos de datos NoSQL, no hay un modelo en grafo estándar. En nuestro caso vamos a suponer que el modelo en grafo responde a un grafo de propiedades etiquetado. Bajo esta asunción, los modelos en grafo están compuestos de nodos, aristas, etiquetas y propiedades:

¿Qué es un grafo?

Un grafo es una representación abstracta de un conjunto de objetos. Los objetos de los grafos se representan mediante vértices (también llamados nodos) y aristas. Los tipos, propiedades y algoritmos sobre grafos se estudian en una rama de las matemáticas denominada matemática discreta.

Los **nodos** o **vértices** son elementos que permiten representar conceptos generales u objetos del mundo real. Vendrían a ser el equivalente a las relaciones en el modelo relacional. No obstante, hay una diferencia importante: en el modelo relacional las relaciones permiten representar conceptos («actor», por ejemplo) y sus filas permiten representar instancias de los mismos, es decir, objetos del mundo real («Groucho Marx», por ejemplo). Por lo tanto, los conceptos y los objetos del mundo real se representan mediante construcciones

distintas. Sin embargo, en el modelo en grafo, los conceptos y los objetos del mundo real se pueden representar de la misma forma: mediante nodos. Es posible que no haya una diferenciación semántica de los mismos a nivel de modelo.

Las **aristas** o **arcos** son relaciones dirigidas que nos permiten relacionar nodos. Las aristas representan relaciones entre objetos del mundo real y serían el equivalente a las claves foráneas en el modelo relacional o a las asociaciones en los diagramas de clases de UML.

Las **etiquetas** son cadenas de texto que se pueden asignar a los nodos y a las aristas para facilitar su lectura y proveer mayor semántica.

Las **propiedades** son parejas <clave, valor>que se asignan tanto a nodos como a aristas. La clave es una cadena de caracteres, mientras que el valor puede responder a un conjunto de tipos de datos predefinidos. Las propiedades serían el equivalente a los atributos en el modelo relacional.

A pesar de que los nodos pueden emplearse para representar conceptos y objectos del mundo real, algunos sistemas gestores de bases de datos NoSQL en grafo, como por ejemplo Neo4J, utilizan las etiquetas para identificar los conceptos a los que pertenecen los objetos del dominio. Así pues, si tenemos el nodo que representa la asignatura de este curso, este nodo puede etiquetarse con la etiqueta «Asignatura» para indicar que el nodo es de tipo asignatura. En estos materiales se utilizará esta estrategia para representar los conceptos de la base de datos.

La característica principal del modelo en grafo es que las relaciones están explícitamente representadas en la base de datos. Eso simplifica la recuperación de elementos relacionados de forma muy eficiente. Esta eficiencia es mayor que en modelos de datos relacionales, donde las relaciones entre datos están representadas de forma implícita (claves foráneas) y requieren ejecutar operaciones de combinación (en inglés *join*) para calcularlas.

Además, la definición de relaciones es más rica en el modelo en grafo, debido a que pueden asignarse propiedades directamente a las aristas. Eso permite que las relaciones entre datos contengan propiedades y que provean una representación más natural, clara y eficiente de las mismas.

El modelo en grafo impone pocas restricciones de integridad. Básicamente dos, siendo la segunda consecuencia de la primera:

1) No es posible definir aristas (arcos) sin nodos origen y/o destino.

2) Los nodos solo pueden eliminarse cuando quedan huérfanos. Es decir, cuando no haya ninguna arista que entre o salga de los mismos.

Igual que los otros modelos de datos NoSQL vistos, este modelo es *schemaless*. Por lo tanto, no es necesario definir un esquema antes de empezar a trabajar con una base de datos en grafo. No obstante, en este caso, hay parte del esquema implícitamente definido en los grafos. Por ejemplo, los arcos que representen el mismo tipo de relación acostumbran a utilizar una etiqueta común para indicar su tipo y semántica (igual que los nodos). Como consecuencia de la característica anterior, este modelo permite añadir nuevos tipos de nodos y aristas fácilmente, y sin realizar cambios en el esquema. Aunque es conveniente hacerlo de forma ordenada y planificada, para evitar añadir tipos de relaciones o nodos redundantes o mal etiquetados.

Los sistemas de gestión de bases de datos que implementan un modelo en grafo acostumbran a proporcionar lenguajes de más alto nivel que los basados en modelos de agregación. Por ejemplo, podemos encontrar el uso de Cypher* en Neo4J, que es un lenguaje declarativo parecido a SQL, y Gremlin**.

*** http://bit.ly/2ElTZY1**
**** http://bit.ly/1pGhRK6**

Cypher

Cypher es un lenguaje de consulta y manipulación de grafos creado soportado por Neo4j.

Gremlin

Gremlin es un lenguaje de dominio de gestión de grafos y que está soportado por varios sistemas gestores de bases de datos en grafo.

Ejemplos de bases de datos NoSQL basadas en este modelo son Neo4j, OrientDB, AllegroGraph y TITAN.

