USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

CAMPUS VIDA
CAMPUS DE EXCELENCIA INTERNACIONAL

Gestión de Información no Estructurada

4º curso, 2º cuatrimestre

Andrea Solla Alfonsín

Grao en Enxeñaría Informática

Escola Técnica Superior de Enxeñaría

Universidade de Santiago de Compostela

18 de enero de 2024

Índice general

Tema 1
Bases de datos NoSQL

Las bases de datos relacionales han sido el modelo dominante durante muchos años, y siguen siéndolo hoy en día, ya que proporcionan muchas características importantes para el uso de bases de datos en aplicaciones:

- Filtrado eficiente de datos persistentes
- Consistencia: proporcionan control en el acceso concurrente.
- Permiten la integración de aplicaciones usando una base de datos compartida. Para esto es importante el control de acceso concurrente y de transacciones.
- Estandarización: tanto el modelo relacional como el lenguaje SQL son estándares muy extendidos y conocidos.

La motivación para llegar a las bases de datos NoSQL reside en los siguientes puntos:

- **Impedancia entre los modelos de disco y memoria**: en disco se almacenan tuplas simples (normalmente en 1ª forma normal) mientras que en memoria se utilizan estructuras complejas, con listas o datos anidados. Para resolver esto se han intentado introducir nuevos modelos como las bases de datos orientadas a objetos, que no triunfaron, o el uso de mapeado objeto-relacional. Este último no llega a resolver bien el problema y crea otros, ya que las consultas son del tipo (select from where) y todas las demás operaciones se realizan en la aplicación, como por ejemplo los cálculos, lo cual tiene un rendimiento peor.
- Las BBDD relacionales se utilizan para **integrar aplicaciones con una base de datos centralizada** (bases de datos de integración). Esto hace que la BD sea compleja, y que al realizar cambios en la misma se necesite coordinación entre las aplicaciones, y añade la necesidad de introducir mecanismos de verificación de consistencia. Una alternativa a este modelo es que cada aplicación tenga su propia base de datos (bases de datos de aplicación), realizando la integración con otras aplicaciones a través de una interfaz. De esta forma, cada BD es mantenida por el equipo de la aplicación correspondiente, y se pueden tener modelos diferentes en cada aplicación. Para este tipo de configuraciones pueden introducirse las bases de datos NoSQL, aunque siguen siendo dominantes las relacionales incluso en este modelo.
- Con el incremento de la escala de algunas aplicaciones aparece la **necesidad del uso de clusters**. Las BBDD relacionales no están diseñadas para clusters. Esto se debe a que, aunque pueden usarse en un cluster, en realidad se centraliza el almacenamiento en un subsistema de discos accesible desde los diferentes servidores del mismo, lo cual supone un único punto de fallo. Otra forma de hacerlo es particionando la base de datos, haciendo que cada parte se ejecute en un servidor diferente. Aunque esta alternativa permite distribuir las cargas, la aplicación debe controlar la fragmentación (en qué servidor encontrar cada bit de datos) y, además, se pierde cualquier control de consultas, integridad referencial, consistencia o transacciones entre servidores.

  Como consecuencia de esta incompatibilidad entre BBDD relacionales y clusters, las organizaciones comenzaron a considerar opciones alternativas. Google y Amazon fueron muy influyentes, desarrollando Big Table y Dynamo, respectivamente.

1.1. Aparición de las bases de datos NoSQL

El término NoSQL genera mucha controversia. Su primera aparición se remonta a los 90, dando nombre a una base de datos relacional de código abierto que no usaba SQL como lenguaje de consulta, *Strozzi NoSQL*. Actualmente, no obstante, esta base de datos no tiene ninguna influencia sobre lo que aquí llamamos NoSQL.

Este término se empezó a utilizar a raíz de un *meeting* organizado en San Francisco en 2009, que llevaba ese nombre. En este *meeting* se debían presentar "bases de datos **no relacionales**, **distribuidas** y **de código abierto**", y en él hubo charlas sobre siete BBDD:

- Voldemort
- Cassandra
- Dynomite
- HBase
- Hypertable
- CouchDB
- MongoBD

Aunque se inició con estas siete BBDD, nunca se ha confinado el término únicamente a ellas. No hay una definición generalmente aceptada, ni ninguna autoridad para proporcionarla, lo único que se puede definir es una serie de características comunes de las BBDD que tienden a denominarse como NoSQL. Es importante tener en cuenta que esto son características comunes a BBDD que han sido descirtas como NoSQL, pero no son definitorias del concepto:

- **No usan SQL**: algunas tienen lenguajes de consulta, y muchos son similares a SQL para facilitar su aprendizaje. Sin embargo, hasta ahora ninguna ha implementado nada que pueda encajar con la noción de SQL estándar.
- **No usan esquema**: se pueden añadir campos en cualquier momento sin tener que definirlos primero en la estructura. Esto es muy útil a la hora de trabajar con datos no uniformes y campos personalizados.
- Son generalmente proyectos de código abierto. Aunque a veces se aplica el término NoSQL a systemas de código cerrado, existe la nocion de que NoSQL es un fenómeno de código abierto.
- Muchas bases de datos NoSQL surgen de la necesidad de ejecutarse en clusters, pero no todas están orientadas a esta necesidad. Las bases de datos basadas en grafos son la excepción, ya que su objetivo es representar relaciones, no poder desplegarse en clusters.
- Dado que estas BBDD generalmente se basan en las necesidades de aplicaciones de principios del siglo 21 (web, etc.), normalmente solo reciben el nombre aquellas desarrolladas durante ese periodo de tiempo. Esto descarta las corrientes de BBDD creadas antes del nuevo siglo.

Las dos primeras características son las más importantes, siendo comunes a todas las BBDD NoSQL.

La aparición de este tipo de BBDD abre un nuevo abanico de posibilidades para el almacenamiento de datos, sin eliminar las relacionales de la ecuación (siguen siendo el tipo más utilizado). Se utiliza el concepto de **persistencia políglota** para referirse a este nuevo punto de vista, que se basa en el uso de diferentes tipos de almacenamiento para diferentes necesidades, este es el resultado más importante de este movimiento. Las razones principales

para considerar el uso de NoSQL son un tamaño y rendimiento que hagan necesario el uso de un cluster o la productividad en el desarrollo de aplicaciones, dado que hacen la interacción con los datos más natural, son más cercanas al entorno de la aplicación.

