Importancia de las bases de datos distribuidas en la computación distribuida

Las bases de datos distribuidas han emergido como un componente esencial en la arquitectura de sistemas de computación distribuida. A medida que las organizaciones lidian con cantidades masivas de datos y buscan maximizar la disponibilidad y la escalabilidad de sus aplicaciones, las bases de datos distribuidas proporcionan soluciones críticas.

Escalabilidad

Uno de los beneficios más significativos de las bases de datos distribuidas es su capacidad para escalar horizontalmente. A medida que crece la cantidad de datos y la carga de trabajo, se pueden añadir nuevos nodos al sistema distribuido para manejar el incremento. Este enfoque contrasta con el escalado vertical, que está limitado por las capacidades físicas del hardware.

Ejemplo: Empresas como Google y Amazon manejan petabytes de datos y millones de transacciones por segundo. Utilizan bases de datos distribuidas para escalar sus servicios de manera eficiente, añadiendo más nodos a medida que aumenta la demanda sin comprometer el rendimiento.

Alta disponibilidad y tolerancia a fallos

Las bases de datos distribuidas están diseñadas para ser altamente disponibles, incluso en caso de fallos de nodos individuales. Mediante la replicación de datos en múltiples nodos y ubicaciones geográficas, las bases de datos distribuidas aseguran que los datos permanezcan accesibles, minimizando el tiempo de inactividad.

Ejemplo: En servicios financieros y de salud, donde la disponibilidad continua es crítica, las bases de datos distribuidas aseguran que las aplicaciones permanezcan operativas incluso si un nodo o centro de datos falla. Esto es esencial para garantizar un servicio ininterrumpido y la integridad de los datos en aplicaciones sensibles.

Rendimiento mejorado

La distribución de datos y la carga de trabajo entre múltiples nodos mejora significativamente el rendimiento del sistema. Las operaciones de lectura y escritura pueden realizarse en paralelo, reduciendo la latencia y aumentando el throughput.

Ejemplo: Redes sociales y plataformas de streaming, como Facebook y Netflix, utilizan bases de datos distribuidas para manejar grandes volúmenes de solicitudes de usuarios simultáneamente, asegurando que las interacciones sean rápidas y fluidas.

Consistencia y flexibilidad

Las bases de datos distribuidas permiten un equilibrio entre consistencia, disponibilidad y tolerancia a particiones (Teorema CAP). Dependiendo de los requisitos de la aplicación, se pueden configurar diferentes niveles de consistencia, desde consistencia eventual hasta consistencia fuerte.

Ejemplo: En aplicaciones de e-commerce, donde la disponibilidad es crucial, se puede optar por una consistencia eventual para asegurar que el sistema permanezca operativo bajo altas cargas, mientras que

en aplicaciones financieras se puede priorizar la consistencia fuerte para asegurar la exactitud de las transacciones.

Gestión eficiente de datos geográficamente distribuidos

Las bases de datos distribuidas son esenciales para aplicaciones globales que necesitan gestionar datos en múltiples regiones. Permiten que los datos se almacenen cerca de los usuarios finales, reduciendo la latencia de acceso y mejorando la experiencia del usuario.

Ejemplo: Aplicaciones como Google Spanner y Amazon DynamoDB utilizan replicación geográfica para asegurar que los datos estén disponibles y sean accesibles rápidamente desde cualquier parte del mundo, proporcionando una experiencia de usuario consistente y de alta calidad.

Soporte para diversos modelos de datos

Las bases de datos distribuidas modernas soportan una variedad de modelos de datos (clave-valor, documento, gráfico, etc.), ofreciendo flexibilidad para gestionar diferentes tipos de datos según las necesidades de la aplicación.

Ejemplo: MongoDB permite almacenar datos semiestructurados en formato de documento, mientras que Neo4j se especializa en datos de grafos, permitiendo a las organizaciones elegir la base de datos que mejor se adapte a su modelo de datos y casos de uso específicos.

Reducción de costos operativos

Al permitir la distribución de la carga de trabajo y el almacenamiento en hardware de menor costo, las bases de datos distribuidas pueden reducir los costos operativos. Esto es especialmente importante para startups y organizaciones que buscan escalar sin incurrir en gastos significativos en infraestructura.

Ejemplo: Empresas emergentes pueden utilizar bases de datos distribuidas en la nube para empezar con una infraestructura mínima y escalar según la demanda, pagando solo por los recursos que utilizan.

Innovación y adaptabilidad

Las bases de datos distribuidas facilitan la innovación al permitir la rápida implementación y prueba de nuevas funcionalidades. Su capacidad para integrarse con tecnologías emergentes como la inteligencia artificial y el aprendizaje automático también permite a las organizaciones adaptar y evolucionar sus aplicaciones rápidamente.

Ejemplo: Plataformas como Amazon Redshift y Google BigQuery permiten análisis de datos en tiempo real y la integración con herramientas de aprendizaje automático, facilitando la toma de decisiones basada en datos y la innovación continua en productos y servicios.

Particionamiento y sharding en sistemas distribuidos

El particionamiento y sharding son técnicas esenciales en la gestión de bases de datos y sistemas distribuidos para mejorar el rendimiento, la escalabilidad y la disponibilidad. Estas técnicas permiten distribuir los datos en múltiples nodos, facilitando el acceso concurrente y reduciendo los cuellos de botella.

Tipos de particionamiento

El particionamiento de datos se puede clasificar en varias categorías, cada una con sus propias características y casos de uso específicos. Los principales tipos de particionamiento son horizontal,

vertical, de rango, de lista y de hash.

Particionamiento Horizontal

El particionamiento horizontal, también conocido como sharding, divide una tabla en filas y distribuye estas filas en diferentes nodos o particiones. Cada partición contiene un subconjunto de las filas de la tabla original. Esta técnica es especialmente útil para mejorar la escalabilidad y la gestión de grandes volúmenes de datos.

Ejemplo: Una tabla de usuarios se puede particionar horizontalmente de manera que los usuarios con IDs entre 1 y 1000 se almacenen en una partición, mientras que los usuarios con IDs entre 1001 y 2000 se almacenen en otra.

Particionamiento vertical

El particionamiento vertical divide una tabla en columnas y distribuye estas columnas en diferentes particiones. Cada partición contiene un subconjunto de las columnas de la tabla original. Esta técnica es útil para optimizar el rendimiento de consultas que acceden frecuentemente a ciertas columnas.

Ejemplo: Una tabla de productos se puede particionar verticalmente de manera que las columnas que contienen información básica (nombre, precio) se almacenen en una partición, mientras que las columnas con información adicional (descripción, especificaciones) se almacenen en otra.

Particionamiento de rango

El particionamiento de rango divide los datos en particiones basadas en rangos definidos. Cada partición contiene un rango específico de valores de una columna de la tabla. Esta técnica es útil cuando los datos tienen un orden natural y las consultas suelen realizarse sobre rangos de valores.

Ejemplo: Una tabla de transacciones puede particionarse por rangos de fechas, de manera que las transacciones de enero se almacenen en una partición, las de febrero en otra, y así sucesivamente.

Particionamiento de lista

El particionamiento de lista divide los datos en particiones basadas en valores específicos de una columna. Cada partición contiene un conjunto predefinido de valores. Esta técnica es útil cuando los datos pueden clasificarse en grupos discretos.