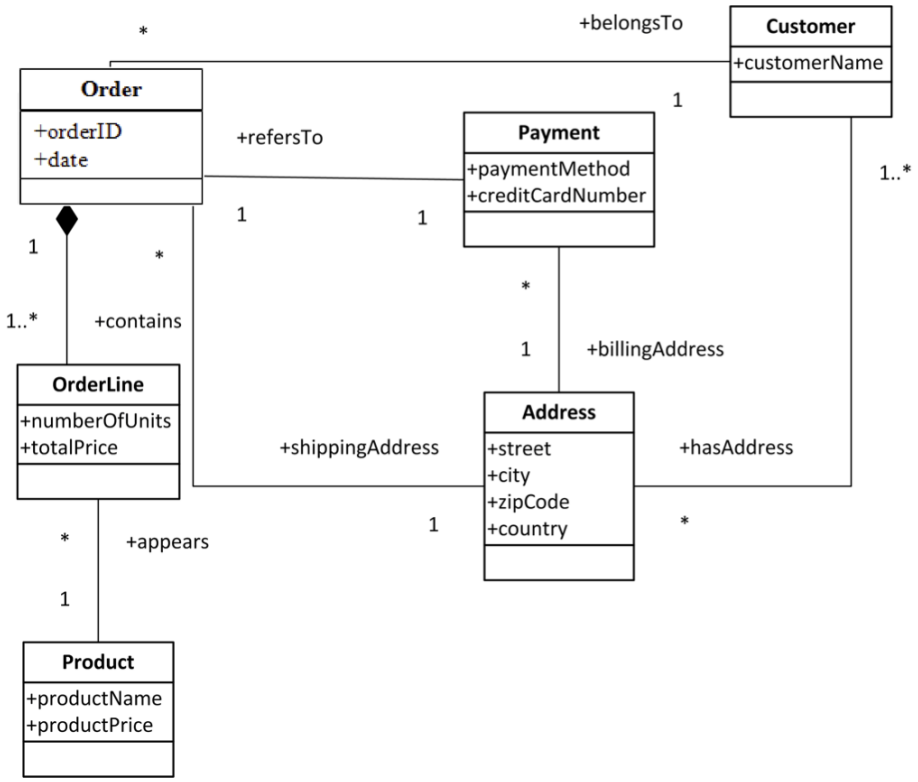
5.2.3. Ejemplo práctico

A continuación, presentaremos un modelo conceptual y explicaremos cómo representarlo en los diferentes modelos NoSQL presentados. El objetivo es mostrar, mediante un ejemplo, cómo se distribuyen los datos en los distintos tipos de bases de datos NoSQL.

El ejemplo que os proponemos constituye una simplificación del carrito de la compra que se utiliza en aplicaciones de *e-commerce* (figura 14).

En primer lugar tenemos clientes (*customer*). De los clientes podemos tener registradas entre cero y múltiples direcciones postales (*address*). Los clientes también pueden haber efectuado entre cero y múltiples pedidos (*order*). De todos los clientes guardamos su nombre. Que un cliente no tenga pedidos (o dirección postal) se puede relacionar, por ejemplo, con el hecho de que el cliente alguna vez se ha registrado en el sistema pero que, por los motivos que sea, no ha completado ningún proceso de compra. Las direcciones quedan caracterizadas por la calle, ciudad, código postal y país. No se guardan direcciones que no pertenezcan a ningún cliente, y podemos tener clientes que compartan domicilio. Un pedido se realiza en una fecha. Los pedidos tienen una dirección postal de envío y datos asociados para efectuar el pago (*payment*). A su vez, cada pago tiene una dirección asociada para el envío de la factura y datos sobre cómo se efectuará el pago (tipo y número de tarjeta).

Figura 14. Esquema conceptual en UML que indica los datos necesarios para gestionar un carrito de la compra



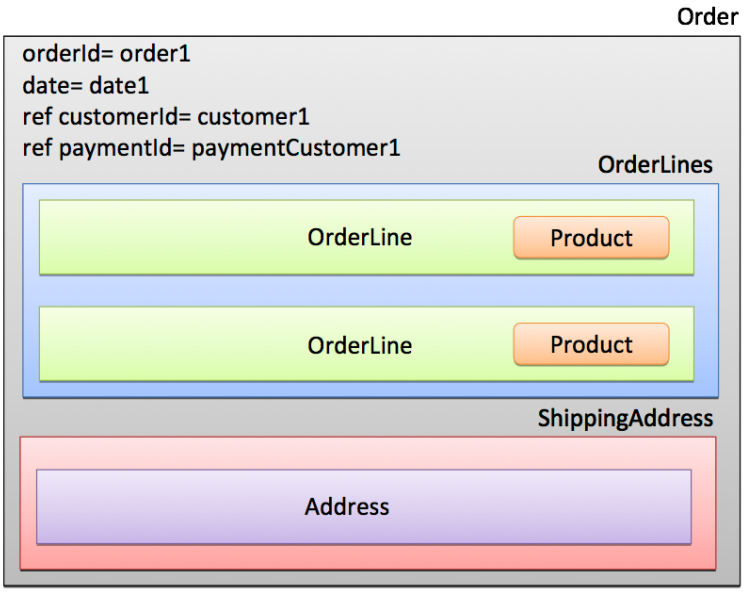
Cada pedido es un objeto compuesto que incluye, al menos, una línea de pedido (*orderline*). El hecho de que sea un objeto compuesto se indica mediante el diamante negro que en UML representa una relación de composición. Esta es un tipo de relación parte-todo, cuya semántica implica que las líneas del pedido no tienen sentido sin el objeto en el que se agregan (el pedido). En otras palabras, cada línea de pedido solo puede formar parte de un pedido y, si el pedido se elimina, también se eliminarán sus líneas. Cada línea de pedido hace referencia a un producto (*product*). También incluye cuántas unidades se adquieren del producto y el precio total. De los productos se guarda su nombre y precio unitario. Hay productos que no aparecen en ninguna línea de pedido (y, en consecuencia, tampoco en ningún pedido), por ejemplo, porque o no han sido adquiridos por nadie o porque todavía no se han puesto en oferta.