1.2. Modelado

Las bases de datos NoSQL se pueden dividir en cuatro categorías:

- **Clave-Valor (*Key-Value*)**
- ***Column-Family***
- **Documentales**
- **Basadas en grafos**

1.2.1. Modelos agregados

A excepción de las bases de datos orientadas a grafos, las bases de datos NoSQL comparten una característica en su modelo de datos denominada **orientación a agregación**. Esta característica permite trabajar con estructuras más complejas que las tuplas admitidas por el modelo relacional. Aunque este almacenamiento complejo no tiene un término común, le llamaremos agregado.

Un **agregado** es un conjunto de datos con el que interaccionamos como una unidad.

A la hora de trabajar con agregados hay que tener en cuenta la forma de acceso a los datos. No hay una forma universal de estructurar los datos agregados, sino que depende del problema en cuestión. Para poder tener el mejor rendimiento, hay que saber cómo se accederá a los datos para diseñar las estructuras teniendo eso en cuenta. Se puede utilizar la desnormalización para minimizar accesos.

Los agregados facilitan a la BD la gestión del almacenamiento en clusters. Además, las bases de datos orientadas a agregación trabajan mejor cuando la mayor parte de la interacción se realiza con el mismo agregado.

La agregación tiene algunas **consecuencias**. Por un lado, aunque beneficie algunas consultas puede perjudicar otras. Otra consecuencia, al trabajar con un cluster, es que hay que determinar qué datos deben ir juntos para minimizar accesos a varios nodos. Por último, las BBDD NoSQL en general no soportan ACID en transacciones que se expanden por varios agregados.

Modelos clave-valor y documentales

Las bases de datos Clave-Valor y Documentales son fuertemente orientadas a agregación, dado que están construidas principalmente a través de agregados. Ambos tipos consisten en muchos agregados, cada uno con una clave que se usa para obtener los datos.

En las BBDD **Clave-Valor** el agregado es opaco para la base de datos, sin imponer restricciones sobre el contenido del agregado. Además, en estas BBDD solo se puede consultar por clave.

Las **documentales** son capaces de entender la estructura de los agregados, definiendo las estructuras posibles, lo cual añade una limitación pero aporta mayor flexibilidad en el acceso.

Estas BBDD permiten obtener solo una parte del agregado, así como crear índices basados en su contenido.

En la práctica, la línea que separa ambos modelos es un poco borrosa, y existen soluciones que presentan características de ambas. Aún así, se mantiene la distinción general.

Modelo column-family

No se debe confundir este tipo de bases de datos, que no tienen esquema, con el modelo de almacenamiento columnar anterior a NoSQL, que usaba el modelo relacional y SQL, cambiando únicamente la forma de almacenamiento. Las column-family son estructuras tabulares con columnas dispersas y sin esquema, que tiene una estructura agregada en dos niveles. Se puede pensar en ellas como *hashmaps* anidados en dos niveles.

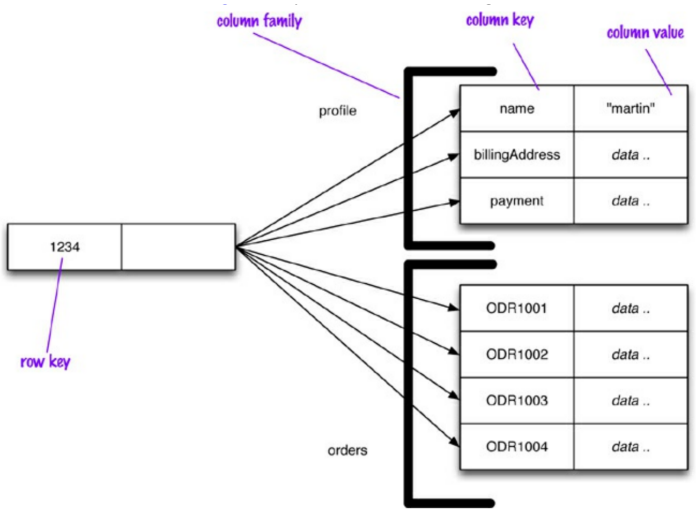

Figura 1.1: Información de cliente en una estructura column-family.

Los datos se estructuran usando cada fila como un agregado y las familias de columnas representan conjuntos de datos útiles de ese agregado. Cada familia de columnas define un tipo de registro, con filas para cada elemento. Estas familias de columnas proporcionan una cualidad bidimensional al modelo.

Este modelo da cierta limitación de estructura al agregado pero permite a la BD aprovechar esa estructura para mejorar la accesibilidad.

1.2.2. Relaciones

En los modelos agregados aparece un problema cuando dos agregados diferentes están relacionados entre si. Una forma de establecer las relaciones es mediante referencias, incluyendo la clave de uno en el otro, pero de esta forma la BD no es consciente de que existe esa relación, lo cual puede ser necesario en muchos casos. Por ese motivo muchas bases de datos incluyen métodos para hacer esas relaciones visibles a la BD.

Un aspecto importante de las relaciones entre agregados es cómo manejar las actualizaciones, dado que las BBDD orientadas a agregación tratan el agregado como unidad de recuperación, por lo que solo pueden asegurar la atomicidad en el contenido de un único agregado. Esto hace que sea complicado trabajar con BBDD orientadas a agregación cuando se necesita operar con muchos agregados, lo cual podría significar que es preferible una BD

relacional para estos casos, pero estas tampoco son muy eficientes cuando se trata de realizar muchos joins. Esta problemática con las relaciones motiva la aparición de las **bases de datos orientadas a grafos**.

Bases de datos orientadas a grafos

Este es el caso especial de las BBDD NoSQL, dado que su motivación no son los agregados ni los clusters, sino el almacenamiento de registros simples con muchas relaciones entre los datos. Este tipo de BBDD tienen aplicaciones en casos como RRSS, preferencias de productos, etc.

El modelo de grafos se basa en **nodos** y **arcos**, pudiendo almacenar datos tanto en unos como en otros. La ventaja que proporciona este modelo es la navegación rápida de las relaciones, aunque la inserción puede ser lenta.

Este tipo de bases de datos no suelen ser utilizados en clusters, sino en un único servidor. Son muy diferentes de los otros tres tipos, teniendo en común únicamente el rechazo por el modelo relacional y la ausencia de esquema.

1.2.3. Esquema

La característica común a todas las bases de datos NoSQL es que no obligan a definir un esquema, aunque algunas tienen una estructura más definida que otras. El orden de más a menos esquema sería el siguiente: clave-valor, documental, column-family, grafos.