Ejemplo: Una tabla de empleados puede particionarse por departamento, de manera que los empleados del departamento de ventas se almacenen en una partición, los del departamento de marketing en otra, etc.

Particionamiento de Hash

El particionamiento de hash utiliza una función hash para distribuir los datos en diferentes particiones. Cada partición contiene los datos cuyo valor de la función hash cae dentro de un rango específico. Esta técnica es útil para distribuir uniformemente los datos y evitar hotspots.

Ejemplo: Una tabla de pedidos puede particionarse aplicando una función hash a los IDs de los pedidos, distribuyendo así los pedidos en diferentes particiones de manera uniforme.

Estrategias de sharding y balanceo de carga

El **sharding** es una forma de particionamiento horizontal que distribuye los datos en múltiples shards (fragmentos) para mejorar la escalabilidad y el rendimiento. Existen varias estrategias para implementar sharding y balancear la carga entre los shards.

Sharding estático vs. dinámico

- Sharding estático: En el sharding estático, los datos se distribuyen en shards de manera fija y no cambian una vez definidos. Esta estrategia es simple pero puede llevar a un desequilibrio de carga si algunos shards crecen más rápido que otros.
- Sharding dinámico: En el sharding dinámico, los datos se redistribuyen periódicamente para mantener el equilibrio de carga. Esta estrategia es más compleja pero garantiza que ningún shard se sobrecargue.

Estrategias de sharding

1. Sharding por rango: Divide los datos en shards basados en rangos de valores. Es simple de implementar pero puede llevar a un desequilibrio si los datos no se distribuyen uniformemente.
2. Sharding por Hash: Utiliza una función hash para distribuir los datos entre los shards. Esto asegura una distribución uniforme pero puede complicar la ejecución de consultas que requieren operaciones de rango.
3. Sharding por entidad: Distribuye los datos basados en una entidad lógica. Por ejemplo, todos los datos relacionados con un usuario específico pueden almacenarse en el mismo shard.
4. Sharding híbrido: Combina múltiples estrategias de sharding para aprovechar los beneficios de cada una y minimizar sus desventajas.

Balanceo de carga

El balanceo de carga en sistemas sharded es crucial para asegurar que todas las shards operen de manera eficiente. Las técnicas comunes de balanceo de carga incluyen:

1. Rebalanceo proactivo: Redistribuye los datos periódicamente para prevenir el desbalance antes de que ocurra.
2. Rebalanceo reactivo: Redistribuye los datos en respuesta a un desbalance detectado.
3. Particionamiento dinámico: Ajusta el número de particiones dinámicamente en función de la carga y el crecimiento de los datos.

Desafíos en la redistribución de datos y re-sharding

La redistribución de datos y el re-sharding presentan varios desafíos en sistemas distribuidos, incluyendo la complejidad de la migración de datos, la gestión de la consistencia y la minimización del tiempo de inactividad.

Complejidad de la migración de datos

Redistribuir datos entre shards puede ser una tarea compleja que requiere la coordinación entre múltiples nodos. La migración debe manejarse cuidadosamente para evitar pérdidas de datos y mantener la integridad del sistema. Además, la migración puede generar un overhead significativo en la red y los recursos del sistema.

Ejemplo: En un sistema de e-commerce, migrar datos de un shard a otro puede requerir la transferencia de grandes volúmenes de información sobre transacciones, inventarios y usuarios, lo que puede afectar el

rendimiento del sistema.

**Gestión de la consistencia*

Mantener la consistencia de los datos durante el proceso de redistribución es crucial. Las operaciones de lectura y escritura deben coordinarse para asegurar que los usuarios no vean datos inconsistentes. Se pueden utilizar técnicas como el locking y la replicación para gestionar la consistencia durante el re-sharding.

Ejemplo: Al redistribuir datos de un shard a otro, se pueden utilizar bloqueos para asegurar que no se realicen escrituras en los datos mientras se migran, o se puede replicar temporalmente los datos en ambos shards hasta que la migración se complete.

Minimización del tiempo de inactividad

El re-sharding puede requerir tiempo de inactividad del sistema, lo cual puede ser inaceptable para aplicaciones críticas. Las estrategias para minimizar el tiempo de inactividad incluyen la migración en caliente (hot migration), donde los datos se transfieren mientras el sistema sigue operativo, y el uso de réplicas para desviar el tráfico mientras se realiza el re-sharding.

Ejemplo: En un sistema de banca en línea, el re-sharding debe realizarse sin interrumpir el acceso de los usuarios a sus cuentas. Esto puede lograrse mediante la migración en caliente y el uso de réplicas para manejar las operaciones de los usuarios mientras los datos se redistribuyen.

Fundamentos teóricos del Teorema CAP

El teorema CAP, propuesto por Eric Brewer en el año 2000 y formalizado posteriormente por Seth Gilbert y Nancy Lynch en 2002, es un principio fundamental en el diseño y análisis de sistemas distribuidos. Este teorema establece que en un sistema distribuido, es imposible alcanzar simultáneamente los tres siguientes objetivos:

1. Consistencia (Consistency, C): Cada lectura recibe la respuesta más reciente escrita, garantizando que todos los nodos del sistema ven el mismo dato al mismo tiempo.
2. Disponibilidad (Availability, A): Cada solicitud de lectura y escritura recibe una respuesta (ya sea exitosa o fallida), es decir, el sistema siempre está disponible para recibir operaciones.
3. Tolerancia a particiones (Partition Tolerance, P): El sistema continúa operando a pesar de la partición de la red que separa los nodos.

Dado que es imposible satisfacer simultáneamente los tres objetivos, los diseñadores de sistemas distribuidos deben hacer compromisos (trade-offs) y elegir dos de los tres. Esta elección depende del tipo de aplicación y de los requisitos específicos de consistencia, disponibilidad y tolerancia a particiones.

Conceptos de consistencia, disponibilidad y tolerancia a particiones

Consistencia

La consistencia en el contexto del Teorema CAP se refiere a la propiedad de que todas las copias de un dato en un sistema distribuido son iguales en cualquier momento dado. Esto implica que cualquier operación de lectura después de una operación de escritura debe devolver el valor actualizado.

Existen varios niveles de consistencia, desde la consistencia fuerte (strong consistency) hasta la consistencia eventual (eventual consistency). La consistencia fuerte asegura que una vez que una escritura

se completa, cualquier lectura subsecuente proporcionará ese valor escrito en todos los nodos del sistema. La consistencia eventual, por otro lado, garantiza que, dado suficiente tiempo sin nuevas escrituras, todos los nodos del sistema convergerán al mismo valor.

Disponibilidad

La disponibilidad implica que el sistema debe responder a todas las solicitudes de lectura y escritura en todo momento, incluso en presencia de fallos parciales en el sistema. En otras palabras, cada solicitud debe recibir una respuesta (aunque sea un error), asegurando que el sistema esté siempre operativo para los usuarios.

Para asegurar la disponibilidad, los sistemas distribuidos pueden utilizar técnicas como el enrutamiento de solicitudes a réplicas disponibles y la utilización de arquitecturas sin estado donde los nodos pueden fallar y recuperarse sin pérdida de datos o funcionalidad.