Imaginemos que sobre este modelo conceptual queremos conocer la información relativa a un pedido concreto a efectos de gestionar su envío. Esta información será más fácil o difícil de recuperar en función del tipo de base de datos elegida para implementar este modelo conceptual.

Para recuperar la información requerida, sería posible diseñar un agregado que incorpore todos los datos necesarios (figura 15). Este agregado (que representa el objeto compuesto pedido) incluye los atributos propios del pedido (su identificador y la fecha en la que se realiza) y las referencias a los agregados que representan los datos de pago y del cliente que efectúa el pedido (dichas referencias nos permitirían navegar a dichos agregados). Adicionalmente, cada

pedido agrega toda la información sobre las líneas de pedido (y los productos a los que hacen referencia), así como la dirección postal de envío.

Figura 15. Diseño del carrito de la compra utilizando agregados



Dado que el agregado es la unidad mínima de acceso y manipulación, obtendríamos toda la información deseada de forma eficiente mediante una sola operación. Con respecto a una base de datos relacional, nos estamos ahorrando las operaciones de combinación. Esto es debido al hecho de que un modelo de agregación se basa en la aplicación de técnicas de desnormalización.

Si la base de datos estuviese distribuida, el agregado se habría utilizado como unidad de distribución de los datos, garantizando de esta manera que estaría almacenado de manera conjunta en alguna de las bases de datos que conforman el sistema distribuido.

El diseño base del agregado puede ser el mismo para las representaciones de los modelos clave-valor, documental y de columnas. A pesar de ello, deberá adaptarse en cada caso, como se muestra a continuación.

Representación mediante un modelo de tipo clave-valor

En el caso de escoger un modelo clave-valor utilizaremos el agregado propuesto como valor y escogeremos el atributo identificador de pedido como atributo clave (figura 16), ya que es un atributo que identifica de forma unívoca los pedidos y es conocida. Si alguien la olvida, simplemente debería recuperar información —escrita o en cualquier formato— sobre el pedido para recuperarla.

Figura 16. Diseño del carrito de la compra utilizando un modelo clave-valor

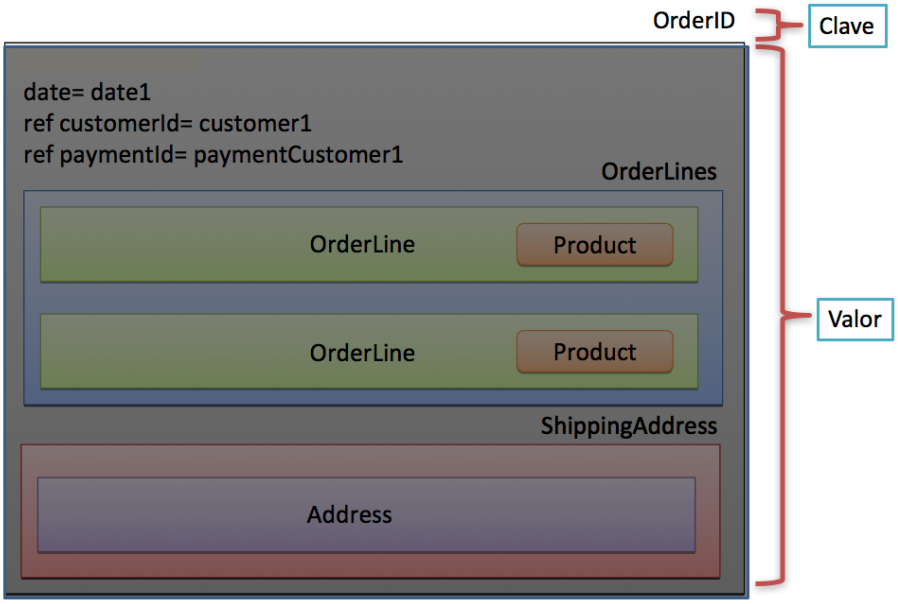


Figura 16

El contenido del agregado (aún teniendo una estructura) no será visible para el sistema gestor de la base de datos, como se indica mediante un sombreado negro del mismo.

A efectos de la base de datos, el valor del agregado será opaco. Es decir, la base de datos no será consciente de qué datos contiene el agregado ni sobre cómo están estructurados. Por tanto, será la aplicación que consulta la base de datos quien deberá interpretar el agregado y darle forma. Para ello, la aplicación deberá tener información sobre qué datos están almacenados en el agregado y con qué estructura. Con dicha información, la aplicación será capaz de interpretar y trabajar con los datos almacenados. El hecho de que el valor del agregado sea opaco para la base de datos proporciona más flexibilidad al almacenar los datos, ya que se podría, si fuera conveniente, almacenar documentos binarios (el PDF de una factura, por ejemplo) como valor de un agregado para algunos registros del sistema.