Trabajar sin esquema tiene múltiples **ventajas**:

- No es necesario hacer asunciones previas.
- Incorporar cambios en los datos es fácil. Si un campo no se necesita seguir utilizando simplemente se deja de insertar, pero no se pierde la información anterior (en el modelo relacional habría que borrar la columna entera).
- Facilidad para trabajar con datos no uniformes.
- Flexibilidad en la inserción de datos. Si hay campos que tienen algunos datos y otros no, se usan solo cuando sea necesario, mientras que con esquema esto llevaría a tener columnas muy dispersas (con muchos valores null).

No obstante, también tiene **inconvenientes**. Aunque se trabaje sin esquema, las aplicaciones necesitan un cierto formato y semántica para nombres de columnas, tipos de datos, etc. Al trabajar sin esquema a nivel de BD, el esquema se codifica en las aplicaciones, lo cual puede dar problemas al ser necesario analizar el código para entender los datos (esto puede llegar a ser muy complejo, depende mucho de la claridad del código). Además, no se pueden implementar restricciones de integridad en la base de datos.

Una base de datos sin esquema desplaza el problema a la aplicación, lo cual puede ser peor si se usan aplicaciones distintas realizadas por personas distintas. Por esto, es mejor en estos casos que el acceso y gestión de la base de datos se encapsule en una única aplicación, a través de la cual las demás interactúen con la BD. También se puede intentar limitar diferentes partes de cada agregado para el acceso a aplicaciones diferentes.

Es importante tener en cuenta que **los esquemas tienen valor**, y su rechazo puede llegar a ser un problema.

1.3. Distribución

Hay dos formas diferentes de distribuir los datos entre nodos, que se pueden combinar:

- **Replicación**: tener los mismos datos copiados en diferentes nodos.
- **Particionamiento (*sharding*)**: tener diferentes datos en diferentes nodos.

Se pueden distinguir cuatro técnicas diferentes (ordenadas de menor a mayor complejidad):

- Servidor único: no hay distribución, evita toda la complejidad que esta conlleva, lo cual la lleva a ser preferible si no hay necesidad de distribuir los datos. El uso de NoSQL se justifica en estos casos por cuestiones relacionadas con el modelo de datos.
- Replicación maestro-esclavo.
- Particionamiento.
- Replicación peer-to-peer.

1.3.1. Particionamiento (*sharding*)

El *sharding* proporciona escalabilidad horizontal. Los datos se distribuyen entre los diferentes nodos, de forma que cada uno hace escrituras y lecturas sobre sus datos. El caso ideal sería que cada usuario acceda a un nodo distinto, pero para esto es necesario que cada usuario acceda a datos distintos y que esos datos se repartan bien entre los nodos, lo cual es bastante difícil.

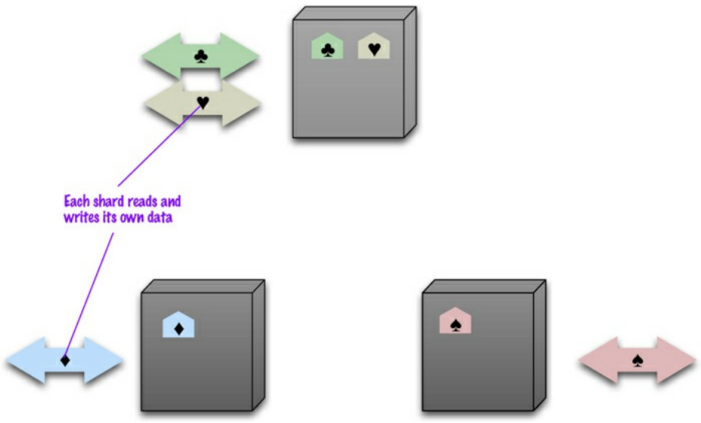

Figura 1.2: Particionamiento.

Se busca con esta técnica aprovechar al máximo la localidad de los datos, intentando que aquellos que se accedan juntos estén almacenados en el mismo nodo. Para conseguir esto se debe intentar colocar los datos que se acceden juntos dentro del mismo agregado, utilizando el agregado como unidad de datos para la distribución. Además, si se conoce el orden más típico de acceso, se pueden mantener juntos agregados que suelen accederse juntos.

También es importante tener los datos lo más cerca posible de donde son utilizados con más frecuencia y mantener todos los nodos con una carga de datos similar (distribución uniforme de los datos). Para esto último es importante decidir bien la clave de particionamiento.

Si la base de datos que se use no implementa el particionamiento, hay que implementarlo a nivel de aplicación. Esto hace que las aplicaciones tengan un código mucho más complejo y que haya que cambiar la aplicación cada vez que se reorganizan los datos. Muchas BBDD NoSQL asumen la responsabilidad del particionamiento.

Rendimiento

El particionamiento es mejor que la replicación para el rendimiento, ya que mejora tanto lecturas como escrituras (se puede leer o escribir en los diferentes nodos simultáneamente).

Fiabilidad

Esta técnica no es buena para la disponibilidad, ya que no la mejora y puede haber fallos parciales. Además, al aumentar el número de servidores aumenta la probabilidad de fallo.

1.3.2. Replicación maestro-esclavo

Con la distribución maestro-esclavo se replican los datos en varios nodos. Uno de los nodos se designa como **maestro** o **primario**, lo cual lo convierte en la autoridad como fuente de datos y el responsable de su actualización. Los demás nodos son **esclavos** o **secundarios**. Los nodos secundarios se sincronizan con el primario mediante el **proceso de replicación**, que es habitualmente asíncrono (si no, puede empeorar la fiabilidad).

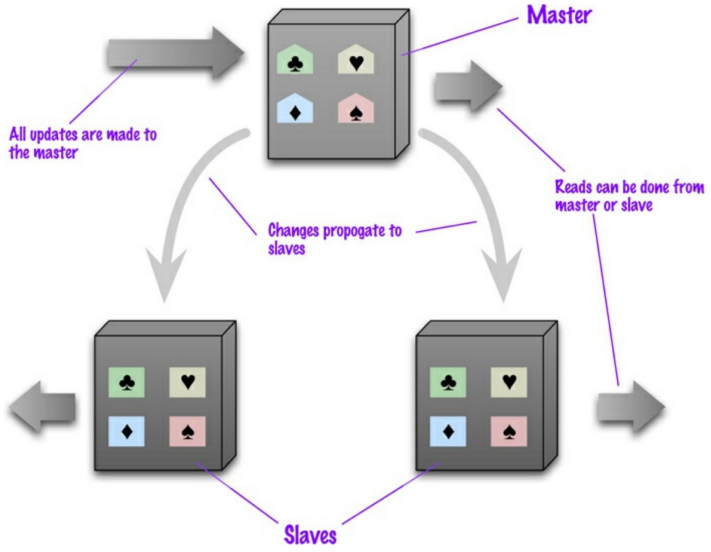

Figura 1.3: Replicación maestro-esclavo.

