Big Data Aplicado

Almacenamiento distribuido

Contenido

Almacenamiento distribuido

Características principales

El almacenamiento distribuido es una técnica fundamental para el manejo de grandes volúmenes de datos (Big Data), ya que permite guardar la información a través de múltiples nodos (ordenadores) interconectados en una red. Esta distribución ofrece una serie de ventajas clave:

- **Escalabilidad:** Los sistemas de almacenamiento distribuido pueden crecer horizontalmente agregando más nodos al clúster. Esto permite manejar el crecimiento exponencial de datos que se produce en muchos entornos de Big Data.
- **Tolerancia a fallos:** La información se replica en múltiples nodos, lo que garantiza la disponibilidad de los datos incluso si un nodo falla.
- **Alto rendimiento:** Los datos se distribuyen entre los nodos donde se van a procesar, lo que reduce la necesidad de transferir grandes cantidades de datos a través de la red.
- **Reducción de costes:** Se pueden utilizar discos duros de bajo coste ("commodity hardware") para construir el sistema de almacenamiento.

Existen varios tipos de almacenamiento distribuido, incluyendo:

- **Sistemas de archivos distribuidos:** Estos sistemas dividen los archivos en bloques y los distribuyen entre los nodos del clúster. Un ejemplo destacado es HDFS (Hadoop Distributed File System).
- **Bases de datos NoSQL:** Estas bases de datos ofrecen una alternativa a las bases de datos relacionales tradicionales y están diseñadas para manejar grandes volúmenes de datos no estructurados o semiestructurados. Existen diferentes tipos de bases de datos NoSQL, como las orientadas a documentos, las orientadas a grafos y las orientadas a columnas.
- **Almacenamiento en la nube:** Los servicios de almacenamiento en la nube, como Amazon S3 o Microsoft Azure, ofrecen una solución escalable y flexible para almacenar grandes volúmenes de datos.

HDFS: Un sistema de archivos distribuido para Big Data

HDFS es uno de los sistemas de archivos distribuidos más utilizados en el contexto de Big Data. Sus características principales incluyen:

- **Arquitectura Master-Slave:** HDFS utiliza una arquitectura Master-Slave, donde un nodo **NameNode** gestiona el sistema de archivos y los nodos **DataNode** almacenan los bloques de datos.
- **Replicación de datos:** Los bloques de datos se replican en múltiples DataNode para garantizar la tolerancia a fallos.
- **Bloques de gran tamaño:** HDFS utiliza bloques de datos de gran tamaño (64 MB o 128 MB por defecto), lo que optimiza el acceso secuencial a grandes archivos.
- **Optimización para la localidad de datos:** HDFS intenta almacenar los bloques de datos en los nodos donde se van a procesar, lo que reduce el tráfico de red.

Ventajas del almacenamiento distribuido

El almacenamiento distribuido ofrece una serie de ventajas clave para el procesamiento de Big Data:

- **Escalabilidad y flexibilidad:** Los sistemas de almacenamiento distribuido pueden crecer horizontalmente para manejar el crecimiento de datos. Además, ofrecen flexibilidad para adaptarse a diferentes tipos de datos y cargas de trabajo.
- **Alta disponibilidad y tolerancia a fallos:** La replicación de datos en múltiples nodos garantiza la disponibilidad de la información incluso si un nodo falla.
- **Rendimiento optimizado:** La distribución de datos y el procesamiento en paralelo permiten un rendimiento optimizado para aplicaciones de Big Data.

Desafios del almacenamiento distribuido

A pesar de sus ventajas, el almacenamiento distribuido también presenta algunos desafios:

- **Complejidad de gestión:** La gestión de un sistema de almacenamiento distribuido puede ser compleja, especialmente a medida que el clúster crece.
- **Consistencia de datos:** Garantizar la consistencia de datos en un sistema distribuido puede ser complicado, especialmente en entornos donde se producen muchas escrituras concurrentes.

- **Seguridad:** La seguridad de los datos es un aspecto importante a considerar en un sistema distribuido, ya que la información se encuentra dispersa en múltiples nodos.

Replicación, Compartición, Paralelismo

Los sistemas de ficheros distribuidos (DFS) son una tecnología fundamental para el almacenamiento y procesamiento de grandes volúmenes de datos, ya que permiten distribuir la información en múltiples nodos interconectados en red. Esta distribución facilita la escalabilidad, tolerancia a fallos y un alto rendimiento. Tres conceptos claves en el funcionamiento de los DFS son la replicación, la compartición y el paralelismo.