Tolerancia a particiones

La tolerancia a particiones es la capacidad del sistema de continuar operando correctamente a pesar de las divisiones (particiones) en la red que separan los nodos del sistema. En un entorno distribuido, las particiones de red son inevitables debido a fallos de hardware, problemas de red o mantenimiento. Un sistema que tolera particiones puede continuar procesando solicitudes y manteniendo la coherencia y disponibilidad de los datos, incluso si algunos nodos no pueden comunicarse entre sí.

Ejemplos y aplicaciones prácticas del teorema CAP

Los sistemas distribuidos deben hacer compromisos basados en el Teorema CAP. A continuación, se presentan algunos ejemplos y aplicaciones prácticas:

Consistencia y disponibilidad (CA)

En sistemas donde la partición de la red es una rareza o donde se puede tolerar la indisponibilidad temporal durante las particiones, se prioriza la consistencia y disponibilidad. Ejemplos incluyen bases de datos relacionales tradicionales como PostgreSQL y MySQL que operan bajo suposiciones de entornos de red confiables.

Consistencia y tolerancia a particiones (CP)

Sistemas que priorizan la consistencia y la tolerancia a particiones, pero pueden sacrificar la disponibilidad durante una partición de red, incluyen las bases de datos distribuidas como HBase y MongoDB cuando están configuradas para un entorno altamente consistente. Estos sistemas garantizan que los datos sean consistentes y permiten que el sistema se recupere después de una partición sin comprometer la integridad de los datos.

Disponibilidad y tolerancia a particiones (AP)

Para aplicaciones donde la disponibilidad es crítica y se puede tolerar la consistencia eventual, se priorizan la disponibilidad y la tolerancia a particiones. Ejemplos incluyen sistemas de almacenamiento distribuido como Cassandra y DynamoDB. Estos sistemas aseguran que los datos sean eventualmente consistentes, pero siempre están disponibles para lecturas y escrituras.

Compromisos y estrategias para gestionar los trade-offs en sistemas distribuidos

Técnicas de replicación

La replicación de datos es una estrategia común para gestionar los trade-offs del Teorema CAP. Existen dos principales estrategias de replicación:

- Replicación sincrónica: Garantiza la consistencia fuerte ya que cada operación de escritura debe ser confirmada por todas las réplicas antes de considerarse completa. Sin embargo, esto puede afectar la disponibilidad si alguna réplica no está disponible.
- Replicación asincrónica: Mejora la disponibilidad ya que las operaciones de escritura se consideran completas tan pronto como se escriben en la réplica principal, y luego se propagan a las réplicas secundarias. Esto, sin embargo, puede comprometer la consistencia.

Modelos de consistencia

Los diferentes modelos de consistencia permiten gestionar los trade-offs de manera efectiva:

- Consistencia fuerte: Asegura que todos los nodos vean el mismo dato al mismo tiempo, pero puede reducir la disponibilidad durante particiones.
- Consistencia débil: Permite que las lecturas vean datos obsoletos, mejorando la disponibilidad.
- Consistencia eventual: Garantiza que todos los nodos convergerán eventualmente al mismo valor, equilibrando la consistencia y la disponibilidad.

Algoritmos de consenso

Los algoritmos de consenso, como Paxos y Raft, ayudan a alcanzar decisiones consistentes en sistemas distribuidos. Estos algoritmos son fundamentales para implementar consistencia fuerte, pero pueden ser complejos y costosos en términos de rendimiento.

Sistemas de tolerancia a fallos

Implementar técnicas de tolerancia a fallos, como la redundancia y la recuperación automática, mejora la disponibilidad y la tolerancia a particiones. Los sistemas distribuidos modernos utilizan arquitecturas sin estado, donde los nodos pueden fallar y recuperarse sin afectar la disponibilidad general del sistema.

Balanceo de carga y particionamiento

El balanceo de carga y el particionamiento de datos son cruciales para mantener la disponibilidad y la eficiencia del sistema. Al distribuir las solicitudes de manera equitativa entre los nodos y particionar los datos, se puede reducir la carga en cada nodo y mejorar la capacidad de respuesta del sistema.

Modelos de consistencia en sistemas distribuidos

En el contexto de los sistemas distribuidos, la consistencia es un concepto crucial que determina cómo y cuándo los datos se actualizan y sincronizan entre diferentes nodos del sistema. Existen varios modelos de consistencia, cada uno ofreciendo diferentes garantías y compromisos.

Consistencia fuerte vs. Consistencia eventual

Consistencia fuerte

La consistencia fuerte, también conocida como consistencia linealizable, asegura que cualquier operación de lectura en un sistema distribuido devolverá el valor más reciente escrito. En otras palabras, una vez que se confirma una operación de escritura, cualquier operación de lectura subsecuente reflejará este valor,

independientemente del nodo desde el cual se realiza la lectura. Este modelo garantiza que todas las operaciones parezcan ejecutarse de manera secuencial, respetando el orden real en el que ocurrieron.

Para lograr consistencia fuerte, las operaciones de escritura deben ser propagadas y confirmadas por todos los nodos antes de considerarse completas. Esto puede implicar un alto costo en términos de latencia y rendimiento, especialmente en sistemas con alta disponibilidad y escalabilidad. Algoritmos de consenso como Paxos y Raft son comúnmente utilizados para implementar consistencia fuerte, asegurando que todos los nodos acuerden el mismo estado del sistema.

Consistencia eventual

La consistencia eventual, en contraste, asegura que si no se realizan nuevas escrituras, todos los nodos del sistema eventualmente convergerán al mismo valor. Este modelo no garantiza que una lectura subsecuente a una escritura refleje inmediatamente el nuevo valor escrito. En su lugar, permite que los nodos actualicen sus estados de manera asincrónica, lo que puede resultar en lecturas temporales de valores desactualizados.

La consistencia eventual es adecuada para aplicaciones donde la disponibilidad y la tolerancia a particiones son críticas, y donde una ligera inconsistencia temporal es aceptable. Sistemas como Amazon DynamoDB y Apache Cassandra utilizan consistencia eventual, priorizando la disponibilidad y el rendimiento sobre la consistencia inmediata.

Consistencia causal, consistencia de sesión y otros modelos intermedios

Consistencia causal

La consistencia causal se basa en la idea de que ciertas operaciones pueden depender causalmente de otras. Por ejemplo, si una operación A ocurre antes de una operación B, cualquier lectura que vea B también debe ver A. Este modelo es más relajado que la consistencia fuerte pero más estricto que la consistencia eventual, ya que preserva el orden causal de las operaciones.

Para implementar consistencia causal, se utilizan técnicas como relojes vectoriales que rastrean las dependencias entre operaciones. Cada nodo mantiene un vector de relojes que se actualiza con cada operación, permitiendo a los nodos determinar el orden causal de los eventos.

Consistencia de sesión

La consistencia de sesión es un modelo intermedio que asegura la consistencia dentro de una sesión de cliente. Una vez que un cliente inicia una sesión, todas las lecturas y escrituras realizadas dentro de esa sesión son consistentes. Esto significa que un cliente verá sus propias escrituras y lecturas subsecuentes reflejarán esos valores, independientemente del nodo desde el cual se realizan.