Esta técnica es una buena solución cuando la aplicación es intensiva en lecturas, gracias a sus dos principales **ventajas**:

- Alta disponibilidad para lecturas: aunque falle el maestro, los esclavos pueden seguir atendiendo las peticiones de lectura.
- Sustitución del maestro ante un fallo: al tener copias del maestro, se puede hacer una replicación en caliente o designar a uno de los esclavos como nuevo maestro. Esto puede ser útil incluso si no hay necesidad de distribuir las lecturas, centralizando el tráfico de lecturas y escrituras en el maestro y usando el esclavo como un backup en caliente.

La selección de un maestro puede ser **manual** (al configurar el cluster se designa un nodo como maestro) o **automática** (el maestro lo eligen los propios nodos). La selección automática reduce el tiempo de recuperación cuando el maestro falla, al elegir el cluster un nuevo maestro directamente.

Aunque esta técnica tiene algunos beneficios, también presenta **inconvenientes**:

- Cuello de botella: todas las modificaciones se deben hacer a través del maestro, lo cual presenta un cuello de botella. Esto limita al sistema a la habilidad del maestro para procesar las actualizaciones y transmitirlas. Como consecuencia, no es un buen esquema para datasets con gran tráfico de escritura (aunque la posibilidad de liberar al maestro de las lecturas ayuda en cierta medida).
- Consistencia: lecturas en esclavos distintos pueden proporcionar resultados distintos, ya que las actualizaciones pueden no haber terminado de propagarse. Esto puede dar lugar a que un cliente no pueda ver los datos que acaba de escribir.

  Incluso solo usando esta técnica como backup en caliente este problema puede darse, de forma que al fallar el maestro el esclavo aun no tuviera las últimas modificaciones y, por tanto, se pierdan datos.

Esta técnica, con respecto a la replicación peer-to-peer, facilita la consistencia (todos los cambios se realizan en el mismo nodo) pero penaliza disponibilidad.

1.3.3. Replicación *peer-to-peer*

La arquitectura maestro esclavo no proporciona escalabilidad para escrituras, teniendo el maestro como cuello de botella. Además, es resistente al fallo de un esclavo pero no al de un maestro, por lo que proporciona un único punto de fallo.

La replicación *peer-to-peer* enfrenta estos problemas eliminando la figura del maestro, de forma que todas las réplicas tienen el mismo peso. Así, todos los nodos aceptan escrituras y la pérdida de uno de ellos no impide el acceso a los datos.

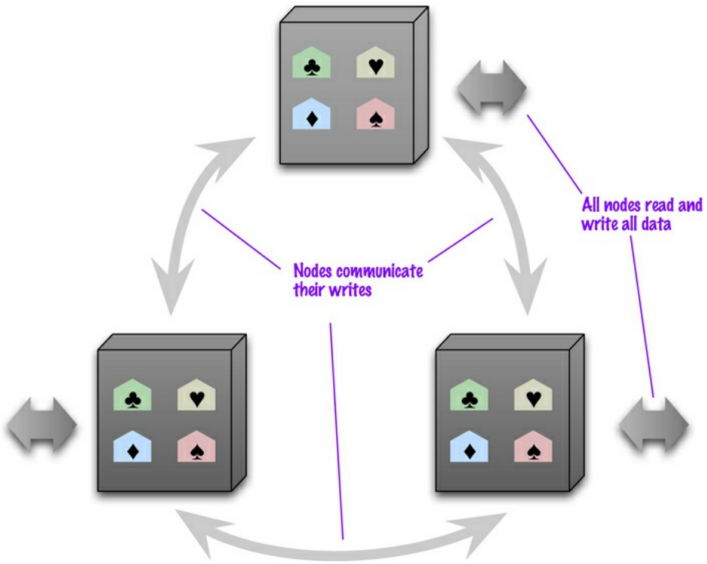

Figura 1.4: Replicación *peer-to-peer*.

A pesar de parecer una buena solución, también presenta una complicación importante: la **consistencia**. Al tener la posibilidad de escribir en dos sitios diferentes, se corre el riesgo de que se intente modificar el mismo dato a la vez en dos nodos distintos, un conflicto *write-write*. A diferencia de los conflictos *read-write*, que crean problemas transitorios, los *write-write* crean **problemas que perduran**.

Las soluciones generales a este problema de consistencia son la coordinación de las réplicas durante las escrituras o asumir las inconsistencias e intentar arreglarlas combinando répli-

cas. La primera solución genera un coste de red adicional, pero es suficiente con actualizar la mayoría de nodos de forma coordinada. La segunda solución permite obtener el mayor rendimiento de la posibilidad de escribir en cualquier réplica.

En conclusión, hay que tomar ciertas decisiones para llegar a un compromiso entre consistencia y disponibilidad.

1.3.4. Combinación de particionamiento y replicación

Ambas técnicas se pueden combinar, pero varían un poco en función del modelo de replicación escogido:

- Con maestro-esclavo: cada partición tiene un único maestro. Se pueden elegir maestro y esclavos a nivel de cluster o para cada partición, dependiendo de la aplicación.
- Con *peer-to-peer*: es muy común en soluciones de tipo *column-family*. Se suele usar replicación con factor 3, de forma que cada *shard* tiene tres copias en tres nodos diferentes y, si uno de ellos falla, se puede recuperar con las demás copias.

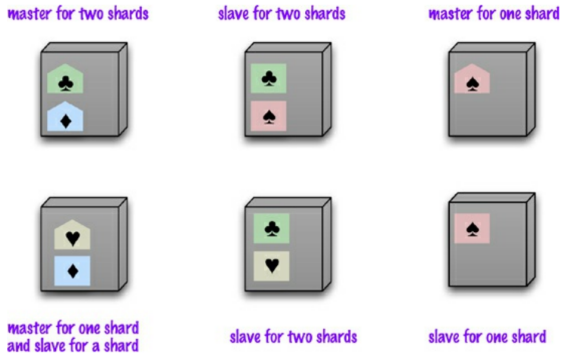

Figura 1.5: Combinación de particionamiento con maestro-esclavo.