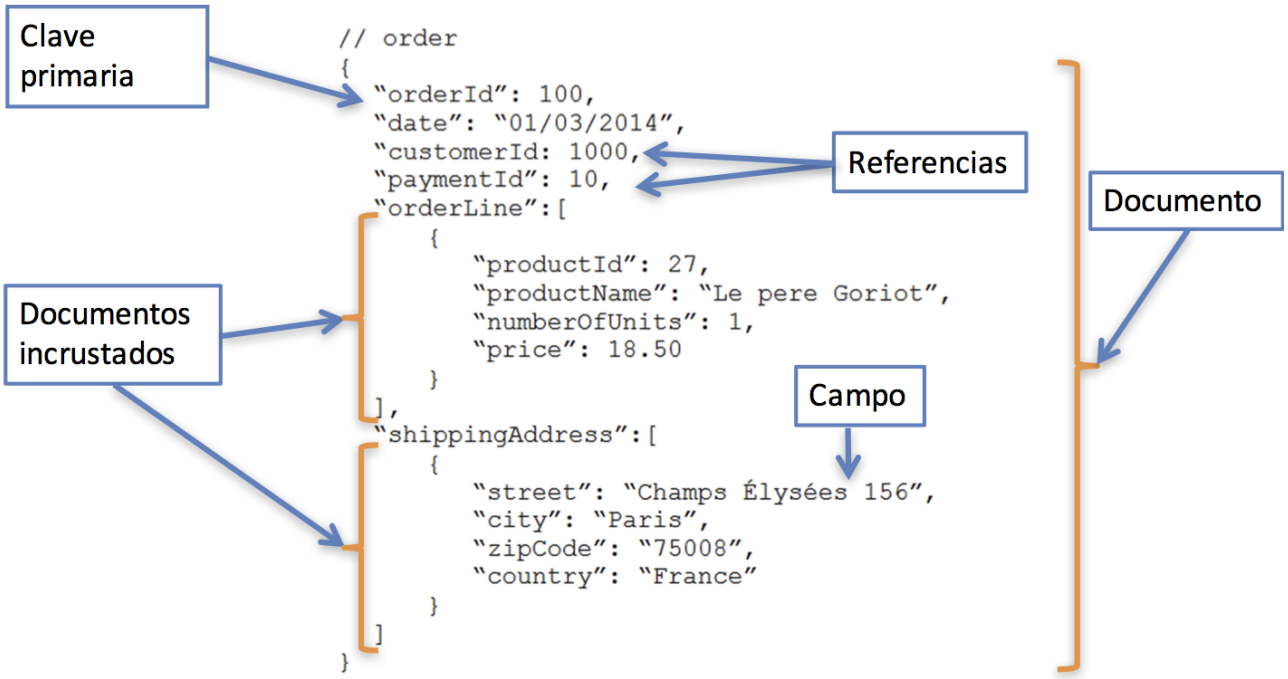
Notad que en un modelo clave-valor puro, si se quiere definir relaciones entre distintos agregados debe hacerse de forma implícita. Es decir, añadir dentro del agregado las claves que referencian a los agregados relacionados. Para explotar dichas relaciones, los programas que consulten el agregado deberán interpretar sus datos, extraer las claves de los agregados relacionados y consultar dichos agregados en la base de datos mediante las nuevas claves. Un proceso poco eficiente que desaconseja el uso de modelos clave-valor cuando el número de relaciones en los datos sea elevado.

Representación mediante un modelo de tipo documental

En el caso de utilizar un modelo de agregación documental, los diferentes pedidos se almacenarían en una colección, que en nuestro caso hemos denominado *order*.

Cada pedido estará representado por un documento en formato JSON (figura 17). El documento de ejemplo contiene una clave primaria (*_id*) con valor *order1*, un campo que permite indicar la fecha en que se efectúa el pedido y un par de referencias que apuntan a otros documentos (agregados) que contienen información del cliente que realizó el pedido y de los datos del pago. Estas referencias podrían haberse definido de forma distinta, ya que podría haber incluido toda la información del cliente y de los datos de pago dentro del documento actual. No obstante, se ha preferido utilizar referencias y gestionar los datos de clientes y de pago mediante documentos aparte para ejemplificar esta opción. En el pedido también podemos ver ejemplos de (sub)documentos incrustados como, por ejemplo, las líneas del pedido y la dirección de envío del pedido.

Figura 17. Diseño del carrito de la compra utilizando un modelo documental



Notad que el hecho de utilizar referencias a otros agregados y no incluir todos los datos relevantes dentro del agregado puede ralentizar las consultas sobre el agregado. No obstante, hacerlo puede ser conveniente en el caso de que los datos de los agregados referenciados aparezcan en muchos agregados y se modifiquen con asiduidad, ya que así se reduce el número de actualizaciones que se deben realizar cada vez que se modifica su valor.