Este modelo es útil para aplicaciones donde la consistencia a nivel de usuario es importante, pero donde la consistencia global puede ser relajada. Sistemas como Amazon S3 y Google Cloud Storage ofrecen consistencia de sesión, proporcionando una experiencia coherente para usuarios individuales.

Técnicas y algoritmos para asegurar diferentes niveles de consistencia

Algoritmos de consenso: Paxos y Raft

Para asegurar consistencia fuerte, se emplean algoritmos de consenso como Paxos y Raft. Estos algoritmos garantizan que todos los nodos de un sistema distribuido acuerden el mismo valor, a pesar de fallos y particiones en la red.

Paxos: Paxos es un algoritmo de consenso que se basa en tres roles principales: proponentes, aceptadores y aprendices. Los proponentes sugieren valores, los aceptadores acuerdan o rechazan estos valores, y los aprendices observan el valor aceptado. Paxos garantiza que, a pesar de fallos parciales, todos los nodos convergerán eventualmente al mismo valor aceptado.

Raft: Raft simplifica el proceso de consenso dividiendo el algoritmo en componentes más comprensibles. Utiliza un líder para coordinar las operaciones de escritura y replicar los cambios a los seguidores. Raft garantiza que solo un líder válido puede proponer cambios, lo que facilita la implementación y el entendimiento del consenso distribuido.

Relojes vectoriales y relojes lógicos

Para implementar consistencia causal, se utilizan relojes vectoriales y relojes lógicos. Estos relojes permiten a los nodos rastrear el orden causal de las operaciones y asegurar que las lecturas reflejen las dependencias causales correctas.

Relojes vectoriales: Cada nodo en el sistema mantiene un vector de contadores, uno para cada nodo. Los relojes vectoriales permiten determinar si una operación ocurrió antes, después o de manera concurrente con respecto a otra operación. Este mecanismo es esencial para preservar el orden causal en sistemas distribuidos.

Relojes lógicos: Los relojes lógicos, como los relojes de Lamport, utilizan un único contador global que se incrementa con cada operación. Aunque no preservan el orden causal completo, son útiles para determinar un orden parcial de eventos en sistemas distribuidos.

Técnicas de replicación y particionamiento

La replicación y el particionamiento son técnicas clave para asegurar disponibilidad y consistencia en sistemas distribuidos. Estas técnicas se emplean tanto en modelos de consistencia fuerte como en modelos de consistencia eventual.

- Replicación sincrónica: Utilizada para consistencia fuerte, la replicación sincrónica asegura que todas las réplicas de un dato sean actualizadas antes de confirmar una operación de escritura. Esto puede implicar una mayor latencia, pero garantiza que las lecturas subsecuentes reflejen el valor más reciente.
- Replicación asincrónica: Empleada en consistencia eventual, la replicación asincrónica permite que las operaciones de escritura se consideren completas tan pronto como se escriben en la réplica principal. Las réplicas secundarias se actualizan posteriormente, lo que puede llevar a lecturas temporales de valores desactualizados.
- Particionamiento: El particionamiento de datos distribuye los datos entre múltiples nodos para mejorar el rendimiento y la disponibilidad. Cada partición puede replicarse para asegurar la tolerancia a fallos. El particionamiento efectivo es crucial para mantener la consistencia y disponibilidad en sistemas a gran escala.

Control de concurrencia distribuida

El control de concurrencia es un aspecto fundamental en el diseño y la implementación de sistemas distribuidos. Asegurar que múltiples transacciones o procesos puedan ejecutarse de manera simultánea sin causar inconsistencias en los datos es crucial para mantener la integridad y la coherencia del sistema.

Algoritmos de control de concurrencia

Control de concurrencia optimista

El control de concurrencia optimista se basa en la suposición de que los conflictos entre transacciones son raros. En este enfoque, las transacciones se ejecutan sin restricciones inicialmente y solo se verifican los conflictos al final de la transacción, durante una fase de validación.

1. Fase de lectura: Durante esta fase, la transacción lee los valores de la base de datos y almacena estos valores localmente, sin realizar ningún bloqueo.
2. Fase de validación: Antes de que la transacción se comprometa, se verifica si ha habido algún conflicto con otras transacciones concurrentes. Si se detecta un conflicto, la transacción se aborta y se vuelve a intentar.
3. Fase de escritura: Si la transacción pasa la fase de validación, los cambios se aplican a la base de datos.

Este método es eficiente en entornos donde los conflictos son infrecuentes, ya que evita el overhead asociado con el bloqueo de recursos. Sin embargo, en sistemas con alta contención de datos, el control de concurrencia optimista puede resultar en un alto número de abortos y reintentos de transacciones.

Control de concurrencia pesimista

El control de concurrencia pesimista asume que los conflictos son comunes y, por lo tanto, evita activamente las colisiones mediante el bloqueo de recursos. Este enfoque utiliza bloqueos para garantizar que una transacción tenga acceso exclusivo a los datos mientras los modifica.

1. Bloqueos de lectura (Shared Locks): Permiten que múltiples transacciones lean un recurso simultáneamente, pero ninguna puede escribir en él mientras el bloqueo está en vigor.
2. Bloqueos de escritura (Exclusive Locks): Garantizan que solo una transacción puede escribir en el recurso y ninguna otra puede leerlo o escribirlo mientras el bloqueo está en vigor.

El control de concurrencia pesimista es efectivo en entornos donde los conflictos son frecuentes, ya que asegura que las transacciones no interfieran entre sí. Sin embargo, puede introducir problemas de rendimiento debido al overhead de la gestión de bloqueos y puede llevar a condiciones de bloqueo mutuo (deadlock) si no se maneja adecuadamente.

Técnicas de control de concurrencia distribuida

Relojes lógicos

Los relojes lógicos son herramientas fundamentales en la sincronización y el control de concurrencia en sistemas distribuidos. Proveen un medio para ordenar eventos sin depender del tiempo físico real.

Relojes de Lamport

El reloj de Lamport es un mecanismo que asigna un valor de tiempo lógico a cada evento en un sistema distribuido, asegurando un orden causal parcial de los eventos.

1. Incremento de reloj: Cada proceso mantiene un contador. Antes de emitir un evento, el proceso incrementa su contador.
2. Envío de mensajes: Al enviar un mensaje, el proceso incluye su contador incrementado en el mensaje.
3. Recepción de mensajes: Al recibir un mensaje, el proceso ajusta su contador al valor máximo entre su contador actual y el contador recibido, incrementado en uno.

El reloj de Lamport garantiza que si un evento A ocurre antes que un evento B (causalmente), entonces el valor de tiempo lógico asignado a A será menor que el valor asignado a B.

Relojes vectoriales

Los relojes vectoriales extienden el concepto de los relojes de Lamport para proporcionar un orden causal más fuerte.

1. Vectores de reloj: Cada proceso mantiene un vector de enteros, uno por cada proceso en el sistema.
2. Actualización de vectores: Antes de emitir un evento, el proceso incrementa su propio componente del vector.
3. Envío y recepción de mensajes: Al enviar un mensaje, el proceso incluye su vector de reloj. Al recibir un mensaje, el proceso actualiza su vector tomando el máximo de cada componente entre su vector y el vector recibido.

