Dominio de las BD relacionales debido a

- Filtrado eficiente de los datos persistentes.
- Control de acceso concurrente. Garantiza consistencia con transacciones ACID.
- Buena integración con aplicaciones.
- Su estadariación: basadas en el modulo relacional y usan SQL como lenguaje. Esto permite no tener que recodificar aplicaciones al cambiar la BD.

Modelando para el acceso a datos

Debemos tener en cuenta cómo va a ser el acceso a datos. Como alternativa podemos dividir el agregado y usar referencias (en BD documentales no es necesario almacenar las referencias en ambas direcciones. Debemos calcular el agregado para hacer consultas analícas (excepto en documentales, pero no es eficiente).

Problemas con las BD relacionales

Impedancia entre modelos de memoria y disco

Los datos en disco se almacenan como tuplas simples (1NF), mientras que en memoria se manejan estructuras complejas (anidadas). Como soluciones se plantean:

- BD orientadas a objetos, pero no llegaron a triunfar.
- El mapeado objeto-relacional no resuleve el problema del todo, pero induce nuevos problemas, como la ineficiencia al evitar el uso del SGBD.

Integración de aplicaciones a través de una BD centralizada

Este tipo de BD es compleja y requiere de mecanismos de verificación de consistencia. Además, los cambios en la BD requieren coordinación entre las aplicaciones. Como alternativa, se plantea que cada aplicación tenga su propia BD (incluso así, el modelo relacional domina). Esto permite:

- El equipo de cada app conoce, mantiene y evoluciona su BD.
- La verificación de integridad se realiza en cada aplicación.
- Integración a través de interfaces como servicios web, con uso de XML o JSON.
- Permite usar distintos modelos de BD, dependiendo de las necesidades de cada app.

Necesidad el uso de cluster

Surge del incremento en la escala de las aplicaciones modernas. Como soluciones se plantean:

- Escalado vertical con máquina más potentes (limitado y costoso).
- Escalado horizontal con *cluster* de máquinas más pequeñas. Esto da mayor resiliencia y abarata costes.

Surgen problemas ya que las BD relacionales no están diseñadas para arquitecturas distribuidas. Implementar entonces una BD sobre sistemas de archivos induce un único punto de fallo, y el almacenamiento fragmentado requiere que la aplicación controle la distribución de datos. Aquí surgen alternativas a tener en cuenta como Big Table de Google o Dynamo de Amazon.

NoSQL

No hay una definición clara del término. Inicialmente era “BD no relaciones, distribuidas y de código abierto”.

Aparición

- No usan SQL (aunque algunos lenguajes sean muy parecidos).
- Funcionan sobre *clusters*: importan sobre el modelo de datos y sobre la solución al problema de consistencia entre ACID y *cluster* (excepto las de grafos).
- Se adaptan a las nuevas necesidades y operan sin esquema, aunque se suelen usar como BD de aplicaciones, no de integración.
- Abre opciones para el almacenamiento de datos, sin restringir la incorporación de nuevos datos.
- Hay que considerar NoSQL cuando el tamaño y rendimiento hagan necesarios el uso de un *cluster* y por temas de productividad, ya que dan una interacción más natural con los datos.

Agregados

La gran diferencia de NoSQL con las BD relacionales es el uso de agregados, ya que el relacional no permite anidar estructuras. Se diseñan las estructuras pensando en cómo va a accederse a ellas: se agrupan y replican los datos en un único documento, minimizando los accesos (desnormalización). Como consecuencia, la agregación puede beneficiar unas consultas y perjudicar otras. Además, al trabajar contra un *cluster*, es necesario determinar qué datos deben ser físicamente próximos, para minimizar el acceso a varios nodos.

Las BD NoSQL, en general, no soportan ACID en trasacciones que se expander por varios agregados. Hay 4 grandes tipos de modelos.

Características comunes BD NoSQL

No tener esquema; no es necesario definir un esquema para almacenar datos. Esquema en la aplicación.