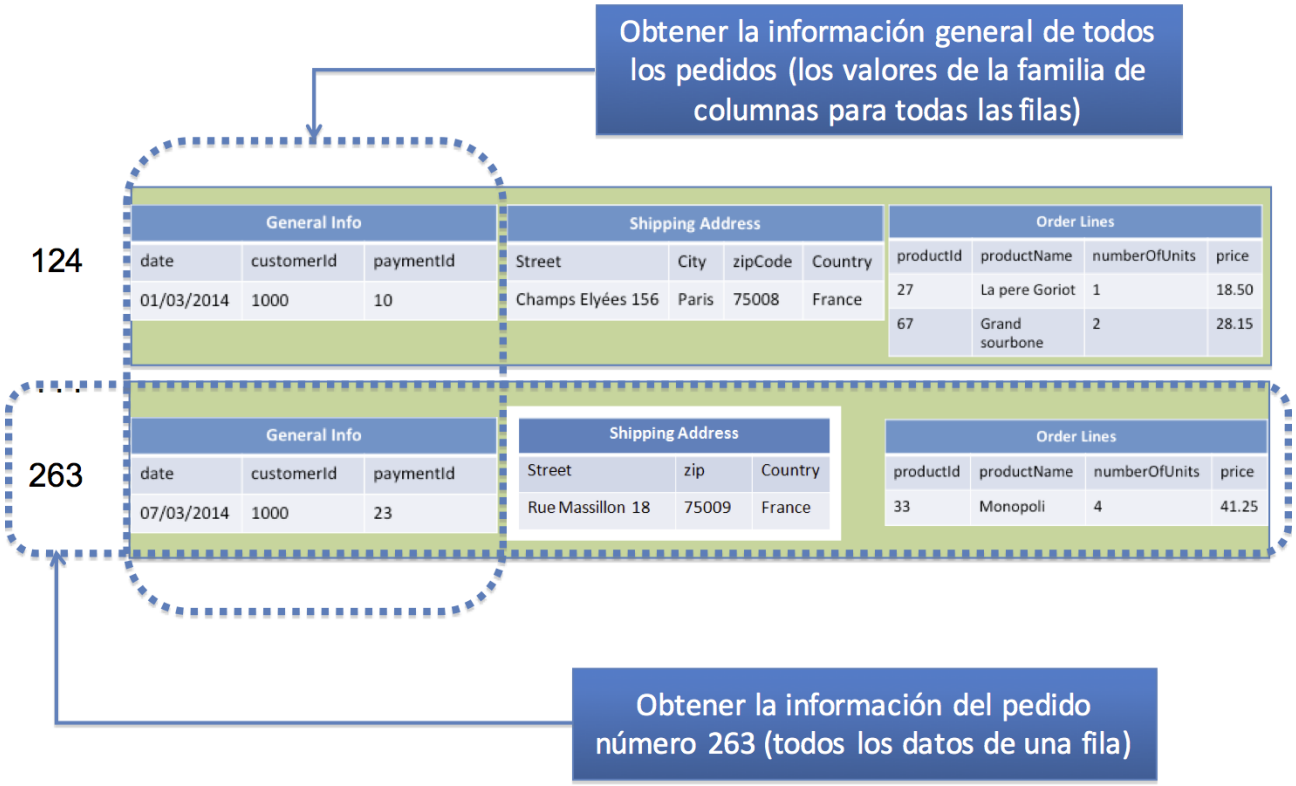
Representación mediante un modelo orientado a columnas

En este caso, igual que en los anteriores, se ha elegido el campo *orderId* como identificador de agregado. Posteriormente, se han agrupado las columnas en distintas familias en base a su significado. Cada familia de columnas tiene un

nombre que la identifica. Se han creado tres familias de columnas, una para representar la información general del pedido (compuesta por las columnas fecha, identificador de cliente e identificador de pago), otra para representar la dirección de entrega (compuesta por las columnas calle, ciudad, código postal y país) y la última para representar las líneas de pedido, compuestas por los identificadores de producto, los nombres de productos, el número de unidades y el precio final.

En la figura 18 podemos ver los agregados de dos pedidos representados mediante el esquema propuesto. Los dos agregados del ejemplo responden a dos pedidos de un mismo cliente.

Figura 18. Diseño del carrito de la compra mediante un modelo orientado a columnas



Las columnas representan los atributos del pedido, como pueden ser la fecha, el cliente, o el domicilio donde se ha enviado. Podemos ver que el número de columnas de los agregados puede ser diferente. El primer agregado tiene una columna ciudad (*city*), mientras que el segundo agregado no la tiene. Otra particularidad es que las columnas pueden tener distinto número de valores en distintos agregados, tal y como podemos ver en el primer agregado, que tiene dos valores para la columna *productId*, mientras que el segundo agregado solo tiene una. Esto también sucede en las familias de columnas. Notad que algunas familias de columnas tienen solo un valor (información general y dirección de entrega), mientras que otras pueden almacenar múltiples valores (líneas de producto).

En un modelo de agregación de columnas se puede acceder a una fila (o agregado) concreto a partir de su clave (por ejemplo, se podría acceder el agregado del segundo pedido a partir de su identificador, con valor 263). Por otro lado, también es posible obtener información sobre una familia de columnas para todos los agregados (por ejemplo, se podrían obtener los valores de la familia de columnas *general info* para todos los agregados. Evidentemente, también puede obtenerse información mediante la combinación de las dos operaciones anteriores.

