Lectura 8-9: Modelos de consistencia

Diego Granados Retana 2022158363

Bases de datos II

4 de noviembre, 2023

Responda las siguientes preguntas:

Explique la diferencia entre modelos de consistencia data-centric y client-centric

La consistencia data-centric es la que se analiza desde el punto de vista de la réplica mientras que la consistencia client-centric es la que se analiza desde el punto de vista del cliente. Con la consistencia data-centric, existen dos dimensiones. La primera es de los modelos para especificar consistencia. Estos describen los modelos que miden y especifican los niveles de consistencia que son tolerables para la aplicación. La segunda es de los modelos del ordenamiento de operaciones consistente. Estos especifican qué orden de operaciones se llevó a cabo y que están en las réplicas. Del lado de los modelos client-centric, hay otras dos dimensiones. La primera es la de consistencia eventual, que dice que todas las réplicas van a volverse gradualmente consistentes si no ocurre una actualización. La segunda dimensión, la garantía de consistencia del cliente, define que cada proceso del cliente necesita asegurar algún nivel de consistencia cuando se accede un dato en diferentes réplicas.

Comente similitudes y diferencias entre los modelos de consistencia de Redis y Cassandra

Redis es una base de datos en memoria y key-value, mientras que Cassandra es una base de datos columna-oriented. Redis optimiza los datos en memoria y se enfoca en el alto rendimiento. Si se trabaja con una sola instancia, es de consistencia fuerte, pero en un clúster es de consistencia eventual cuando se lee de las réplicas. Cassandra se enfoca en sistema distribuidos y prioriza la alta disponibilidad con particiones de red. Cassandra se puede configurar para tener consistencia fuerte en particiones de red. Utiliza un timestamp en cada columna, con el fin de resolver conflictos. Redis distribuye llaves en 16384 espacios en un hash. Se pueden alocar más de una llave al mismo espacio con Hash tags. Redis tiene un clúster donde cada nodo tiene conexiones de entrada y salida con los otros nodos. Funciona con un modelo maestro-esclavo donde la aplicación que hizo la solicitud de un dato es redireccionada al nodo que lo tiene. El maestro asíncronamente propaga los cambios a las réplicas y le da la confirmación al cliente, aún cuando no se haya asegurado la replicación. Por lo tanto, si el nodo maestro muere antes de replicarse y luego de dar la confirmación al cliente, se pierde la información. Cuando se particiona una red, si el nodo maestro dura mucho en responder, se escoge uno nuevo entre las réplicas y el nodo maestro anterior para de responder solicitudes de escritura, por lo que Redis no es la mejor opción para tener alta disponibilidad en particiones de la red. También, si en la partición una réplica llega a ser inalcanzable, se migra una réplica para que se convierta en la que se perdió. Finalmente, existe la posibilidad de habilitar que se pueda leer de nodos réplica, pero esto puede causar consistencia eventual. Cassandra también utiliza un clúster con nodos, donde cada uno puede responder a consultas de un cliente. Un nodo se vuelve el coordinador de una consulta y es el responsable de pedir los datos a los otros nodos y responder el resultado. Se particionan los datos con un hash de la llave de las filas y los datos tienen a quedar distribuidos equitativamente por el todo el clúster. Para replicarse y ser altamente disponible, Cassandra utiliza una estrategia y un factor. La estrategia determina en cuáles nodos se ponen

réplicas y el factor determina cuántos nodos tienen los mismos datos. Los datos se replican en sentido de las manecillas de reloj y los datos se copian a la cantidad de vecinos de acuerdo con el factor. Esto forma un tipo de anillo. Cassandra inicialmente tiene consistencia eventual, alta disponibilidad y latencia baja, pero se pueden configurar los niveles de consistencia de escritura y lectura:

- ALL: Los datos se replican en todas las réplicas antes de dar la confirmación. El dato leído se retorna luego que todas las réplicas hayan respondido.
- QUORUM: Los datos se escriben en un número de réplica llamado quórum antes de dar la confirmación. El dato leído se retorna cuando el quórum de réplicas ha respondido.
- ONE: Los datos se copian en al menos una réplica. El dato leído se retorna de la réplica más cercana. Por predeterminado, Cassandra utiliza una configuración donde se actualizan todos los nodos que han sido consultados a que tengan el valor más actualizado. Para resumir las diferencias, Cassandra utiliza un nodo coordinador mientras que Redis utiliza un nodo maestro. Redis no puede garantizar la consistencia en la escritura siempre mientras que Cassandra sí puede con su configuración de ALL. Redis es peor para los eventos de partición de red. Redis permite la consistencia causal mientras que Cassandra permite la consistencia fuerte. Redis tiene alta disponibilidad y la menor latencia, ya que es en memoria mientras que Cassandra es más barata, robusta, tiene alta disponibilidad y latencia un poco más alta. Redis y Cassandra también tiene aspectos en común. El modo de configuración de Cassandra de QUORUM se asemeja más al modelo de Redis, ya que ambos confirman la solicitud antes de asegurar que se haya propagado por todas las réplicas. Ambos Cassandra y Redis permiten la consistencia eventual.