- **Clave-valor:** Cualquier valor se puede insertar para una clave.
- **Documentales:** No hay restricciones sobre el contenido de cada documento.
- **Column-family:** Cualquier dato se puede almacenar sobre una columna de una fila.
- **Grafos:** las propiedades de nodos y arcos son libres.

Modelos Key-Value

- Estos modelos funcionan mediante pares (Clave, Agregado), donde cada clave única tiene asociado un agregado (que es opaco para la BD, y no tiene restricciones en el contenido).
- Solo se puede consultar por clave, y no tiene esquema ni estructura.

Modelos Documentales

- Las BD entienden la estructura del agregado. Se permite ver y consultar dentro del agregado.
- No tienen esquema y los datos se almacenan en forma de documentos, generalmente JSON o XML.

Modelos Column-Family

- No confundir con el almacenamiento columnar !
- Eficientes en lectura.
- Evoluciona del modelo clave-valor y añade estructura (muy poca). El esquema es el nombre de las familias, lo único que declaramos. La ventaja de estos modelos son las familias.
- Tiene una estructura tabular (no es una tabla) con columnas dispersas.
- Es un clave-*column family*-valor: para cada clave de una fila (cada fila es un agregado) tenemos *column-families* (cada *column-family* define un tipo de registro, y es un bloque dentro del agregado). Para cada familiar tenemos pares clave-valor.

Cassandra

- Aquí una fila solo puede pertenecer a una *column-family*, y una *column-family* puede tener columnas anidadas (supercolumnas, equivalentes a las *column-family* de HBase o Big Table).
- La estructura y almacenamiento de las filas de una tabla puede ser:
  - *Skinny row:* pocas columnas. Mismas columnas en casi todas las filas.
  - *Wide row:* muchas columnas (miles). Filas con columnas muy variadas, qe modelan una lista.

Modelos de grafos

- Motivado por registros simples pero con muchas relaciones entre ellos.
- Modelo compuesto por nodos y arcos, pudiendo tener datos en ambos.
- Las consultas suelen centrarse en acceder a un nodo y mover por los arcos desde el mismo.
- Son rápidas en navegación, pero lentas en inserción.
- Ejemplo Neo4J, que tiene objetos java como propiedades de los nodos y arcos (no tiene esquema). Suele usarse para redes sociales, recomendaciones, etc.

Relaciones

- La BD no conoce la existencia de la relaciones entre datos, se almacenan como una referencia a una clave.
- Las BD relacionales proporcionan mecanismos explícitos para gestionar las relaciones mediantes transacciones ACID o FK. Son poco eficientes si hay que seguir muchas relaciones, ya que se deben hacer muchas *join*.
- Las NoSQL no suelen soportar la modificación de varios agregados en una misma transacción.

Trabajar sin esquema

Ventajas

- No hay que hacer asunciones a priori.
- Facilidad para incorporar cambios en los datos y para trabajar con datos no uniformes.
- Tener esquema resulta inflexible en la inserción de datos, es común un campo de texto done va cualquier cosa.

Desventajas

- Las aplicaciones suelen necesitar cierto formato y semántica en los datos, cierto esquema implicito en los datos. Por esto, hay que minimizar el uso de literales, para que el código se pueda cambiar y mejorar de forma fácil.
- Tener el esquema codificado en las aplicaciones puede dar problemas, ya que la BD no puede utilizarlo para mejorar la eficiencia.
- Los esquemas tienen valor. Una BD sin esquema solo traslada el problema del esquema a la aplicación (es como usar un lenguaje de tipado débil).

Soluciones

- Integrar todo el acceso a la BD en una única app que proporciona servicios web para las demás.
- Delimitar diferentes partes de cada agregado para el acceso de apps diferentes.

Los esquemas en BD relacionales son "flexibles", permiten añadir y quitar columnas por ejemplo. El problema de almacenar los datos de formas nuevas en BD NoSQL es que las apps deben funcionar con datos nuevos y antiguos. Esto se puede arreglar añadiendo una columna y conviviendo con dos columnas, una con nulos hacia arriba y otra con nulos hacia abajo.