Representación mediante un modelo en grafo

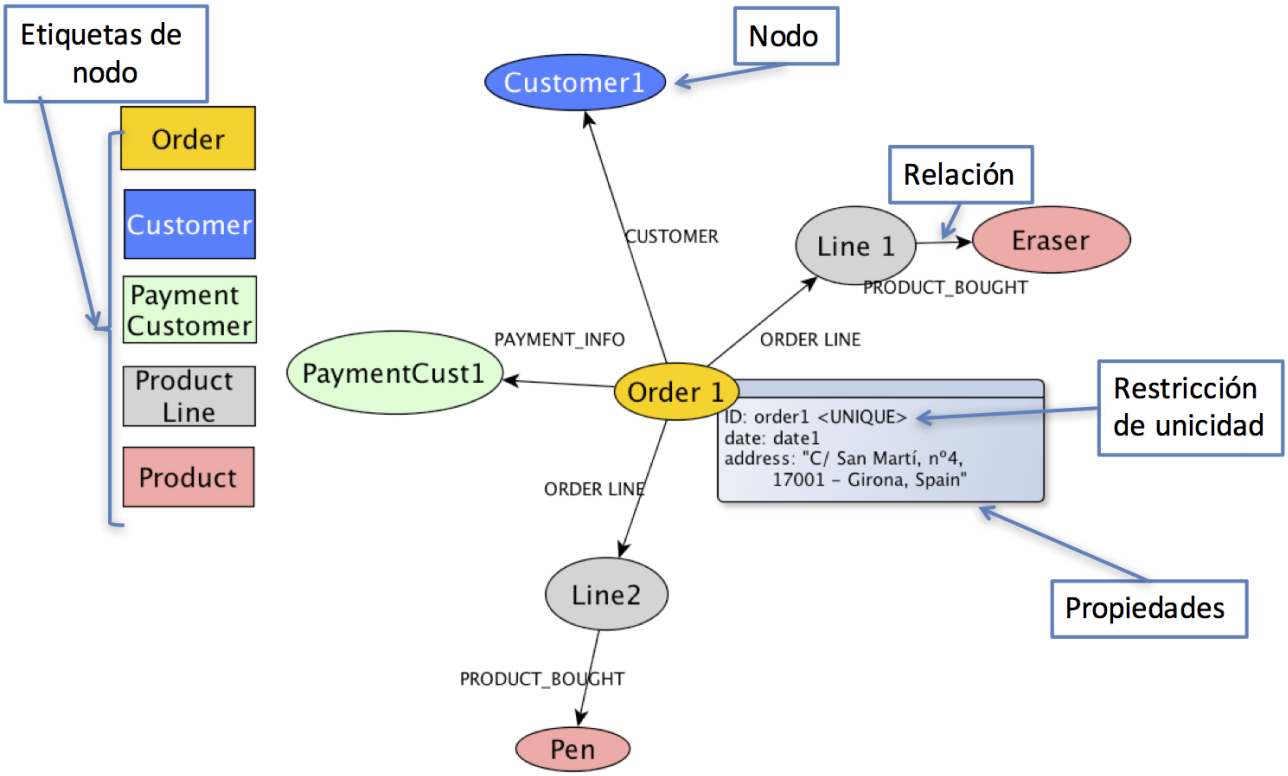
La representación de los pedidos utilizando un modelo en grafo diferirá mucho de las representaciones de agregados vistas hasta ahora, ya que el objetivo de este modelo no es representar los datos de forma agregada, sino de forma dispersa pero relacionada.

En la figura 19 podemos ver una posible representación de los pedidos mediante una base de datos NoSQL en grafo. Los pedidos se almacenan como nodos y cada pedido constituye un nodo distinto. En este caso, podemos ver cómo el nodo *order 1* determina el pedido que hemos estado utilizando en el ejemplo. Véase también que se ha utilizado una etiqueta llamada *order* para indicar que el nodo *order 1* es de tipo pedido. Una vez definido el nodo del pedido y su tipo, se pueden indicar sus datos mediante propiedades. En el ejemplo podemos ver que se han definido tres propiedades: el identificador del pedido (*ID*), la fecha del pedido (*date*) y la dirección de envío (*address*). Además, en este caso, hemos decidido definir en la propiedad ID una restricción de unicidad, por lo tanto, no se permitirán dos propiedades *ID* con el mismo valor en nodos de tipo *order*. Las líneas de pedido se han representado mediante un conjunto de nodos y relaciones, pero también podrían haberse representado mediante propiedades. Expresarlo de una forma u otra constituye una decisión de diseño de la base de datos y deberá escogerse en función del uso esperado de la base de datos.

El cliente que realizó el pedido se ha definido mediante un nodo *customer 1* de tipo *customer*. Para indicar el hecho de que *customer 1* es el cliente que realizó el pedido, se ha creado una relación con nombre *customer* entre los nodos *order 1* y *customer 1*. De forma parecida se ha creado un nodo para los datos de pago llamado *paymentCust 1* y una relación para indicar que dichos datos de pago son relativos al pedido *order 1*. Notad que la nomenclatura utilizada para los nodos y las relaciones son diferentes: los primeros se indican con la primera letra en mayúscula y los segundos con todas las letras en mayúscula*.

*** La nomenclatura utilizada en este ejemplo es la utilizada por el sistema gestor de bases de datos Neo4j.**

Figura 19. Diseño del carrito de la compra mediante un modelo en grafo



Respecto a las líneas de pedido, se ha creado un tipo de nodo llamado *product line* y se han definido dos nodos de este tipo para las líneas de pedido: *line 1* y *line 2*. Dichas líneas se han relacionado con el pedido mediante dos relaciones, ambas llamadas *order line*. Fijaos en que aunque diferentes nodos del mismo tipo contengan nombres distintos, las relaciones comparten nombre. Eso es porque el nombre de la relación se utiliza en Neo4j para identificar su tipo. Es decir, como las relaciones entre *order 1* y *line 1* y entre *order 1* y *line 2* tienen el mismo nombre (ese nombre es *order line*), eso significa que son del mismo tipo (*order line*). Esto nos resultará de utilidad más adelante a la hora de consultar el grafo, ya que nos permitirá realizar consultas que satisfagan un patrón, como por ejemplo «Dame todas las líneas de pedido del producto Eraser». El patrón detrás de esta consulta sería «los nodos de tipo *product line* relacionados mediante una relación *order line* con un nodo cuyo nombre es *Eraser*». Los nodos del grafo que verifiquen este patrón satisfarán la consulta anterior.

Los productos que aparecen en cada línea de pedido se han definido mediante un nuevo tipo de nodo denominado *product*. En particular, se han definido dos nodos de este tipo: *Eraser* y *Pen*. Para indicar que estos nodos aparecen en una línea de pedido se ha creado dos relaciones de *product bought* que los relacionan con las líneas de pedido en las que aparecen.