Replicación: Garantizando la Disponibilidad y Tolerancia a Fallos

La **replicación** consiste en crear copias de los bloques de datos y distribuirlas entre los diferentes nodos del clúster. Este mecanismo es fundamental para garantizar la **tolerancia a fallos**, ya que en caso de que un nodo falle, los datos seguirán disponibles en otros nodos. Además, la replicación puede mejorar el **rendimiento** del sistema al permitir que las lecturas se realicen desde el nodo más cercano al cliente.

En HDFS, el **NameNode** es el responsable de gestionar la replicación de datos. Al crear un archivo, se especifica un **factor de replicación** que determina cuántas copias del archivo se crearán. Por defecto, HDFS utiliza un factor de replicación de 3, lo que significa que se crearán tres copias de cada bloque de datos.

La ubicación de las réplicas se determina mediante una **política de ubicación de réplicas**. HDFS utiliza una política de ubicación de réplicas que tiene en cuenta la ubicación de los nodos en la red, con el objetivo de optimizar la tolerancia a fallos y el rendimiento. Por ejemplo, las réplicas se suelen almacenar en diferentes racks para que un fallo en un rack no afecte a la disponibilidad de los datos.

Compartición: Un Espacio de Nombres Global

Los sistemas de ficheros distribuidos proporcionan un **espacio de nombres global** que permite a los usuarios acceder a los datos de forma transparente, independientemente de su ubicación física. Esto significa que los usuarios pueden acceder a los archivos como si estuvieran almacenados en un único sistema de archivos, sin tener que preocuparse por la distribución física de los datos.

En HDFS, el NameNode es el responsable de mantener el espacio de nombres global. El NameNode almacena la información sobre los archivos, los bloques de

datos y su ubicación en los DataNode. Los clientes acceden a los datos a través del NameNode, que les proporciona la información necesaria para acceder a los bloques de datos en los DataNode.

Paralelismo: Aprovechando la Potencia de Múltiples Nodos

El **paralelismo** es una de las principales ventajas de los sistemas de ficheros distribuidos. Al distribuir los datos en múltiples nodos, las operaciones de lectura y escritura se pueden realizar en paralelo, lo que aumenta significativamente el rendimiento del sistema.

En HDFS, el paralelismo se utiliza en diversas operaciones, como la creación de archivos, la lectura de archivos y el procesamiento de datos. Por ejemplo, al crear un archivo, los bloques de datos se escriben en los DataNode en paralelo. Del mismo modo, al leer un archivo, los bloques de datos se pueden leer desde los DataNode en paralelo.

Ejemplos de Replicación, Compartición y Paralelismo en HDFS:

- **Replicación:** Cuando un usuario almacena un archivo en HDFS, el sistema crea tres copias (por defecto) del archivo y las distribuye en diferentes nodos del clúster. Si un nodo falla, el usuario aún puede acceder al archivo desde las otras copias.
- **Compartición:** Múltiples usuarios pueden acceder al mismo archivo en HDFS de forma concurrente. El sistema se encarga de gestionar el acceso concurrente y garantizar que los datos se mantengan consistentes.
- **Paralelismo:** Una aplicación de procesamiento de datos puede leer un archivo grande de HDFS en paralelo desde múltiples nodos, lo que acelera significativamente el procesamiento.

Conclusiones

La replicación, la compartición y el paralelismo son conceptos fundamentales en el funcionamiento de los sistemas de ficheros distribuidos. Estos mecanismos permiten a los DFS ofrecer escalabilidad, tolerancia a fallos y un alto rendimiento, lo que los convierte en una tecnología esencial para el almacenamiento y procesamiento de grandes volúmenes de datos.

Bases de datos distribuidas

Conceptos Clave: Replicación, Partición y Transacciones

Las bases de datos distribuidas son esenciales en el manejo de grandes volúmenes de datos, especialmente en el contexto del Big Data. Para comprender su funcionamiento, es crucial entender los conceptos de replicación, partición y transacciones.