Cambios en el esquema

Se le da importancia a los métodos ágiles para cambiar fácilmente de esquema. En NoSQL se pueden hacer cambios rápidos, pero hay que tener cuidado con las migraciones de esquema.

BD relacional

- Un cambio de esquema puede ser un proyecto en si mismo, ya que necesita *scripts* para la migración de datos.
- Con los proyectos nuevos simplemente almacenamos los cambios con los *scripts* de migración de datos.
- Con proyectos *legacy* extraemos el esquema de la BD y procedemos con los proyectos nuevos. Es importante mantener la compatibilidad hacia atrás y tener en cuenta que hay una fase de transición, donde ambos esquemas conviven.

BD NoSQL

- Se intenta no tener que realizar cambios de esquema.
- El esquema está en la aplicación ! Debe *parsear* los datos obtenidos de la BD, por lo que hay que cambiar el código que la app lee y escribe. Si no cambiamos la aplicación, el error de esquema que daría la BD lo dará la aplicación.
- Se hace una migración incremental, ya que mover todos los datos puede ser muy costoso. Transicionamos a partir e la propia aplicación, lee archivos de ambos esquemas (viejo y nuevo) pero escribe en el nuevo. De esta forma, hay datos que no llegan nunca a migrarse.
- En el caso de una BD de grafos, se pueden definir nuevos arcos con el nuevo esquema.

Interés NoSQL

- Funciona aprovechando los recursos de un *cluster* de computación
- Usa el agregado como unidad natural de distribución de datos.

Ventajas

- Capacidad para gestionar un mayor volumen de datos.
- Se puede incrementar el tráfico de operaciones de lectura y escritura (rendimiento), así como la disponibilidad (tolerancia a fallos).

Desventajas

- Sistema más complejo. Usar solo en caso necesario

Formas de replicación de los datos

- Replicación: se hacen varias copias del mismo dato en distintos nodos. Válida para arquitecturas maestro-esclavo o *peer-to-peer*.
- Particionamiento (sharding): repartir los datos entre los nodos.
- Técnicas ortogonales: se pueden combinar

Sharding

El particionamiento es una técnica usada cuando se tienen distintos usuarios accediendo a partes distintas de la base de datos. Para mejorar el rendimiento, se dividen los datos y se coloca cada parte en un nodo distinto. Cada *shard* lee y escribe sobre su propia partición de datos. Esto nos proporciona escalabilidad horizontal.

Debemos intentar mantener la localidad de los datos, es decir, que los datos que se acceden juntos estén en el mismo nodo y cerca en el disco. Para ello, mantenemos los datos que se suelen acceder juntos en el mismo agregado y usamos el agregado como unidad de datos para la distirbución. También es recomendable mantener cerca agregados que se suelen usar juntos.

Además, es recomendable mantener todos los nodos con una carga de datos similar, lo que podemos conseguir haciendo una distribución uniforme de los datos.

Implementar el *sharding* en el nivel de la aplicación es mucho más complejo. Muchas BD NoSQL gestionan el *sharding* de forma automática.

- En cuanto a rendimiento, el particionamiento mejora lectura y escrituras, mientras que la replicación puede mejorar lecturas pero empeorar escrituras.
- En cuanto a la fiabilidad, el particionamiento no mejora la disponibilidad, aumenta la probabilidad de fallo y puede haber fallos parciales.

Un solo servidor

En este caso no existe la distribución de datos. Es la opción preferida por ser el mas simple, y el uso de NoSQL se justificaría por cuestiones relacionadas con el modelo de datos. Las BD de grafos suelen usar esta arquitectura. Los datos agregados se procesan en el nivel de la aplicación y se recuperan juntos.

Combinación particionamiento-replicación

- Con maestro-esclavo: usariamos un maestro para cada partición. Dependiendo de la configuración, podemos elegir maestro y esclavos a nivel *cluster* o para cada partición.
- Con *peer-to-peer*: común en NoSQL con modelos de tipo *column-family*. Se usa una replicación con factor 3, por lo que cada partición tiene 3 copias en 3 nodos y si un nodo falla, sus particiones se distribuyen entre los demás.