En este ejemplo, hemos creado una versión simplificada del posible grafo resultante, obviando algunos nodos y propiedades que podrían haberse añadido. Aún siendo incompleto, el número de nodos y relaciones creados en el

ejemplo es elevado. En un caso real el número de elementos del grafo crece rápidamente. Por ese motivo, será necesario hacer un buen diseño del grafo para identificar qué conceptos deben ser representados mediante nodos, cuáles mediante relaciones y cuáles mediante propiedades. Asimismo, es importante hacer una buena gestión de índices para optimizar la consulta de los datos a partir del valor de sus propiedades.

5.3. Ventajas e inconvenientes de las bases de datos NoSQL

A continuación, enumeramos brevemente las ventajas e inconvenientes de las bases de datos NoSQL respecto las bases de datos relacionales. Cabe destacar que, dada la gran diversidad que existe en las bases de datos NoSQL, es imposible enumerar ventajas e inconvenientes que sean ciertas para el 100 % de todas ellas. Lo que sea válido para algunas (de agregación) puede no ser válido para otras (en grafo). Además, cabe tener en cuenta que el movimiento NoSQL va incorporando más tipos de datos con características propias: *event oriented*, multimodelo, etc.

Si hablamos de bases de datos NoSQL de agregación, las ventajas más relevantes son el soporte de la distribución y replicación de datos de forma más natural, el tratamiento más eficiente de grandes volúmenes de datos, una mayor tolerancia a fallos, la posibilidad de escalar horizontalmente y de permitir almacenar la información de forma desnormalizada (eso puede ser también un inconveniente) —de manera que se permite también así tener preparados los datos en la base de datos para su posterior consumo—, la posibilidad de ofrecer un esquema de datos más flexible, capacidades de corregir problemas sin la intervención del administrador de la base de datos (caídas de servidores por ejemplo) y que, al haber distintos tipos de modelos, permiten adaptarse muy bien a distintos problemas.

En las bases de datos NoSQL en grafo, las ventajas son claras: soportan más naturalmente información relacionada, permiten representar más eficientemente relaciones y permiten consultar datos relacionados de forma más fácil y eficiente.

Los principales problemas de las bases de datos NoSQL son quizá la falta de estándares en sus lenguajes de manipulación y consulta de datos, su rápida y caótica evolución, la dificultad de programar y mantener dichos sistemas (conceptualmente parece que la flexibilidad de esquema puede ser un verdadero problema para la evolución de los programas, aunque tenemos que esperar algún tiempo para ver hasta qué punto y cómo se soluciona), la falta de soporte en algunos casos, la baja consistencia que ofrecen (básicamente las de agregación) y la reticencia al cambio que pueden encontrarse en las empresas. Otro punto flaco de las bases de datos NoSQL fundamental es la seguridad de los datos. Por un lado, porque no son productos tan maduros como sus

homólogos relacionales y, por otro, porque cuanta más distribución y replicación haya, más potenciales agujeros de seguridad existen y más puntos de acceso a los datos disponibles para posibles atacantes.

Otro problema de las bases de datos deriva de su gran variabilidad y del elevado número de productos disponibles, actualmente más de doscientos. Esto puede provocar dispersión, una evolución de productos más lenta, más dificultad en encontrar profesionales cualificados para cada producto, una curva de aprendizaje más acusada, dificultades de interoperabilidad y compatibilidad y una percepción de inmadurez. Parece coherente pensar que en el futuro debería haber una racionalización de los productos NoSQL.

Hay algunos aspectos que aparecen el la bibliografía de NoSQL pero que, bajo el punto de vista del autor, no pueden clasificarse actualmente como ventajas ni como inconvenientes. Son los aspectos siguientes:

- La falta de madurez de la tecnología NoSQL: actualmente existen productos muy potentes y maduros en el mercado y que han sido utilizados en casos de éxito impactantes y claros. Ejemplos son Cassandra, MongoDB, Riak, DynamoDB o Neo4j. Por otro lado, a diferencia de años atrás, actualmente la mayoría de herramientas de inteligencia de negocio soportan sistemas NoSQL.

- El ahorro debido a su gratuidad y disponibilidad de código: el hecho de que la mayoría sea de código libre puede ser una ventaja, pero también un inconveniente, y no garantiza que el coste de implantación o mantenimiento sea menor, ya que el coste total de propiedad puede ser mayor. Además, algunos de los sistemas gestores de bases de datos NoSQL tienen una versión profesional de pago que pueden hacer que su implantación sea más costosa que la de una base de datos relacional.

- Dificultad de relacionar datos (o realizar *joins*): si se hace un buen diseño y se utilizan distintos modelos de bases de datos de forma combinada para representar y manipular los datos de forma eficiente, lo que se llama persistencia políglota*, los datos ya estarán preparados para su consumo y ello facilitará su consumo.

*** Más información en el siguiente vídeo: https://vimeo.com/121661296**

Persistencia políglota

Se refiere a la estrategia de utilizar la base de datos más adecuada para cada problema. En problemas complejos puede implicar utilizar más de una base de datos de forma coordinada.

Resumen

En este módulo didáctico hemos presentado las tareas y arquitecturas asociadas a la captura, el preprocesamiento y el almacenamiento de datos masivos (*big data*).

En la primera parte de este módulo hemos discutido acerca de los procesos de captura y preprocesamiento de datos masivos, haciendo especial énfasis en el concepto de datos generados de forma continua (*streaming data*) y cómo pueden capturarse, distribuirse y almacenarse en entornos distribuidos.