Los relojes vectoriales permiten determinar no solo el orden de los eventos, sino también identificar si los eventos son concurrentes.

Protocolos de bloqueo distribuidos

Los protocolos de bloqueo distribuidos son esenciales para coordinar el acceso a los recursos compartidos en un sistema distribuido.

Protocolo de dos fases (Two-Phase Locking, 2PL)

El protocolo de dos fases es uno de los mecanismos más utilizados para garantizar la serializabilidad de las transacciones en un entorno distribuido. Consiste en dos fases:

1. Fase de crecimiento: Durante esta fase, una transacción puede adquirir nuevos bloqueos pero no puede liberar ninguno.
2. Fase de reducción: Una vez que una transacción libera un bloqueo, entra en la fase de reducción y no puede adquirir nuevos bloqueos.

El protocolo de dos fases asegura que todas las transacciones sigan un orden serializable, evitando conflictos de concurrencia. Sin embargo, puede llevar a situaciones de bloqueo mutuo, que deben ser gestionadas mediante técnicas de detección y resolución de deadlocks.

Protocolo de bloqueo por tiempos (Timestamp Ordering)

El protocolo de ordenación por marcas de tiempo (timestamp ordering) asigna una marca de tiempo única a cada transacción y utiliza estas marcas para ordenar las operaciones de las transacciones.

1. Lecturas y escrituras: Las operaciones de lectura y escritura de las transacciones se ejecutan de acuerdo con sus marcas de tiempo.
2. Conflictos de ordenación: Si una operación de escritura de una transacción T2 con marca de tiempo mayor que una lectura anterior de otra transacción T1 intenta actualizar el mismo dato, T2 se aborta para mantener el orden de las marcas de tiempo.

Este protocolo garantiza la serializabilidad temporal, asegurando que las transacciones se ejecuten en un orden que respete sus marcas de tiempo.

Protocolo de bloqueo multiversión (Multiversion Concurrency Control, MVCC)

El protocolo de control de concurrencia multiversión (MVCC) permite que múltiples versiones de un dato existan simultáneamente, mejorando el rendimiento y la concurrencia.

1. Versiones de datos: Cada escritura crea una nueva versión del dato, etiquetada con la marca de tiempo de la transacción que la creó.
2. Lecturas: Las operaciones de lectura seleccionan la versión más reciente del dato que precede a la marca de tiempo de la transacción de lectura.
3. Conflictos de escritura: Las escrituras se verifican contra la marca de tiempo de las versiones para asegurar que no se sobreescriban versiones inconsistentes.

MVCC permite que las transacciones de lectura no bloqueen las transacciones de escritura y viceversa, mejorando significativamente la concurrencia en sistemas con alta contención de datos.

Replicación de datos en sistemas distribuidos

La replicación de datos es una técnica fundamental en sistemas distribuidos que garantiza la disponibilidad, tolerancia a fallos y rendimiento. Consiste en mantener múltiples copias de datos en diferentes nodos del sistema, permitiendo que el sistema siga operando incluso si algunos nodos fallan.

Replicación síncrona vs. asíncrona

Replicación síncrona

La replicación síncrona asegura que los datos se escriben en todas las réplicas antes de confirmar la operación al cliente. Este enfoque garantiza una consistencia fuerte, ya que todas las copias de los datos están actualizadas en todo momento. La replicación síncrona implica los siguientes pasos:

1. Escritura en réplicas: Cuando se recibe una solicitud de escritura, el dato se envía a todas las réplicas.
2. Confirmación de réplicas: Cada réplica confirma la escritura.
3. Confirmación al cliente: Una vez que todas las réplicas han confirmado, se envía una confirmación al cliente.

Este método asegura que cualquier lectura subsecuente reflejará el valor más reciente. Sin embargo, puede introducir latencia significativa debido a la necesidad de esperar las confirmaciones de todas las réplicas, y puede afectar la disponibilidad del sistema si alguna réplica no está disponible.

Replicación asíncrona

En la replicación asíncrona, la operación de escritura se confirma al cliente tan pronto como se completa en la réplica principal, sin esperar a que las réplicas secundarias se actualicen. Este enfoque prioriza la disponibilidad y el rendimiento sobre la consistencia inmediata. Los pasos son:

1. Escritura en réplica principal: La solicitud de escritura se completa en la réplica principal.
2. Confirmación al cliente: La operación se confirma al cliente inmediatamente.
3. Propagación a réplicas: La réplica principal propaga los cambios a las réplicas secundarias en segundo plano.

Este método reduce la latencia percibida por el cliente y mejora la disponibilidad del sistema, ya que no depende del estado de las réplicas secundarias. Sin embargo, puede llevar a situaciones donde las lecturas subsecuentes devuelvan datos desactualizados hasta que todas las réplicas se sincronicen.

Protocolos de replicación: Raft y Paxos

Los protocolos de replicación como Raft y Paxos son fundamentales para coordinar la consistencia en sistemas distribuidos. Ambos protocolos aseguran que todas las réplicas acuerden sobre el mismo estado del sistema, incluso en presencia de fallos de red o nodos.

Mecanismos de consenso en sistemas distribuidos

El consenso en sistemas distribuidos es crucial para garantizar que todos los nodos acuerden el mismo estado, especialmente en presencia de fallos. Los mecanismos de consenso aseguran la coherencia de los datos replicados y son fundamentales para la integridad del sistema.

Algoritmos de consenso

- Paxos: Paxos garantiza la seguridad (nunca dos nodos diferentes aceptan diferentes valores en la misma ronda) y la liveness (eventualmente, un valor es aceptado) en sistemas distribuidos. Su robustez lo hace ideal para aplicaciones donde la consistencia es crítica.
- Raft: Diseñado para ser más comprensible que Paxos, Raft asegura que el sistema siga siendo operativo a través de elecciones de líder y replicación de logs. Su enfoque modular facilita la implementación y el mantenimiento.
- Algoritmo de acuerdo Bizantino (BFT): En entornos donde los nodos pueden ser maliciosos o fallar de manera arbitraria, los algoritmos BFT, como PBFT (Practical Byzantine Fault Tolerance), aseguran que los nodos honestos lleguen a un consenso a pesar de la presencia de nodos bizantinos.

Protocolos de bloqueo y validación

Además de los algoritmos de consenso, los sistemas distribuidos utilizan protocolos de bloqueo y validación para asegurar la coherencia.

- Two-Phase Commit (2PC): Es un protocolo de coordinación que asegura la atomicidad de las transacciones distribuidas. Consta de una fase de preparación, donde el coordinador solicita a los participantes que preparen la transacción, y una fase de compromiso, donde el coordinador decide si comprometer o abortar la transacción basada en las respuestas de los participantes.
- Three-Phase Commit (3PC): Mejora el 2PC añadiendo una fase intermedia para evitar bloqueos en caso de fallos. Introduce una fase de precompromiso que asegura que todos los nodos acuerden comprometer antes de que la transacción sea finalizada.
- Validation-Based Protocols: Utilizados en control de concurrencia optimista, estos protocolos validan las transacciones al final de su ejecución para asegurarse de que no haya conflictos antes de comprometer los cambios.