Replicación Maestro-Esclavo

- Buena solución cuando la aplicación es intensiva en lecturas. Aquí, los datos se reparte en varios nodos.
- Un nodo es elegido como maestro (autoridad como fuente de los datos y responsable de su actualización) y el resto son esclavos.
- El proceso de replicación sincroniza los esclavos con el maestro. Cuanto más síncrono, mayor consistencia pero menos rendimiento y disponibilidad.

Ventajas

- Alta disponibilidad en lecturas !! ya que si el maestro falla, los esclavos pueden atender peticiones de lectura.
- En caso de fallo del maestro hay que sustituirlo: si tenemos replicación completa del maestro podemos hacer recuperación en caliente (manual o automática).

Problemas

- El maestro es un cuello de botella en modificaciones, dando problemas en caso de muchas escrituras.
- Baja disponibilidad en lecturas.
- Consistencia: un esclavo puede leer datos no actualizados aún (*read-write*).

Replicación Peer-to-peer

- Motivada por los problemas de la replicación maestro-esclavo: escalabilidad en escritura y baja disponibilidad, el maestro es cuello de botella y punto único de fallo.
- Ahora, se elimina la distinción entre maestro y esclavo, todos los nodos se comunican entre ellos y pueden leer y escribir todos los datos.

Problemas

Existen problemas de consistencia por conflictos *write-write*: dos usuarios modifican el mismo dato a la vez en distintos nodos !! Esto es grave ya que genera problemas que perduran, no como los de *read-write*, que generan problemas transitorios.

Soluciones generales

- Coordinar las replicas durantes la escritura: con actualizar la mayoría de forma coordinada es suficiente.
- Asumir inconsistencias e intentar arreglarlas combinando réplicas.
- Más adelante veremos soluciones más específicas.

Consistencia. Relacional vs NoSQL

- Las BD NoSQL dan consistencia eventual, dada por el teorema CAP.
- Las BD relacionales proporcionan alta consistencia con ACID:
  - Atomicidad: gracias a un registro histórico.
  - Consistencia: la garantiza el usuario con el uso de PK, FK y otras restricciones.
  - Aislamiento: el SGBD garantiza la ejecución aislada.
  - Durabilidad: gracias al registro histórico y copias.

Consistencia en modificaciones

Este problema se da cuando dos usuarios intentan modificar el mismo dato en paralelo. Es un conflicto de tipo *write-write*. Por ejemplo, en una *peer-to-peer* dos replicas pueden aplicar las mismas modificaciones en distinto orden: uno lee, modifica y escribe, y otro modifica entre que el primero lee y modifica. El primero escribe y luego el segundo escribe. DEBE HABER UN COMPROMISO ENTRE CONSISTENCIA Y EFICIENCIA.

Estrategias

- **Pesimista**: realiza acciones para evitar casos de inconsistencia, como bloqueos o *timestamps*. Son generalmente poco eficientes (poca concurrencia) y pueden tener interbloqueos.
- **Optimista**: permiten la ejecución normal de la transacción y actúan en el momento del compromiso solo si se detectan problemas, mediante versiones por ejemplo. Una forma optimista de resolver conflictos *write-write* sería almacenar ambos cambios, indicando que existe un conflicto, e intentar mezclar ambas versiones para obtener una versión consistente (complicado).

Concurrencia

Para gestionar el control de concurrencia en distribuido, podemos dar consistencia secuencial, aplicando las operaciones en el mismo orden en todos los nodos.

La mayoría de modelos de distribución usan una única copia de los datos para modificar, lo que simplifica las soluciones para evitar conflictos de escritura (esto no es el caso de la *peer-to-peer*).

Consistencia en lecturas

Se dan inconsistencias cuando una operación de lectura y una de escritura afectan al mismo dato o a un conjunto de datos relacionados. Hay que asegurar la consistencia lógica, varios elementos de datos deben ser consistentes cuando se tratan de forma conjunta (ejemplo de lectura de cuenta bancaria).