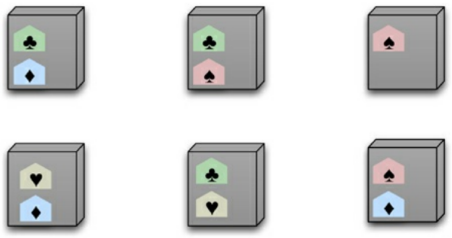

Figura 1.6: Combinación de particionamiento con *peer-to-peer*.

1.3.5. *Map-Reduce*

En una arquitectura centralizada, el procesamiento puede realizarse en el cliente (mayor flexibilidad pero hay que mover los datos) o en el sercidor (menos cómodo pero más eficiente). En cambio, en una arquitectura distribuida hay muchas máquinas entre las que repartir la carga, siendo importante minimizar el tráfico de datos entre nodos. Surge así la necesidad de encontrar una forma de aumentar el paralelismo sin hacer que el intercambio de datos sea elevado.

El patrón **Map-Reduce** viene de la programación funcional (define funciones). Es una forma de programar el procesamiento para minimizar el tráfico entre nodos. El funcionamiento

básico de este patrón se divide en dos operaciones:

- **Map**: para cada agregado genera un conjunto de pares clave-valor.
- **Reduce**: sobre el resultado del Map, agrega valores de elementos con la misma clave. Es decir, para cada clave aplica una función que reduce todos los valores.

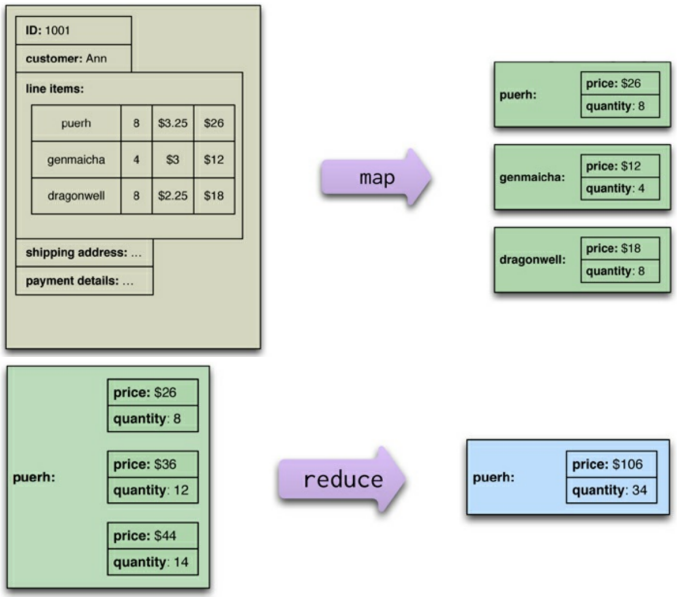

Figura 1.7: Funcionamiento básico de Map-Reduce.

Paralelización

Dado que las operaciones Map en cada agregado son independientes, se pueden paralelizar fácilmente. Por otro lado, la operación Reduce opera uniendo todas las claves iguales, por lo que para paralelizarla es necesario particionar la salida del Map por la clave. Esta operación de particionado se conoce como ***Shuffling***.

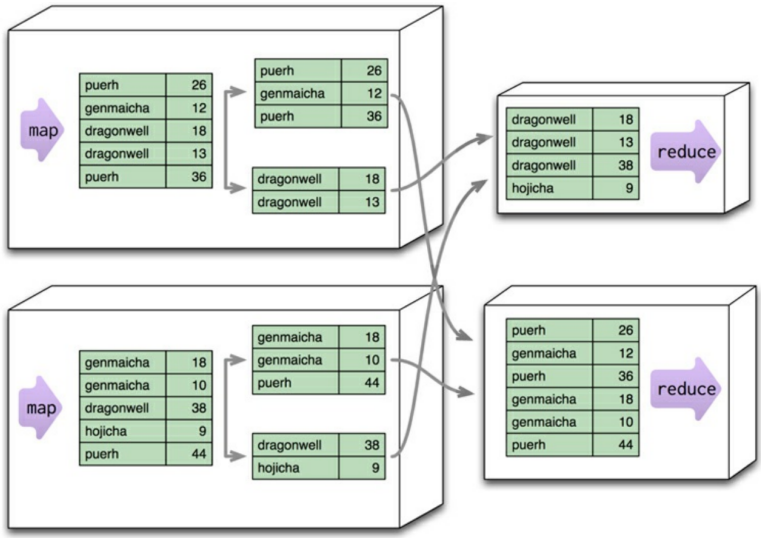

Figura 1.8: Paralelización de Map-Reduce.

Entre las operaciones de Map y Reduce se mueve una cantidad elevada de datos entre

los nodos, muchos de ellos repetitivos (varios pares clave-valor con la misma clave). Para minimizar ese movimiento se puede introducir una función de combinación, ***Combine***, a continuación del Map en cada nodo. Esta función es como la función Reduce, pero tiene una limitación: no todas las funciones son combinables. Si, por ejemplo, solo se quiere sumar la cantidad de veces que aparece cada palabra, se pueden hacer sumas parciales en cada nodo con Combine y luego calcular el resultado final con Reduce. Pero si en lugar de eso se quiere hacer, por ejemplo, una media, no se puede usar este cálculo intermedio. Para que un reducer sea combinable su salida debe tener la misma forma que la entrada.

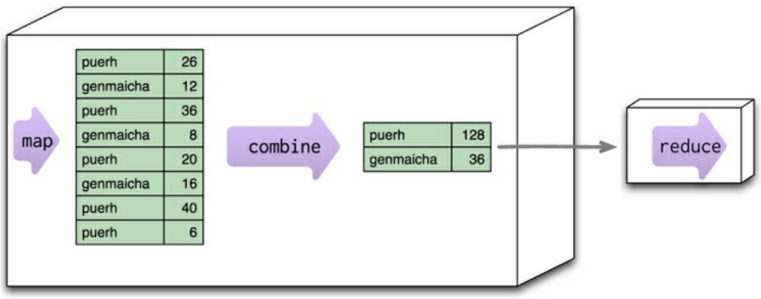

Figura 1.9: Introducción de la función Combine.

Cálculos

A la hora de realizar cálculos, hay que tener en cuenta como funcionan las operaciones, que tienen sus propias restricciones:

- Map solo opera sobre un agregado.
- Reduce solo opera sobre una clave.

A veces es necesario calcular varios campos para poder hacer el cálculo final en el Reduce. Por ejemplo en el caso de la media, hay que calcular el total y el recuento, para poder dividirlos al final y obtener la media (no se puede hacer la media de medias parciales). En otros casos más complejos es necesario descomponer el cálculo en varias etapas Map-Reduce.