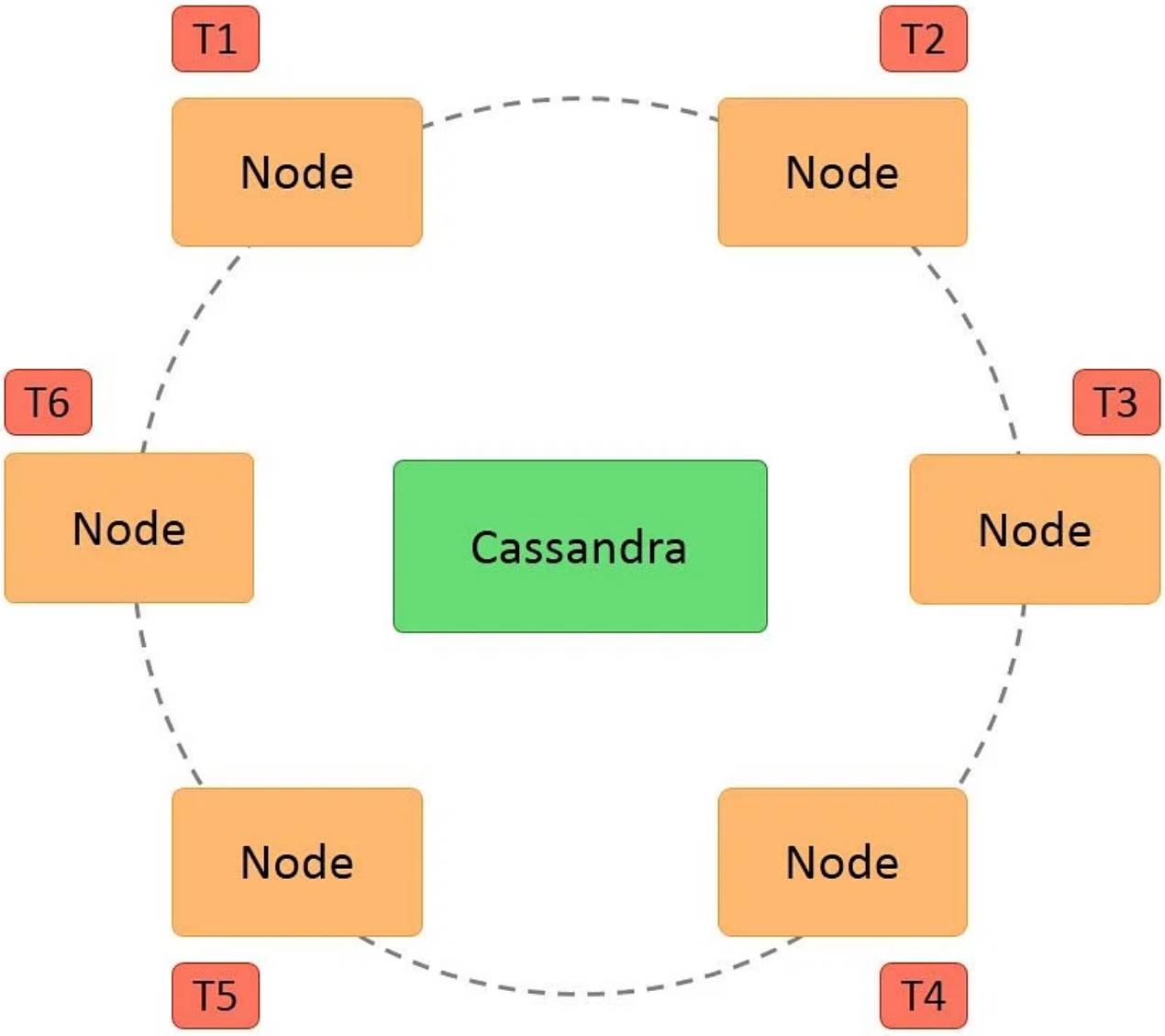
Arquitecturas y modelos de datos en sistemas distribuidos

Apache Cassandra y MongoDB son dos bases de datos NoSQL ampliamente utilizadas que destacan por su capacidad para manejar grandes volúmenes de datos y su escalabilidad horizontal. A pesar de sus diferencias en arquitectura y modelo de datos, ambas ofrecen soluciones robustas para el particionamiento, replicación y balanceo de carga.

Apache Cassandra

Apache Cassandra adopta una arquitectura peer-to-peer, donde todos los nodos en un clúster son iguales, eliminando puntos únicos de fallo y permitiendo una escalabilidad lineal. En este modelo, no existe un nodo maestro; en cambio, todos los nodos pueden recibir solicitudes de lectura y escritura, distribuyendo la carga de manera equitativa. Los nodos se comunican entre sí mediante un protocolo de gossip para compartir información sobre el estado del clúster, asegurando que todos los nodos tengan una visión coherente del sistema.

La distribución de datos en Cassandra se basa en un anillo lógico, donde cada nodo es responsable de un rango de hashes de claves. Este enfoque permite una distribución uniforme de los datos y facilita la adición o eliminación de nodos sin interrumpir el servicio.



Modelo de consistencia eventual y reconciliación de datos

Cassandra ofrece un modelo de consistencia eventual, lo que significa que, dado suficiente tiempo sin nuevas escrituras, todas las réplicas de un dato convergerán al mismo valor. Este modelo es ideal para aplicaciones donde la disponibilidad y la tolerancia a fallos son prioritarias, y donde una breve inconsistencia temporal es aceptable.

Para gestionar la reconciliación de datos, Cassandra utiliza una técnica llamada "read repair". Durante una operación de lectura, si se detecta una discrepancia entre las réplicas, el nodo coordinador actualiza las

réplicas obsoletas con el valor más reciente. Además, un proceso de "anti-entropy" se ejecuta periódicamente para sincronizar las réplicas en segundo plano, asegurando la convergencia de los datos a lo largo del tiempo.

Particionamiento y replicación

Cassandra utiliza un particionamiento basado en hashes para distribuir los datos. Cada clave se pasa a través de una función hash para determinar su partición correspondiente, que se asigna a un nodo específico en el anillo lógico. Esta técnica asegura una distribución uniforme de los datos, evitando hotspots y permitiendo una escalabilidad horizontal eficiente.

La replicación en Cassandra se configura a nivel de keyspace, donde se define el factor de replicación (número de réplicas por dato). Cassandra soporta diferentes estrategias de replicación, como la replicación simple y la replicación en múltiples centros de datos. Durante una operación de escritura, el nodo coordinador envía la actualización a todas las réplicas correspondientes. La operación se considera exitosa una vez que un número configurable de réplicas ha confirmado la escritura, permitiendo un equilibrio entre consistencia y disponibilidad.