En este sentido, se ha profundizado en el envío de datos en *streaming* desde los productores a sus potenciales consumidores. De todos ellos, se ha puesto especial foco en los sistemas de mensajería, que ofrecen una mayor escalabilidad y disponibilidad. Como ejemplo de estos sistemas, se ha mostrado el funcionamiento básico de Apache Kafka.

En la segunda y última parte de los materiales se ha tratado en más detalle los sistemas que permiten almacenar datos masivos. Concretamente, hemos visto los sistemas de ficheros distribuidos, su arquitectura y funcionalidades básicas, centrándonos en el sistema de ficheros distribuidos de Hadoop (*Hadoop Distributed File System*, HDFS).

Hemos finalizado este módulo introduciendo las bases de datos NoSQL. Concretamente, hemos revisado los distintos modelos existentes y las principales ventajas e inconvenientes de cada uno de estos modelos.

Glosario

checksum *f* Véase **suma de verificación**.

commodity hardware *sust.* La computación de «bajo coste» implica el uso de un gran número de componentes de computación ya disponibles para la computación paralela con la intención de obtener la mayor cantidad de computación útil a bajo coste.

Cypher *sust.* Lenguaje de consulta y manipulación de grafos creado soportado por Neo4j.

datos estructurados *m pl* Datos que siguen un patrón igual para todos los elementos y que, además, es conocido *a priori*. Por ejemplo, los datos de una hoja de cálculo presentan los mismos atributos para cada fila.

datos no estructurados *m pl* Datos que no siguen ningún tipo de patrón conocido *a priori*. Por ejemplo, dos documentos de texto o imágenes.

datos semiestructurados *m pl* Forma de datos que no contiene una estructura fija predefinida *a priori*, pero que contiene etiquetas u otros marcadores para separar los elementos semánticos y hacer cumplir jerarquías de registros y campos de los datos. Por ejemplo, las documentos JSON o HTML.

grafo *m* Representación abstracta de un conjunto de objetos. Los objetos de los grafos se representan mediante vértices (también llamados nodos) y aristas. Los tipos, las propiedades y los algoritmos sobre grafos se estudian en una rama de las matemáticas denominada matemática discreta.

Gremlin *sust.* Lenguaje de dominio de gestión de grafos que está soportado por varios sistemas gestores de bases de datos en grafo.

invocación a procedimiento remoto *f* En computación distribuida, programa que utiliza una computadora para ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambas. *en* Remote procedure call

metadato *m* Dato que describe otros datos. En general, un grupo de metadatos se refiere a un grupo de datos que describen el contenido informativo de un objeto al que se denomina recurso.

persistencia políglota *f* Se refiere a la estrategia de utilizar la base de datos más adecuada para cada problema. En problemas complejos puede implicar utilizar más de una base de datos de forma coordinada.

remote procedure call *f* Véase **invocación a procedimiento remoto**. sigla **RPC**

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

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

Bibliografía

Apache Kafka (2017). *Documentation*. [Fecha de consulta: septiembre 2017]. https://kafka.apache.org/documentation/

Atzeni, P; Jensen, C. S.; Orsin, G.; Ram, S. (2013). «The relational model is dead, SQL is dead, and I don?t feel so good myself». *SIGMOD Record* (vol. 42, núm. 2, págs. 64-68).

Bengfort, B.; Kim, J. (2016). *Data Analytics with Hadoop*. Boston: O'Reilly Media.

Canal Vimeo de bases de datos de los EIMT de la UOC [canal en línea]. UOC. https://vimeo.com/channels/basesdedatos/videos

Cattell, R. (2010). *Scalable SQL and NoSQL Data Stores* [documento en línea]. [Fecha de consulta: junio 2017]. http://cattell.net/datastores/Datastores.pdf

Celko's, J. (2013). *Complete Guide to NoSQL*. Boston: Elsevier.

Dunning, T.; Friedman, E. (2015). *Real-World Hadoop*. Boston: O'Reilly Media.

Garofalakis, M.; Gehrke, J.; Rastogi, R. (2016). *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Berlin: Springer.

Golab, L.; Özsu, M. T. (2003). «Issues in data stream management». *ACM Sigmod Record* (vol. 32, núm. 2, págs. 5-14).

Kleppmann, M. (2016). *Making Sense of Stream Processing: The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms*. Boston: O'Reilly Media.

Popescu, A. *NoSQL and Polyglot Persistence* [documento en línea]. [Fecha de consulta: agosto 2017]. http://bit.ly/1ggTT4k.

Redmond, E.; Wilson, J. (2012). *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. The Pragmatic Bookshelf.

Robinson, I.; Webber J.; Eifren, E. (2013). *Graph Databases*. Boston: O'Reilly.

Sadalage, P. J.; Fowler, M. (2013). *NoSQL Distilled. A brief Guide to the Emerging World of Polyglot Persistence, Pearson Education* [documento en línea]. [Fecha de consulta: agosto 2017]. http://bit.ly/1koKhBZ.

Sitto, K.; Presser, M. (2015). *Field guide to Hadoop: an introduction to Hadoop, its ecosystem, and aligned technologies*. Boston: O'Reilly Media.

Stonebraker, M.; Çetintemel, U. (2005). «One Size fits all: An Idea whose time has come and gone». *Proceedings of the International Conference on Data Engineering* (págs. 2-11).

White, Tom (2015). *Hadoop: The Definitive Guide, 4th Edition*. Boston: O'Reilly Media.

Zaharia, M.; Karau, H.; Konwinski, A.; Wendell, P. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*. Boston: O'Reilly Media.