Hay lenguajes especializados en este tipo de programación, como Pig o Hive. Map-Reduce es un paradigma importante al trabajar con archivos en sistemas distribuidos (fuera de NoSQL).

Incremental

A veces los datos están llegando en un flujo continuo, por lo que hay que ir modificando los resultados. La operación Map es fácil de ejecutar incrementalmente, ya que cada tarea Map es independiente de las demás. Sin embargo, las tareas Reduce dependen de muchos resultados de diferentes Map, aquellas que no hayan cambiado pueden evitar recalcularse y, si es un reducer combinable y los cambios son aditivos, solo es necesario aplicar el reduce sobre los cambios. También se pueden usar redes de dependencias para recalcular solo las partes necesarias.

1.4. Consistencia

Las bases de datos relacionales proporcionan alta consistencia, ACID (atomicidad, consistencia, aislamiento y durabilidad). No obstante, la consistencia es lo único que la BD no

puede asegurar, dado que depende del usuario.

Por otro lado, las bases de datos NoSQL tienen **consistencia eventual**, esto quiere decir que puede haber momentos en los que no sea consistente pero que en algún momento llegará a serlo. Se aplica a estas BBDD el **teorema CAP**.

1.4.1. Modificaciones

Cuando dos usuarios intentan modificar el mismo dato en paralelo, se da un **conflicto *write-write***, un problema de modificación perdida (el primer cambio que se aplique será inmediatamente sobreescrito por el otro). Para solucionar este tipo de conflictos, se pueden aplicar estrategias optimistas o pesimistas.

Estrategias pesimistas

Realizan acciones para evitar que se produzcan inconsistencias. Pueden ser protocolos basados en bloqueos y marcas de tiempo.

Esta estrategia hace los sistemas más lentos, dado que están actuando siempre, independientemente de que fuera a haber un conflicto o no. Normalmente los usuarios suelen preferir estas estrategias *a priori* para evitar problemas, pero no siempre es la mejor solución ya que suelen ser poco eficientes y pueden producir interbloqueos.

Estrategias optimistas

Permiten la ejecución normal de las transacciones y actúan en el momento de compromiso sólo si se detectan problemas. Pueden ser protocolos basados en validación y versiones.

Un ejemplo de estrategia optimista es una actualización condicional, en la que cuando un cliente realiza una actualización comprueba el valor justo antes, para comprobar si ha cambiado desde la última lectura. Otro ejemplo sería guardar las dos actualizaciones que entren en conflicto y registrar que hay conflicto (control de versiones) finalmente, en algún momento se realiza un *merge* (automático o preguntando al usuario).

La parte negativa de esta estrategia es que, si hay problemas, hay que deshacerlo todo. En la actualidad se opta más por este tipo de estrategias.

Consistencia secuencial

Tanto las estrategias optimistas como las pesimistas se basan en una serialización consistente de las actualizaciones. Esto se refiere a que, si hay varios servidores, pueden aplicar las actualizaciones en diferente orden, obteniendo un resultado diferente en cada nodo.

Usualmente, cuando se habla de concurrencia en sistemas distribuidos, se habla de consistencia secuencial. Esto se refiere a asegurar que todos los nodos aplican las operaciones en el mismo orden.

Compromiso entre consistencia y eficiencia

La mayoría de modelos de distribución usan solo una copia para modificar (maestro-esclavo). Esto simplifica las soluciones que eviten conflictos write-write.

1.4.2. Lecturas

Tener un almacén de datos que mantenga la consistencia de actualización no garantiza que las lecturas sean siempre consistentes. Además, se pueden leer datos entre dos escrituras, lo cual puede suponer una pérdida de datos en la lectura. Asegurar que diferentes elementos de datos tienen sentido juntos (por ejemplo la suma del dinero de dos cuentas entre las que se ha movido dinero[1]) se denomina **consistencia lógica**. En las BBDD relacionales hay soporte para transacciones, que resuelve este problema, y es común decir que las bases de datos NoSQL no tienen este tipo de mecanismos, pero eso se refiere más bien a BBDD de agregados.

Las BBDD de grafos suelen proporcionar soporte ACID, mientras que las que trabajan con agregados soportan modificaciones atómicas dentro de un mismo agregado. Para las transacciones con varios agregados, se denomina **ventana de inconsistencia** al tiempo entre el cual se pueden producir lecturas inconsistentes, que suele ser muy corto (menos de 1 segundo).

La replicación introduce nuevos problemas de inconsistencias. Esto se debe a que, si se actualiza un dato, desde algunos nodos puede verse actualizado mientras que desde otros se sigue viendo sin actualizar. Se llama **consistencia de replicación** a asegurar que un mismo dato se lee igual desde todas las réplicas. Por otro lado, la **consistencia eventual** se refiere a ese tipo de situaciones, significa que puede haber réplicas inconsistentes pero en algún momento la actualización llegará a todos los nodos.

Aunque la consistencia de replicación es independiente de la consistencia lógica, la replicación puede aumentar una inconsistencia lógica alargando su ventana de inconsistencia. Las garantías de consistencia no son algo global en una aplicación, normalmente se puede especificar el nivel de consistencia que se quiere en cada operación individualmente. Esto permite tener un nivel bajo de consistencia en general, cuando no es un problema, pero pedir alta consistencia cuando sí lo es.

El caso de que dos personas lean estados diferentes desde lugares diferentes es algo problemático, ya que normalmente los usuarios actúan independientemente. Las ventanas de consistencia pueden ser particularmente problemáticas cuando un usuario obtiene inconsistencias consigo mismo (no poder ver algo que acaba de escribir, porque la lectura y la escritura se hacen en nodos diferentes). Esto se llama **consistencia *read-your-writes***, significa que, una vez que haces una modificación, tienes garantizado continuar viendo esa actualización. Una forma de conseguir este tipo de consistencia es proporcionando **consistencia de sesión**: durante una sesión de usuario hay consistencia *read-your-writes*. Algunas formas de implementar la consistencia de sesión son mediante *sticky session* o con marcas de versiones:

- ***Sticky session***: la sesión se vincula a un nodo (también se conoce como afinidad de sesión). Tiene como inconveniente que reduce la capacidad del balanceador de carga. Si se utiliza maestro-esclavo, esta técnica tiene otro problema: se puede leer de los esclavos, pero siempre se modifica en el maestro. Para solucionar esto, se puede modificar en el esclavo y hacerlo responsable de actualizar el maestro o mover la sesión al maestro mientras se propaga la modificación (una vez hecha la sesión vuelve al esclavo).
- **Marcas de versiones**: se trata se asegurar que toda interacción con los datos incluye la última versión vista por la sesión. De esta forma, el nodo que responda a la petición debe asegurarse de que tiene las actualizaciones que incluyen la marca de versión antes de responder.