Las BD de grafos suelen dar soporte para ACID. Por el contrario, las bases de datos con agregados se aseguran de que las modificaciones sobre un agregado sean atómicas. Esto genera una ventana de inconsistencia al hacer modificaciones sobre varios agregados, ya que se ejecutan como operaciones separadas. En este intervalo de tiempo (muy corto), pueden existir inconsistencias en lectura.

Al introducir replicacion (y añadir redundancia), perdemos consistencia, ya que hay que gestionar las réplicas de forma correcta. En general, a mayor redundancia, menor consistencia

- Hay que asegurar que el mismo dato se lee igual desde todas las réplicas.
- Un problema de lectura es temporal, dando una inconsistencia eventual.
- El nivel de consistencia lo ajustamos dependiendo de la aplicación; muchas operaciones pueden hacerse con niveles bajos de consistencia.

Consistencia de sesión

Los problemas de consistencia con el mismo usuario son del tipo *read-your-write*, y los solucionamos mediante consistencia de sesión:

- *Sticky session*: la sesión se vincula a un nodo. El problema es que baja el rendimiento.
- *Version stamps*: se usan marcas de versiones, lo que asegura que cada operación actúa sobre la última versión. En caso contrario se espera para responder.
- Problemas con *sticky session* y maestro-esclave: se puede leer de los esclavos, pero el maestro es el que modifica. Para esto hay dos soluciones:
  - Modificar en el esclavo y que notifique al maestro.
  - Mover la sesión al maestro mientras se realiza la modificación y, al actualizar cambios, volver al esclavo.

Ejemplo. Reserva habitación

Supongamos dos intentos de reserva de forma concurrente. Cuando se pulsa el botón de reservar, el estado puede haber cambiado ya y no estar disponible. Para solucionarlo, podemos dividir la transacción en dos partes:

- Parte 1: Interacción con el usuario (no hay transacción en la BD)
- Parte 2: Finalización de la transacción (transacción en la BD, vuelve a leer y avisa si no está disponible)
  - Ejecución atómica en una transacción de BD.
  - Verificación de la consistencia de los datos

Relajar consistencia

A veces es necesario sacrificar consistencia para mejorar otras propiedades, incluso en sistemas centralizados, donde teníamos los niveles de consistencia de SQL:

- **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 completamente el uso de transacciones, como sitios *web* muy grandes con necesidad de *sharding*, como eBay.

Teorema CAP

Teorema

Un sistema solo puede tener dos de las tres siguientes características:

- Consistencia
- Disponibilidad (Availability): significado aquí un poco distinto al habitual; si podemos comunicar con un nodo del cluster, entonces podemos leer y escribir datos.
- Tolerancia al particionamiento: el cluster puede sobrevivir a fallos de comunicación entre los nodos que lo separan en varias partes.

Interpretación práctica

Si un sistema puede sufrir particiones (sistema distribuido), hay un compromiso entre la consistencia y la disponibilidad. Aquí, si incrementamos la consistencia, disminuye la disponibilidad. Para mejorar esta última en, por ejemplo, una *peer-to-peer*, podemos elegir un maestro y dirigir las modificaciones hacia él o, incluso, aceptar peticiones incluso si la red falla, perdiendo consistencia (en el caso de la reserva de habitación, tendríamos *overbooking*). Otro ejemplo sería el carro de la compra de Amazon, que escribe todos los carros aunque falle y luego une.

Conclusión

- Como programador hay que buscar consistencia, pero se puede relajar para conseguir disponibilidad y mejorar eficiencia.
- La consistencia de lectura es importante, junto con la duración de la ventana de inconsistencia y la toleracia a lecturas viejas. El último dato es de gran importancia.

NoSQL. Propiedades BASE

- *Basically Available, Soft State, Eventual Consistency.* Las soluciones intermedias buscan un compromiso entre ACID y BASE.
- Con esto, el compromison entre consistencia y latencia es más claro.
  - Mejoramos la consistencia introduciendo más nodos en la interacción, pero empeoramos la eficiencia: cada nuevo nodo incrementa el tiempo de respuesta. Ejemplo: a leer de 5 nodos estaremos más seguros de la consistencia que si leemos de 2. Si al escribir no devolvemos el control hasta que 5 nodos hayan escrito, tendremos más consistencia que si esperamos por 2.
  - La disponibilidad es el límite de la latencia que podemos tolerar en la aplicación. Aquí relacionamos disponibilidad con eficiencia: cuando la latencia es muy alta, nos rendimos y asumimos que la operación no puede terminar.