Replicación: Aumentando la Disponibilidad y Tolerancia a Fallos

La replicación implica la creación de copias de los datos y su distribución a través de múltiples nodos en un clúster. Este proceso ofrece dos ventajas principales:

- **Alta Disponibilidad:** Incluso si un nodo falla, los demás nodos que contienen las réplicas pueden seguir sirviendo los datos, garantizando que el sistema se mantenga funcional.
- **Tolerancia a Fallos:** La redundancia que ofrece la replicación protege la información de pérdidas ante la falla de un nodo, ya que las réplicas en otros nodos pueden utilizarse para recuperar los datos.

En bases de datos NoSQL, la capacidad de replicar datos de forma bidireccional, incluyendo réplicas parciales basadas en criterios específicos, permite una distribución eficiente.

Partición: Escalabilidad Horizontal para Grandes Volúmenes de Datos

La partición, también conocida como **sharding**, consiste en dividir una gran base de datos en subconjuntos más pequeños llamados particiones. Este proceso permite distribuir las particiones a diferentes nodos, facilitando la escalabilidad horizontal. La partición es fundamental para gestionar el volumen masivo de datos en entornos de Big Data.

Las estrategias de partición varían según el tipo de datos y las necesidades del sistema. En bases de datos clave-valor, por ejemplo, la partición por hash es una técnica común. En este método, se utiliza una función de hash para asignar claves a diferentes particiones, distribuyendo los datos de forma uniforme.

Teorema CAP en Sistemas Distribuidos

El teorema CAP, también conocido como la conjetura de Brewer, es un concepto fundamental en el diseño de sistemas distribuidos, especialmente relevante en el contexto de Big Data. El teorema establece que en un sistema distribuido es imposible garantizar simultáneamente las tres propiedades siguientes: **consistencia (C), disponibilidad (A) y tolerancia a particiones (P)**.

Definición de las tres propiedades:

- **Consistencia (Consistency):** Todos los nodos del sistema ven los mismos datos al mismo tiempo. En un sistema consistente, una lectura siempre devuelve el valor más reciente escrito, independientemente del nodo al que se acceda.
- **Disponibilidad (Availability):** El sistema siempre está disponible para responder a las solicitudes de los clientes, incluso si algunos nodos fallan. Esto implica que el sistema debe seguir funcionando incluso ante fallos de hardware o de red.
- **Tolerancia a Particiones (Partition Tolerance):** El sistema puede seguir funcionando incluso si la red se divide en particiones, es decir, si los nodos no pueden comunicarse entre sí. Esta propiedad es crucial en entornos de Big Data donde la red puede ser grande y compleja.

El Dilema del Teorema CAP:

El teorema CAP establece que solo se pueden garantizar dos de estas tres propiedades a la vez. Esto significa que los diseñadores de sistemas distribuidos deben tomar decisiones sobre qué propiedades priorizar en función de las necesidades de su aplicación.

Ejemplos de elecciones:

- **Consistencia y Disponibilidad (CA):** Esta combinación es adecuada para sistemas donde la consistencia de datos es crítica, como los sistemas bancarios. Sin embargo, estos sistemas no son tolerantes a particiones y fallarán si la red se divide.
- **Consistencia y Tolerancia a Particiones (CP):** Esta opción es adecuada para sistemas donde la consistencia de datos es importante, pero se puede tolerar cierta indisponibilidad en caso de fallos de red. Un ejemplo son los sistemas de gestión de datos distribuidos como Apache Cassandra.
- **Disponibilidad y Tolerancia a Particiones (AP):** Esta combinación es adecuada para sistemas donde la disponibilidad es primordial, como los sistemas de redes sociales. En estos sistemas, se puede sacrificar la consistencia de datos a corto plazo para garantizar que el sistema siga respondiendo a las solicitudes.

Limitaciones del Teorema CAP:

El teorema CAP ha sido objeto de algunas críticas por ser demasiado simplista y no reflejar la complejidad de los sistemas distribuidos modernos. Algunos autores

argumentan que el teorema solo considera una forma de consistencia (consistencia linealizable) y un tipo de fallo (particiones de red).

Alternativas al Teorema CAP:

Existen modelos alternativos para comprender los compromisos en sistemas distribuidos, como el teorema PACELC. Este teorema extiende el CAP al considerar la latencia (L) y la consistencia en caso de errores (E).

Conclusión:

El teorema CAP es una herramienta útil para comprender los compromisos en el diseño de sistemas distribuidos. Ayuda a los diseñadores a tomar decisiones informadas sobre qué propiedades priorizar en función de las necesidades de su aplicación. Sin embargo, es importante recordar que el teorema es una simplificación y no debe utilizarse como una regla absoluta.