[1] Si se escribe la retirada de dinero de la cuenta A y se lee antes de que se haga la inserción en la cuenta B, en la lectura va a faltar la cantidad de dinero que se estuviera traspasando.

Por último, hay que tener en cuenta que estamos hablando de consistencia en el almacén de datos, no en la aplicación. Es importante tener cuidado de no mantener transacciones abiertas durante la interacción con el usuario, dividiéndolas en dos partes: primero se obtiene la información y se interactúa con el usuario y, después, se vuelve a comprobar la información (para ver que no haya cambiado) y se realiza la transacción.

1.4.3. Relajar consistencia

Aunque la consistencia es importante, a veces hay que sacrificarla. Es posible diseñar un sistema que evite inconsistencias, pero es imposible hacerlo sin perder otras características del sistema. En conclusión, muchas veces hay que cambiar consistencia por mejorar otras propiedades. Cuanto más se conozca la aplicación, mejor se podrá encontrar el equilibrio.

En los sistemas centralizados también se reduce la consistencia en ciertas ocasiones. Por ejemplo, en SQL hay varios niveles de consistencia:

- **Secuenciable**: ejecución secuencial (casi siempre).
- **Lectura repetible**: lectura de datos comprometidos y lectura repetible.
- **Lectura comprometida**: lectura de datos comprometidos.
- **Lectura no comprometida**: lectura de datos no comprometidos.

Muchos sistemas evitan el uso de transacciones completamente. Por ejemplo, MySQL era muy popular cuando no tenía gestión de transacciones, dado que en muchas aplicaciones web se cargan los datos al principio y luego solo se realizan lecturas, y la ausencia de transacciones proporciona mayor velocidad. Otros casos, como los de sitios web grandes que necesitan usar *sharding* o si se produce interacción con sistemas remotos, necesitan eliminar las transacciones para tener una eficiencia aceptable.

Teorema CAP

Fue formulado por Eric Brewer en 2000 y pobado formalmente por Seth Gilbert y Nancy Lynch en 2002. En el ámbito de BBDD NoSQL se suele hablar de este teorema como la razón para relajar la consistencia.

El teorema CAP plantea que un sistema solo puede tener dos de las tres siguientes características:

- **Consistencia**.
- **Disponibilidad**: si podemos comunicarnos con un nodo del cluster, entonces podemos leer y escribir datos (significado particular en este teorema).
- **Tolerancia a particionamiento**: el cluster puede sobrevivir a fallos de comunicación que lo separan en varias partes.

Un **sistema centralizado** tiene CA. Un servidor único no se puede particionar, por lo que no tiene tolerancia a particionado. Por otro lado, al ser único es consistente y si está levantado está disponible.

Un **cluster** puede ser CA, pero esto significa que, si se particiona en algún momento, debe pararse por completo. En la definición habitual de disponibilidad, esto implicaría una falta de la misma pero, como en este contexto la definición cambia, al no recibir peticiones los nodos (están parados) no tienen que responder. Esto es posible, pero muy costoso (al tener que parar todos los nodos) y detectar las particiones a tiempo también supone un problema.

En realidad, los clusters deben tolerar particionamientos en la red. Y en este punto surge la aplicación real del teorema CAP: aunque en teoría es "solo puedes tener dos de las tres características", en la práctica quiere decir que un sistema que puede sufrir particiones, como ocurre con los sistemas distribuidos, tiene que hallar un compromiso entre consistencia y disponibilidad (mejorar una empeora la otra).

La conclusión es que, aunque normalmente se busca consistencia, se puede relajar para conseguir disponibilidad y mejorar la eficiencia. Para poder llegar al mejor compromiso se necesita conocer el dominio de la aplicación, ya que en algunos casos es más importante la consistencia y, en otros, el rendimiento.

Se suele decir que los sistemas NoSQL siguen, en lugar de las propiedades ACID, las **popiedades BASE**. Estas propiedades son ***Basically Available***, ***Soft state*** y ***Eventual consistency***, aunque no tienen una definición clara. Normalmente se suelen elegir soluciones intermedias, en lugar de verlo como una elección binaria.

A veces, en lugar de compromiso entre consistencia y disponibilidad, es mejor (más claro) hablar de compromiso entre consistencia y latencia. Podemos pensar en la disponibilidad como el límite de latencia que podemos tolerar, cuando esta es demasiado alta se considera que el dato no está disponible. De esta forma, cuanta mayor consistencia (incluyendo más nodos en la interacción para mejorarla) mayor latencia (más lento).

1.4.4. Relajar durabilidad

Hasta ahora hemos hablado de consistencia, que es a lo que se suele referir la gente cuando habla de propiedades ACID. La clave de la consistencia es serializar peticiones formando unidades de trabajo atómicas y aisladas. En cuanto a la durabilidad, puede no parecer útil (qué objetivo tiene un almacén de datos si pierde actualizaciones?). Sin embargo, hay casos en los que se prefiere cambiar algo de durabilidad por mejor rendimiento.

Si una base de datos puede correr principalmente en memoria, aplicar las actualizaciones a su representación en memoria y periódicamente volcar los cambios al disco, puede proporcionar una mayor capacidad de respuesta a peticiones. El coste de este método es que, si el servidor cae, cualquier actualización desde el último volcado se perdería. Además, normalmente se puede especificar las necesidades de durabilidad en cada escritura, de forma que las actualizaciones más importantes pueden forzar un volcado a disco.

En algunos casos compensa el cambio, por ejemplo, el almacenamiento del estado de una sesión de usuario. Un gran sitio web puede tener muchos usuarios y mantener información temporal de lo que hace cada usuario, pero al tener muchos usuarios a la vez y estar registrando su actividad habrá mucha actividad en el estado de la sesión, lo cual afecta a la capacidad de respuesta de la página. En este tipo de casos, perder los datos de sesión no es un gran problema, por lo que compensa conseguir algo más de margen para el rendimiento.

Otro caso relacionado con esto es la **durabilidad de replicación**. Un posible problema de este tipo sería que un maestro falle antes de escribir las últimas actualizaciones en algún esclavo, se designa un nuevo maestro y se siguen escribiendo actualizaciones, cuando el maestro que había fallado vuelva a estar disponible, pueden aparecer conflictos. Para evitar este problema se puede esperar a que los esclavos estén actualizados para comprometer la transacción, pero esto empeora el rendimiento de las escrituras y disminuye la disponibilidad (aumenta la probabilidad de fallo de escritura).