- En sistemas centralizados, tenemos consistencia y disponibilidad (CA), pero no tolerancia al particionamiento, ya que una sola máquina no se puede particionar. Si el nodo está levantado, entonces está disponible.
- Un *cluster* puede particionarse en dos trozos no conectados. Para tener un *cluster* CA; para ello, debemos pararlo por completo para que no pueda usar la partición (si existe). En este contexto, la disponibilidad significa que cada petición recibida por un nodo que no falla debe ser respondida.

En la práctica, resulta muy costoso parar todos los nodos cuando hay una partición.

Relajar durabilidad

La atomicidad es la clave para la consistencia, definir unidades atómicas de ejecución (transacciones).

Relajar la durabilidad implica aceptar que se pueden perder las actualizaciones de algunas transacciones terminadas con éxito (pérdida de datos en memoria si hay un fallo en el sistema). Es necesario buscar un compromiso entre durabilidad y rendimiento. Una opción es especificar en cada consulta si necesitamos o no durabilidad; esto llevaría a la posible pérdida de solo algunos datos.

Otro factor clave es la durabilidad de replicación. Un ejemplo de problema por falta de esta durabilidad sería cuando el maestro falla y algunos datos del maestro no se habían replicado a un esclavo; ahí, el esclavo busca otro maestro y sigue recibiendo cambios a través del nuevo maestro. Si el maestro antiguo se recupera, puede haber conflictos. Una posible solución es esperar a tener las réplicas generadas para comprometer la transacción, lo que mejora la durabilidad de replicación, empeora el rendimiento de las escrituras y disminuye la disponibilidad (aumenta la probabilidad de fallo de la escritura).

Soluciones reales para la consistencia o durabilidad

Quorums

Buscamos cuántos nodos involucrar para obtener una respuesta. Así, en caso de conflicto de escritura, se puede elegir el valor que más nodos han escrito. Sean W y R el número de nodos que participan en la escritura y lectura, respectivamente, y N el total de nodos involucrados en la replicación (factor de replicación, no el total):

- **Quorum de escritura:** $W > N/2$.
- **Quorum de lectura:** $R + W > N$ (asumiendo replicación *peer-to-peer*). Básicamente cuántos nodos necesitamos leer para asegurar que leemos el último dato.

Hay que ajustar W y R a las necesidades de la aplicación: si queremos lecturas rápidas y consistentes, debemos fijar W alto para que R sea bajo, y las escrituras serán lentas; si queremos escrituras rápidas de baja consistencia, W tiene que ser bajo y, para que R también sea bajo, N debe ser bajo.

Versiones

Las transacciones del sistema no funcionan bien con la interacción con el usuario. Por eso diferenciamos entre transacción de negocio y de sistema. La primera se divide en dos partes, interacción con el usuario y transacción de sistema, donde se lee todo de nuevo para comprobar si ha cambiado (se hace por medio de marcas de versión, como contadores, *timestamps* o *hash* de elementos).

Marcas de versión en múltiples nodos

Funcionan bien en un nodo o maestro-esclavo (solo un nodo controla las versiones); en *peer-to-peer* dos nodos pueden tener valores distintos. Lo más simple es implementar un contador, pero si tenemos más de un maestro necesitamos soluciones más avanzadas. El uso de *timestamps* es problemático, ya que necesita consistencia temporal entre nodos.

Una solución muy común es el *vector stamp*. Aquí, cada nodo tiene asociado un vector con su propio contador. Estos vectores se sincronizan cuando dos nodos se comunican. Los contadores mayor indicará el más reciente, si ambos vectores tienen un valor mayor que el otro entonces existe un conflicto de tipo *write-write*. Este método solo sirve para detectar inconsistencias, se pueden dejar sin resolver o abordarlas.