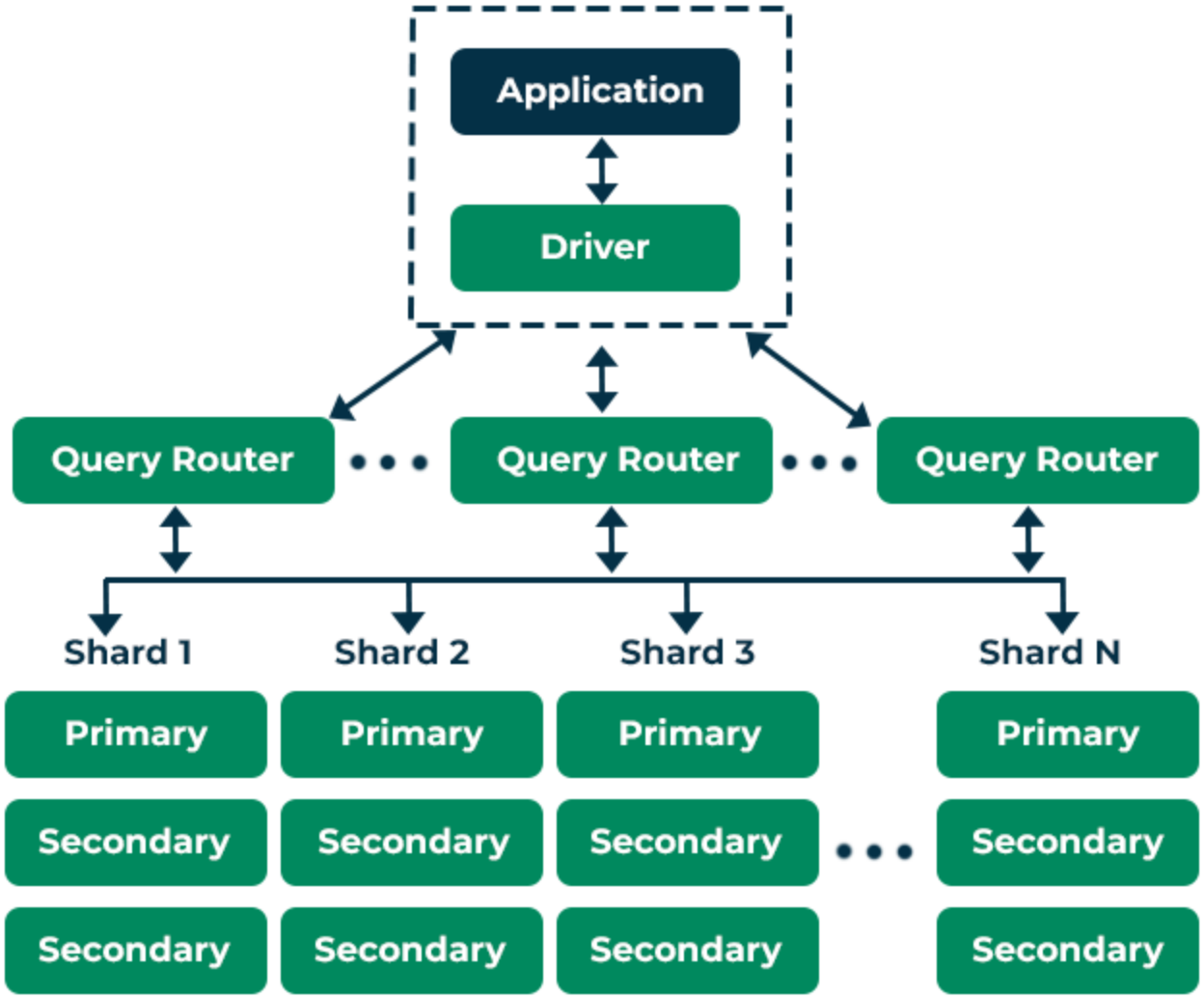
MongoDB

Modelo de datos orientado a documentos

MongoDB utiliza un modelo de datos orientado a documentos, donde los datos se almacenan en documentos BSON (Binary JSON). Cada documento es una estructura de datos flexible que puede contener campos anidados y arrays, permitiendo una representación rica y compleja de la información. Este modelo es ideal para aplicaciones que requieren flexibilidad en el esquema de datos, como aplicaciones web y móviles.

Los documentos se agrupan en colecciones, y cada colección puede tener documentos con diferentes estructuras, proporcionando una flexibilidad adicional. MongoDB permite consultas ad hoc, indexación y agregaciones avanzadas, lo que facilita la gestión y análisis de datos.

La arquitectura de mongoDB en una configuración de sharding.



A continuación, se detalla cada componente de esta arquitectura:

1. Application (aplicación):
    - Es la capa en la que se ejecuta el software o la aplicación que interactúa con la base de datos MongoDB.
2. Driver (controlador):
    - El controlador es una biblioteca de software que permite a la aplicación comunicarse con la base de datos MongoDB. Proporciona una API que facilita las operaciones CRUD (Crear, Leer, Actualizar y Eliminar).
3. Query Router (enrutador de consultas):
    - Los enrutadores de consultas, también conocidos como mongos, son los encargados de dirigir las consultas y operaciones de la aplicación a los shards correspondientes. Actúan como intermediarios que distribuyen las operaciones a los nodos de la base de datos de acuerdo con la clave de partición.
4. Shards:

- Los shards son fragmentos individuales de la base de datos distribuidos en diferentes servidores. Cada shard contiene un subconjunto de los datos y puede ser responsable de una o más colecciones.
- En la imagen, se presentan múltiples shards (`Shard 1`, `Shard 2`, `Shard 3`, ... `Shard N`).

5. Primary (Primario) y Secondary (Secundario):

    - Cada shard contiene un nodo primario y varios nodos secundarios.
    - Primary (primario): Es el nodo que recibe todas las operaciones de escritura y lectura. Los datos se replican desde el nodo primario a los nodos secundarios.
    - Secondary (secundario): Son nodos que mantienen copias del conjunto de datos del nodo primario. Los nodos secundarios pueden atender operaciones de lectura para equilibrar la carga y proporcionar redundancia para la alta disponibilidad.

Funcionamiento general:

1. La aplicación envía solicitudes a MongoDB a través del driver.
2. El driver envía las solicitudes a los enrutadores de consultas (query routers).
3. Los enrutadores de consultas determinan a qué shard enviar cada solicitud basándose en la clave de partición.
4. Cada shard tiene un nodo primario y varios nodos secundarios, proporcionando replicación de datos y alta disponibilidad.

Este diseño de sharding permite a MongoDB manejar grandes volúmenes de datos y operaciones distribuyendo la carga entre múltiples servidores y asegurando la disponibilidad y redundancia de los datos.

Sharding y balanceo de carga

MongoDB implementa sharding para distribuir datos y carga de trabajo en múltiples servidores. El sharding se basa en una clave de shard, que se utiliza para determinar cómo se distribuyen los documentos entre los shards. MongoDB soporta diferentes estrategias de particionamiento, como particionamiento de rango y particionamiento de hash, cada una adecuada para diferentes tipos de cargas de trabajo.

El balanceo de carga en MongoDB es gestionado por un componente llamado mongos, que actúa como un enrutador y distribuye las operaciones de lectura y escritura entre los shards. MongoDB monitorea continuamente la carga en cada shard y migra los datos entre los shards para mantener un balance equilibrado. Este proceso de migración es transparente para las aplicaciones, asegurando una distribución uniforme de los datos y minimizando los hotspots.

Replicación y alta disponibilidad

MongoDB utiliza conjuntos de réplicas para garantizar la alta disponibilidad y la tolerancia a fallos. Un conjunto de réplicas consiste en un nodo primario y uno o más nodos secundarios. El nodo primario maneja todas las operaciones de escritura, mientras que los nodos secundarios replican los datos del primario y pueden manejar operaciones de lectura, mejorando el rendimiento y la disponibilidad.