Comente como afecta el rendimiento y funcionamiento de una base de datos los siguientes modelos de consistencia:

Strong Consistency

Este modelo es el que hace la base de datos más lenta, ya que se necesita tomar en cuenta una gran cantidad de medidas, como el locking de los recursos compartidos, la sincronización de las bases de datos en sistemas distribuidos, mantener el log al día, hacer que las transacciones solo hagan commit cuando ya se haya terminado en todas las bases de datos, entre otros. No obstante, también es la base de datos más confiable, ya que uno tiene asegurado que la información va a ser la correcta. No obstante, esto se paga en el tiempo de respuesta y disponibilidad.

Weak Consistency

Este modelo es el que hace que la base de datos sea lo más rápida posible. Una lectura no garantiza el valor más actualizado. Este sistema es muy escalable porque no necesita tener más nodos y réplicas. Esto se ve reflejado en los tiempos de respuesta, ya que tiene que hacer ninguna o pocas validaciones de los datos.

Eventual Consistency

En este modelo, las réplicas se convergen a lo largo del tiempo para llegar al mismo estado en los datos. No obstante, el sistema sigue estando disponible, por lo que hay una ventana de inconsistencia donde se pueden leer datos viejos. Este modelo es más rápido que el de consistencia fuerte porque permite que se hagan consultas a réplicas con información inconsistente. Esto implica que el usuario no siempre tenga la información más nueva, pero por lo menos sí la va a poder acceder.

Causal Consistency

En este modelo, si un proceso actualiza un objeto, otros procesos que reconozcan la actualización del objeto van a acceder al nuevo dato. Es más consistente que la consistencia eventual. Como es más fuerte, esto implica que es más lento que consistencia eventual y tiene menor disponibilidad. La coordinación de los procesos que lean el dato actualizado debe ser lo que genera la complejidad, ya que se debe coordinar cuáles procesos tienen que trabajar con el dato y su orden.

Read-your-writes Consistency

Esta consistencia asegura que una réplica está actualizada lo suficiente para poder leer los cambios de una transacción. Se pueden utilizar los commits de las transacciones para saber que esos cambios ya se aplicaron. Las transacciones se aplican en serie, es decir, aisladas. Si un proceso actualiza un dato, siempre lo va a considerar y otros procesos lo considerarán después. Cuando un registro se actualiza, todo intento de lectura va a tener el valor leído. Esta consistencia debe tener un impacto en el rendimiento de la base de datos, ya que como las transacciones se aplican en serie, esto debe causar que no se puedan hacer concurrentemente y se genere un atraso.

Session Consistency

Si un proceso usa una sesión y realiza una solicitud, esa sesión va a poder leer los cambios de acuerdo con lo que tenía antes y lo que ha cambiado. Esto implica que hay alta disponibilidad y rendimiento, ya que no se tienen que esperar a que se liberen recursos y se sincronice. No obstante, no leerá los cambios de otras sesiones, por lo que se pierde consistencia.

Monotic Reads Consistency

Si un proceso lee un valor, todas las lecturas posteriores retornan el mismo valor o uno más nuevo, por lo que sirve un orden monótono. Al igual que session consistency, la monotonía de operaciones y consistencia no se garantiza entre procesos al mismo objeto, por lo que pueden ocurrir problemas de dirty reads. No obstante, como no se tienen que aplicar operaciones de otros procesos, son totalmente disponibles, incluso cuando hay una partición de red.

Monotic Writes Consistency

Este modelo requiere que una lectura en un objeto tiene que completarse antes de que el mismo proceso pueda escribirlo otra vez, por lo que las escrituras siguen un orden monótono. Como tiene que esperar a que termine la primera escritura, puede ser que se genere un retraso. Todos los procesos pueden ver la lectura en el objeto en su orden monótono. Las escrituras monótonas solo aplican en el mismo proceso, por lo que otro proceso podría escribir a la vez, causando problemas de lost update. Esto posiblemente genera una consistencia débil, pero un mejor rendimiento porque no se tienen que realizar una gran cantidad de validaciones.