1.4.5. Quorums

Cuando se cambia consistencia o durabilidad por eficiencia, no es un cambio de todo o nada. Cuantos más nodos están involucrados en una petición, mayor será la probabilidad de evitar una inconsistencia. Esto lleva a cuestionar cuantos nodos deben involucrarse para una consistencia fuerte.

Se denomina **quórum de escritura** al número de nodos necesarios para confirmar una escritura (la mayoría, osea, más de la mitad). Se expresa como $W > N/2$, donde W son los nodos que participan en la escritura y N el número de nodos involucrados en la replicación, conocido como **factor de replicación**.

El **quórum de lectura** es el número de nodos que hay que consultar para asegurar que se tiene la copia más reciente (R). De esta forma, se puede tener una lectura fuertemente consistente si $R + W > N$.

En ambos casos nos referimos a un modelo de distribución *peer-to-peer*. Con un maestro-esclavo, llega con leer y escribir en el maestro para evitar conflictos *write-write* y *read-write*.

No se debe confundir el factor de replicación con el número de nodos del cluster. Un cluster puede tener 12 nodos y un factor de replicación 3 (3 copias de cada dato).

El factor de replicación 3 es muy común, ya que permite un fallo de un nodo mientras se realiza una nueva réplica. En general, depende de la aplicación ajustar W y R para que el resultado sea acorde a sus necesidades. Para tener lecturas consistentes y rápidas W tiene que ser alto para poder tener R bajo, por lo que las escrituras serán lentas. Por otro lado, si importa menos la consistencia pero se busca escritura rápida, se puede reducir W aumentando R. Como se puede ver, no se trata de un simple compromiso entre disponibilidad y consistencia, sino que es algo más complejo.

1.4.6. Versiones

Aunque algunos críticos señalan la falta de soporte para transacciones en las bases de datos NoSQL, muchas bases de datos NoSQL orientadas a agregados sí admiten actualizaciones atómicas dentro de un agregado, lo cual proporciona cierto nivel de consistencia.

Sin embargo, es importante considerar las necesidades transaccionales al elegir una base de datos. Incluso en sistemas transaccionales, existen actualizaciones que requieren intervención humana y no pueden ejecutarse dentro de transacciones debido a su duración. Para abordar esto, se utilizan las marcas de versión, que resultan útiles en situaciones más allá del modelo de distribución de un solo servidor.

Transacciones de negocio y sistema

La consistencia de las actualizaciones sin transacciones es una necesidad común en los sistemas, incluso en aquellos basados en bases de datos transaccionales. Las transacciones empresariales, como completar una compra en línea, generalmente no ocurren dentro de una transacción de base de datos, ya que bloquearía los elementos de la base de datos durante todo el proceso. Esto puede resultar en cálculos y decisiones basados en datos que han cambiado, como precios actualizados o información del cliente modificada.

Una técnica útil para abordar este problema es el **Bloqueo Optimista sin Conexión**, que implica que una operación del cliente vuelva a leer los datos en los que se basa la transacción y verifique si han cambiado desde la lectura original. Una forma de implementar esto es

mediante el uso de marcas de versión, que son campos en los registros de la base de datos que cambian cada vez que los datos subyacentes cambian. Al leer los datos, se registra la marca de versión, lo que permite verificar si ha habido cambios al escribir los datos. Esto permite manejar situaciones en las que los cálculos y decisiones se basan en datos que pueden cambiar durante la interacción con el usuario.

Formas de implementarlas

Las marcas de versión ayudan a mantener la consistencia y son especialmente útiles en situaciones en las que los cálculos y decisiones se basan en datos que pueden cambiar. Hay varias formas de implementar las marcas de versión, como el uso de **contadores**, **GUID** (número aleatorio único), **hashes** de contenido o **marcas de tiempo**. Cada enfoque tiene sus ventajas y desventajas en términos de comparación de versiones, generación de valores únicos y eficiencia.

Marca de versión vectorial

Las marcas de versión funcionan bien en un nodo o con replicación maestro-esclavo, ya que un solo nodo controla las versiones. Sin embargo, en los sistemas *peer-to-peer* dos nodos pueden tener valores distintos porque a uno de ellos aún no llegó la modificación o porque hay algún tipo de inconsistencia en la modificación.

En sistemas distribuidos *peer-to-peer*, se suele utilizar una forma especial de marca de versión conocida como marca de versión vectorial. Consiste en un conjunto de contadores, uno por cada nodo en el sistema. Cuando los nodos se comunican, sincronizan sus marcas de versión vectoriales, lo que les permite detectar conflictos de actualizaciones en diferentes nodos.

En esencia, una marca de versión vectorial es un conjunto de contadores, uno para cada nodo. Una marca de versión vectorial para tres nodos (azul, verde, negro) se vería así: [azul: 43, verde: 54, negro: 12]. Cada vez que un nodo tiene una actualización interna, actualiza su propio contador, por lo que una actualización en el nodo verde cambiaría el vector a [azul: 43, verde: 55, negro: 12]. Cuando dos nodos se comunican, sincronizan sus marcas de versión vectoriales.

Al utilizar este esquema, se puede determinar si una marca de versión es más nueva que otra porque la marca más reciente tendrá todos sus contadores mayores o iguales que los de la marca más antigua. Por lo tanto, [azul: 1, verde: 2, negro: 5] es más reciente que [azul: 1, verde: 1, negro: 5] porque uno de sus contadores es mayor. Si ambas marcas tienen un contador mayor que el otro, por ejemplo, [azul: 1, verde: 2, negro: 5] y [azul: 2, verde: 1, negro: 5], entonces hay un conflicto de escritura-escritura.

Conclusión

Las marcas de versión son herramientas valiosas para detectar conflictos de concurrencia, pero no resuelven automáticamente estos conflictos. La resolución de conflictos depende del dominio y del equilibrio entre consistencia y latencia que se desee lograr.

En resumen, las marcas de versión son utilizadas para mantener la consistencia y detectar conflictos en las bases de datos NoSQL. Proporcionan una forma de verificar la integridad de los datos y son especialmente útiles en sistemas distribuidos donde múltiples nodos pueden realizar actualizaciones concurrentes.

Bibliografía

[1] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. 1ª edición. New York: Cambridge University Press, 2008.

[2] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[3] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. 6th Edition. McGraw-Hill, 2014.