Transacciones en bases de datos

El concepto de **transacciones** es fundamental en el ámbito de las bases de datos para garantizar la integridad y consistencia de los datos. Las transacciones son secuencias de operaciones que se ejecutan como una unidad indivisible, asegurando que todas las operaciones se completen con éxito o que, en caso de fallo, no se aplique ningún cambio. Este enfoque previene estados inconsistentes en la base de datos y protege la información ante posibles errores o interrupciones.

Propiedades ACID

Las transacciones son secuencias de operaciones que se ejecutan como una unidad indivisible, garantizando la **integridad** y **consistencia de los datos**. Para asegurar su fiabilidad, las transacciones deben cumplir las propiedades ACID:

- **Atomicidad:** Una transacción se trata como una unidad indivisible, o se ejecuta completamente o no se ejecuta en absoluto. Si ocurre un fallo durante la transacción, se deshacen todos los cambios realizados hasta ese punto para mantener la integridad de los datos. Esta propiedad es crucial para evitar estados inconsistentes en la base de datos, asegurando que los datos se mantengan válidos incluso ante errores o interrupciones.
- **Consistencia:** Una transacción lleva al sistema de una estado consistente a otro estado consistente. Esto significa que cualquier transacción debe cumplir con todas las reglas y restricciones definidas en la base de datos, manteniendo la integridad de los datos.

- **Aislamiento:** Las transacciones se ejecutan de forma aislada entre sí, evitando que los cambios intermedios de una transacción afecten a otras transacciones. Esta propiedad previene interferencias entre transacciones concurrentes, asegurando que cada transacción vea una vista consistente de los datos.
- **Durabilidad:** Una vez que la transacción se completa con éxito, los cambios son permanentes y persisten incluso en caso de fallos del sistema. Esta propiedad asegura que los cambios realizados por una transacción se almacenen de forma permanente y no se pierdan ante fallos del sistema.

Workspaces y Journals: Mecanismos para Garantizar ACID

Para implementar las propiedades ACID, las bases de datos suelen utilizar mecanismos como workspaces y journals:

- **Workspaces:** Un workspace es un área de memoria privada que se asigna a una transacción para realizar sus operaciones. Los cambios realizados por la transacción solo se hacen visibles a otras transacciones una vez que la transacción se confirma y se escriben los cambios en la base de datos. Este enfoque permite que las transacciones se ejecuten de forma aislada y previene que los cambios intermedios afecten a la consistencia de la base de datos. El uso de workspaces específicos para cada transacción incrementa la concurrencia en métodos optimistas, ya que las transacciones de lectura no se bloquean entre sí.
- **Journals:** Un journal es un registro persistente de todas las operaciones realizadas por las transacciones. Este registro se utiliza para recuperar la base de datos a un estado consistente en caso de fallo del sistema. Los journals registran las operaciones de forma secuencial, lo que permite reconstruir el estado de la base de datos hasta el punto del fallo y deshacer las transacciones incompletas. En HDFS, los cambios en el sistema de archivos se registran en archivos separados llamados EditLogs, análogos a un punto de control o breakpoints en un registro de transacciones de una base de datos relacional.

Transacciones Distribuidas: Desafíos y Soluciones

Las transacciones distribuidas, como su nombre indica, involucran operaciones en múltiples nodos de una base de datos distribuida. Este tipo de transacciones presenta desafíos adicionales para mantener la consistencia y la integridad de los datos, ya que las operaciones deben coordinarse a través de la red.

Uno de los protocolos más utilizados para implementar transacciones distribuidas es el **two-phase commit (2PC)**. Este protocolo divide la transacción en dos fases: una fase de preparación, donde cada nodo participante vota si está listo para confirmar la transacción, y una fase de confirmación, donde se confirma o se aborta la transacción en todos los nodos participantes.

Aunque 2PC es un protocolo robusto, presenta algunas limitaciones, como el riesgo de bloqueos si el coordinador de la transacción falla. Además, el rendimiento de 2PC puede verse afectado por la latencia de la red y la necesidad de coordinar múltiples nodos.

En la práctica, las transacciones distribuidas pueden ser complejas de implementar y gestionar, especialmente en entornos de Big Data con grandes volúmenes de datos y nodos distribuidos. Algunos sistemas NoSQL relajan las restricciones de ACID para mejorar la escalabilidad y el rendimiento, adoptando modelos de consistencia eventual.