En caso de fallo del nodo primario, los nodos secundarios realizan una elección automática para seleccionar un nuevo primario, asegurando una mínima interrupción del servicio. MongoDB soporta diferentes niveles de consistencia de lectura, desde lecturas estrictamente consistentes desde el nodo primario hasta lecturas eventual y paralelamente consistentes desde los nodos secundarios.

Ejercicios

Teorema CAP:

- Pregunta: Explica cómo el Teorema CAP influye en el diseño de una base de datos distribuida y proporcione ejemplos de sistemas que priorizan diferentes pares de propiedades (C, A, P).
- Ejercicio: Diseña un sistema distribuido teórico para una aplicación bancaria que debe priorizar la consistencia y la disponibilidad, explicando cómo manejaría las particiones de red.
- Pregunta: Explica cómo el Teorema CAP impacta el diseño de sistemas de almacenamiento distribuidos modernos, como Apache Cassandra y MongoDB.
- Ejercicio: Diseña un sistema de mensajería instantánea que debe operar con alta disponibilidad y tolerancia a particiones, explicando cómo manejaría los compromisos entre consistencia y disponibilidad.

Modelos de consistencia:

- Pregunta: Compara y contraste la consistencia fuerte y la consistencia eventual en el contexto de bases de datos distribuidas.
- Ejercicio: Desarrolla un algoritmo que implemente consistencia causal utilizando relojes vectoriales y describa cómo maneja las escrituras concurrentes.
- Pregunta: Analiza cómo la consistencia eventual puede afectar la experiencia del usuario en una aplicación de redes sociales.
- Ejercicio: Diseña un sistema de consistencia fuerte para una aplicación de transacciones financieras, detallando los mecanismos para asegurar que todas las réplicas de datos estén siempre sincronizadas.

Control de concurrencia distribuida:

- Pregunta: Explica las diferencias entre los algoritmos de control de concurrencia optimista y pesimista, y discuta los escenarios en los que cada uno es más apropiado.
- Ejercicio: Diseña un protocolo de bloqueo distribuido que use relojes de Lamport para asegurar la serializabilidad de transacciones en un sistema de bases de datos distribuidas.
- Pregunta: Compara los beneficios y desventajas del control de concurrencia optimista frente al control de concurrencia pesimista en un entorno de base de datos distribuida.
- Ejercicio: Desarrolla un protocolo para controlar la concurrencia en una base de datos distribuida utilizando relojes vectoriales, y explique cómo manejaría los conflictos de actualización.

Replicación de datos

Replicación Síncrona vs. Asíncrona:

- Pregunta: Analiza las ventajas y desventajas de la replicación síncrona y asíncrona en un sistema distribuido.
- Ejercicio: Proporciona un diseño para un sistema de replicación síncrona en una base de datos distribuida de e-commerce, explicando cómo manejaría la latencia y los fallos de red.
- Pregunta: Explica los escenarios donde la replicación síncrona es preferible a la replicación asíncrona y viceversa.
- Ejercicio: Diseña un sistema de replicación asíncrona para un servicio de streaming de video, detallando cómo garantizaría la consistencia eventual de los datos replicados.

Protocolos de replicación:

- Pregunta: Describa el funcionamiento del protocolo de consenso Raft y compárelo con Paxos.
- Ejercicio: Implementa un algoritmo simplificado de Raft en pseudocódigo y explique cómo manejaría la elección de líderes y la replicación de logs.
- Pregunta: Compara y contrasta los protocolos de consenso Raft y Paxos, centrándose en su uso en bases de datos distribuidas.
- Ejercicio: Implementa en pseudocódigo un protocolo de consenso simplificado basado en Paxos y explique cómo manejaría las fallas de nodo.

Particionamiento y sharding

Tipos de particionamiento:

- Pregunta: Explica los diferentes tipos de particionamiento (horizontal, vertical, de rango, de lista, de hash) y sus aplicaciones prácticas.
- Ejercicio: Diseña un esquema de particionamiento de rango para una base de datos de registros médicos y explique cómo manejaría la re-partición cuando los datos crezcan.
- Pregunta: Analiza las ventajas y desventajas del particionamiento horizontal frente al vertical en un sistema de bases de datos distribuida.
- Ejercicio: Diseña un esquema de particionamiento de hash para una base de datos de inventario de una gran cadena de tiendas, explicando cómo manejaría el balanceo de carga.

Estrategias de sharding y balanceo de carga:

- Pregunta: Discute las estrategias de sharding y sus implicaciones en el balanceo de carga y la consistencia de datos.
- Ejercicio: Diseña una estrategia de sharding para una red social que debe manejar millones de usuarios, explicando cómo garantizaría la consistencia y la disponibilidad de los datos.
- Pregunta: Discute cómo el sharding puede mejorar la escalabilidad de una aplicación web de alto tráfico.
- Ejercicio: Proporciona una estrategia de sharding para un servicio de correo electrónico distribuido, explicando cómo garantizaría la consistencia y la disponibilidad de los mensajes.

Tecnologías y herramientas

Bases de datos NoSQL: Apache Cassandra y MongoDB:

- Pregunta: Compara la arquitectura de Apache Cassandra y MongoDB, centrándose en sus modelos de datos y mecanismos de replicación.
- Ejercicio: Diseña un sistema utilizando Apache Cassandra para una aplicación de análisis de big data, explicando cómo manejaría la partición y replicación de los datos.
- Pregunta: Discute cómo el sharding puede mejorar la escalabilidad de una aplicación web de alto tráfico.
- Ejercicio: Proporciona una estrategia de sharding para un servicio de correo electrónico distribuido, explicando cómo garantizaría la consistencia y la disponibilidad de los mensajes.

In [ ]:
```
### Tus respuestas
```

Ejercicio: Implementa una simulación en Python que ilustre el Teorema CAP. Crea un sistema distribuido simple con nodos que pueden experimentar fallos de red (particiones) y muestra cómo afecta la consistencia y la disponibilidad. Incluye opciones para configurar el sistema en modos CP, AP y CA, y demuestra el comportamiento bajo diferentes escenarios de fallos.

In [ ]:
```
## Tu respuesta
```

Ejercicio: Implementa un sistema de consistencia eventual en Python utilizando hilos (threads). Simula varios nodos que escriben y leen datos, y muestra cómo los datos eventualmente convergen en todos los nodos.

In [ ]:
```
## Tu respuesta
```

Ejercicio: Implementa una base de datos distribuida simple en Python que soporte replicación síncrona y asíncrona. Incluye un mecanismo para configurar el tipo de replicación y muestra cómo se comportan las escrituras y lecturas en cada modo.

In [ ]:
```
## Tu respuesta
```

Ejercicio: Implementa una base de datos particionada utilizando particionamiento de hash en Python. Crea un sistema que distribuya las claves entre varias particiones y proporciona operaciones de escritura y lectura.

In [ ]:
```
## Tu respuesta
```

Ejercicio: Usando la biblioteca pymongo, escribe un script en Python que se conecte a una base de datos MongoDB, cree una colección con índices, y realice operaciones de inserción y consulta.

In [ ]:
```
## Tu respuesta
```