Sin embargo, en entornos de Big Data, donde la escalabilidad y disponibilidad son prioritarias, algunos sistemas NoSQL relajan el cumplimiento estricto de ACID, adoptando modelos como BASE (Basically Available, Soft state, Eventually consistent).

BASE

BASE, por otro lado, significa Básicamente Disponible, Estado Blando, Consistencia Eventual. Este modelo prioriza la disponibilidad y la tolerancia a particiones, sacrificando la consistencia inmediata en favor de un estado finalmente consistente. Las características principales de BASE son:

- **Básicamente Disponible:** El sistema garantiza la disponibilidad de los datos, incluso en caso de fallos parciales. La respuesta a una solicitud siempre estará disponible, aunque pueda no ser la más actualizada o consistente en ese momento.
- **Estado Blando:** El estado de los datos puede cambiar con el tiempo, incluso sin nuevas entradas, debido a la naturaleza de la consistencia eventual.
- **Consistencia Eventual:** El sistema converge a un estado consistente eventualmente, después de un período de tiempo. Esto significa que las actualizaciones pueden propagarse a través del sistema con cierto retraso, pero finalmente todos los nodos tendrán una vista consistente de los datos.

Mientras ACID es ideal para sistemas transaccionales que requieren una alta consistencia, BASE es más adecuado para sistemas distribuidos a gran escala,

como las aplicaciones Big Data, donde la alta disponibilidad y la tolerancia a particiones son primordiales.

Las bases de datos NoSQL, que a menudo se utilizan en sistemas Big Data, suelen seguir el modelo BASE para manejar las grandes cantidades de datos y las altas tasas de concurrencia. Esto permite un mejor rendimiento y escalabilidad en entornos distribuidos, a cambio de una consistencia relajada.

Es importante destacar que BASE no significa la ausencia total de consistencia, sino que la consistencia se logra de forma eventual, en lugar de inmediata. El tiempo que tarda el sistema en alcanzar un estado consistente depende de la implementación específica y de las características del sistema.

Etapas en el acceso a bases de datos distribuidas:

- Peticiones remotas
- Transacciones remotas
- Transacciones distribuidas
- Peticiones distribuidas

Etapas en el acceso a bases de datos distribuidas

Las bases de datos distribuidas presentan desafíos únicos en términos de acceso y gestión de datos. Para comprender mejor las etapas involucradas, se pueden clasificar las operaciones de acceso en cuatro categorías principales:

1. Peticiones Remotas: Accediendo a Datos en Nodos Distantes

Una petición remota se produce cuando un cliente necesita acceder a datos que residen en un nodo diferente al suyo. En este escenario, el cliente envía una solicitud al nodo remoto, que procesa la petición y devuelve los resultados al cliente.

Este proceso es similar al acceso a archivos remotos en un sistema de archivos distribuido como HDFS. Cuando un cliente solicita un archivo, el NameNode, que gestiona el espacio de nombres, proporciona la ubicación de los bloques de datos. El cliente entonces se conecta directamente a los DataNodes que contienen los bloques y recupera la información.

2. Transacciones Remotas: Ejecutando Transacciones en un Nodo Remoto

Una transacción remota implica la ejecución de una transacción completa en un solo nodo remoto. Aunque la transacción se inicia desde un cliente en un nodo diferente, todas las operaciones de la transacción se realizan en el nodo remoto seleccionado.

Este tipo de transacción es más simple de gestionar que una transacción distribuida, ya que no requiere la coordinación entre múltiples nodos. Sin embargo, la responsabilidad de garantizar las propiedades ACID recae completamente en el nodo remoto.

3. Transacciones Distribuidas: Coordinación Compleja a través de Múltiples Nodos

Las transacciones distribuidas son las más complejas, ya que involucran operaciones en múltiples nodos de una base de datos distribuida. Para mantener la consistencia e integridad de los datos, este tipo de transacciones requiere una cuidadosa coordinación entre los nodos participantes.

El protocolo two-phase commit (2PC) es un método común para implementar transacciones distribuidas. En 2PC, un coordinador de transacciones gestiona la transacción, comunicándose con los nodos participantes para preparar y confirmar la transacción.

Las transacciones distribuidas son esenciales para mantener la consistencia de los datos en sistemas distribuidos, pero pueden afectar al rendimiento debido a la complejidad de la coordinación y la latencia de la red.

4. Peticiones Distribuidas: Procesamiento Paralelo para Consultas Complejas

Las peticiones distribuidas involucran la ejecución de una consulta en múltiples nodos de una base de datos distribuida. Este enfoque se utiliza para procesar grandes conjuntos de datos de forma paralela, dividiendo la consulta en subconsultas que se ejecutan en diferentes nodos.

Sistemas como MapReduce y Spark son ejemplos de plataformas que facilitan la ejecución de peticiones distribuidas. Estos sistemas dividen los datos en particiones, distribuyen las tareas de procesamiento a diferentes nodos y luego combinan los resultados parciales para obtener la respuesta final.

Las peticiones distribuidas son esenciales para el análisis de grandes volúmenes de datos en entornos de Big Data, ya que permiten aprovechar el paralelismo y la escalabilidad de los sistemas distribuidos.

Tablas distribuidas

Las tablas distribuidas son un elemento clave en las bases de datos distribuidas, donde los datos se almacenan y gestionan en múltiples nodos. Para optimizar el rendimiento, la escalabilidad y la disponibilidad de los datos, existen diferentes estrategias para dividir y replicar las tablas:

1. Divisiones Horizontales de Tabla: Segmentación de Datos en Filas

En la división horizontal, también conocida como fragmentación horizontal o ***sharding***, los datos de una tabla se dividen en subconjuntos de **filas** que se almacenan en diferentes nodos. Esta técnica es especialmente útil para distribuir grandes conjuntos de datos y mejorar la **escalabilidad**, ya que cada nodo solo gestiona una parte de los datos totales.

Por ejemplo, una tabla de clientes podría dividirse horizontalmente por regiones geográficas, almacenando los clientes de cada región en un nodo diferente.

Las divisiones horizontales permiten:

- **Paralelizar las operaciones**: Las consultas y actualizaciones pueden ejecutarse simultáneamente en diferentes nodos, lo que mejora el rendimiento general.
- **Escalar el sistema**: Se pueden agregar nuevos nodos al sistema para manejar el crecimiento del volumen de datos sin afectar el rendimiento.
- **Mejorar la disponibilidad**: Si un nodo falla, solo se afecta la parte de los datos almacenada en ese nodo, mientras que el resto del sistema permanece operativo.

2. Divisiones Verticales de Tabla: Separación de Datos en Columnas

En la división vertical, los datos de una tabla se dividen en subconjuntos de **columnas** que se almacenan en diferentes nodos. Esta técnica es útil cuando las tablas contienen un gran número de columnas y las aplicaciones solo acceden a un subconjunto de ellas.

Por ejemplo, una tabla de empleados con información personal, historial laboral y datos financieros podría dividirse verticalmente, almacenando cada tipo de información en un nodo diferente.

Las divisiones verticales permiten:

- **Optimizar el acceso a datos**: Las aplicaciones pueden acceder a los datos relevantes sin tener que leer columnas innecesarias, lo que mejora el rendimiento de las consultas.
- **Reducir la contención**: Las operaciones de lectura y escritura en diferentes columnas se pueden realizar en paralelo, lo que reduce la contención en los nodos.

- **Simplificar la gestión de datos**: Las diferentes partes de la tabla pueden gestionarse de forma independiente, lo que facilita tareas como el respaldo y la recuperación.

3. Tablas Reflejadas: Copias Idénticas para Alta Disponibilidad

Las tablas reflejadas son copias idénticas de una tabla que se almacenan en diferentes nodos. Esta técnica se utiliza para mejorar la **alta disponibilidad** y la **tolerancia a fallos**. Si un nodo falla, las aplicaciones pueden acceder a la copia de la tabla en otro nodo sin interrupción del servicio.

Las tablas reflejadas son útiles para:

- **Garantizar la disponibilidad de datos**: Las aplicaciones críticas pueden acceder a los datos incluso si uno o más nodos fallan.
- **Mejorar el rendimiento de lectura**: Las consultas se pueden distribuir entre las diferentes copias de la tabla, lo que mejora el rendimiento general.
- **Simplificar la recuperación ante desastres**: La pérdida de un nodo no implica la pérdida de datos, ya que existen copias en otros nodos.

Las tablas reflejadas pueden implementarse utilizando diferentes técnicas de replicación, como la replicación síncrona o asíncrona. La elección de la técnica depende de los requisitos de consistencia y rendimiento del sistema.