5. La nube como base de datos

Ahora dirigimos nuestra atención a la arquitectura de datos y a cómo reformularla para crear barreras de entrada que protejan a los servicios descendentes de las interrupciones en los servicios ascendentes.

La lucha contra la gravedad de los datos

Cuando pensamos en arquitectura, tendemos a centrarnos en cómo organizamos y disponemos el código fuente para mantenerlo limpio y sin ciclos. Sin embargo, los datos de un sistema son, sin duda, su activo más valioso y, al mismo tiempo, su mayor obstáculo para el cambio. **Esto se debe a que los datos tienen gravedad. A medida que un sistema crece y evoluciona, también lo hacen sus datos y su estructura de datos.**

Debemos dedicar tanto esfuerzo —o más— a cómo organizamos y disponemos nuestros datos como al código fuente. De lo contrario, se volverá frágil, inflexible e imposible de mantener.

Demandas en competencia

Las bases de datos monolíticas han demostrado ser igualmente deficientes, porque la fuerza de su gravedad de datos frena la innovación.

El modelado relacional de datos nos enseña a normalizar los esquemas de base de datos hasta el tercer grado (es decir, la tercera forma normal) para que no haya datos duplicados. Esto ha contribuido a consolidar la creencia de que toda duplicación de datos es mala. En otras palabras, incentiva el uso compartido de tablas entre módulos.

Cada módulo que accede a una tabla compartida probablemente representa un actor diferente; cada uno impone demandas en competencia sobre la estructura del esquema de base de datos. Con el tiempo, el esquema crece y se deforma para dar cabida a los distintos requisitos.

Esto genera dependencias ocultas entre los módulos. Una mala comprensión de estas dependencias ocultas puede dar lugar a errores que provoquen interrupciones del sistema.

Capacidad insuficiente

Este escenario es habitual, porque poseer y operar una base de datos con todas las garantías necesarias no es tarea sencilla. Nuestros datos son nuestro activo más importante y debemos protegerlos. Sin embargo, fortalecer una base de datos para garantizar seguridad y redundancia requiere habilidad, y los administradores de bases de datos están sobrecargados de trabajo.

En cambio, hacemos que una sola base de datos soporte cada vez más aplicaciones y añadimos más y más capacidad vertical y horizontal. Nunca parece ser suficiente. Siempre hay capacidad insuficiente, por lo que seguimos añadiendo más y más.

Volúmenes ingobernables

Nuestras bases de datos monolíticas tienden a acumular datos y nunca soltarlos, por si acaso los necesitamos. Mientras tanto, las aplicaciones que crean y usan esos datos deben evolucionar para satisfacer nuevos requisitos. Esto afecta inevitablemente a los esquemas de datos, por lo que los datos antiguos deben convertirse. A medida que el volumen de datos crece, el esfuerzo de conversión se vuelve más difícil y costoso en tiempo.

Muy probablemente, las demandas en competencia sobre los datos han creado tantas interdependencias que no podemos archivar los datos antiguos. Así que tienen que quedarse donde están y, a medida que los datos crecen, se vuelve cada vez más difícil gobernarlos y manipularlos.

Se vuelven ingobernables. Es un efecto compuesto. No podemos archivarlos, así que debemos añadir más capacidad. Es difícil convertirlos, así que resistimos el cambio, y la fuerza de la gravedad de los datos crece cada vez más.

Adopción del ciclo de vida de los datos

Necesitamos fragmentar las bases de datos monolíticas para garantizar que la gravedad de nuestros datos no frene la innovación.

Para fragmentar bases de datos monolíticas, los enfoques y patrones más aplicables son la arquitectura del ciclo de vida de los datos y el patrón Backend for Frontend (BFF).

La Arquitectura del Ciclo de Vida de los Datos (Data Life Cycle Architecture, DLC) reconoce que, a lo largo de la vida de un dato, distintos actores interactúan con él en diferentes fases de su ciclo de vida. Estos actores son la fuente de las demandas en competencia sobre los datos. Siguiendo el Principio de Responsabilidad Única (SRP), DLC recomienda fragmentar la base de datos según las fases del ciclo de vida de los datos.

Esto es análogo al enfoque tradicional de separar una base de datos de Online Transaction Processing (OLTP) de una de Online Analytical Processing (OLAP).

Además de las diferencias funcionales, estos actores tienen distintos patrones de acceso y requisitos de almacenamiento. Esta es la oportunidad perfecta para aplicar la práctica de polyglot persistence, eligiendo la tecnología de base de datos más adecuada para cada fase del ciclo de vida de los datos.

En el Capítulo 2, Definición de límites y cesión del control, utilizamos la técnica de event storming para descubrir los eventos (es decir, los verbos) que un sistema emitirá, el orden relativo de esos eventos y los actores que los producen.

Con esta técnica, descubrimos de forma natural las fases del ciclo de vida de los datos, ya que identifica el orden en que los diferentes actores interactúan con los agregados de dominio.

Figure 5.1: Data life cycle phases

Fase de creación

En la fase de creación del ciclo de vida de los datos, queremos usar almacenes de datos y formatos de datos optimizados para la escritura. Para datos generados e integrados, podemos simplemente escribir eventos en un stream, como AWS Kinesis.

Para la entrada manual de datos, debemos usar un formato de datos normalizado para que los usuarios no tengan que ingresar datos duplicados. Las bases de datos de clave-valor, de documentos y relacionales, como AWS DynamoDB y AWS Aurora, son muy adecuadas para la entrada de datos.

Es importante que la base de datos seleccionada proporcione una característica de Change Data Capture (CDC) para que podamos implementar la variante database-first del patrón Event Sourcing.

Otra característica importante de la creación de datos es la velocidad a la que los creamos. Algunos datos los creamos con poca frecuencia, como los datos de referencia o datos maestros, mientras que los datos transaccionales los producimos de forma continua. Podemos referirnos a estos como datos lentos y datos rápidos, respectivamente.

Fase de uso

Creamos todos los datos con un propósito. Pero los actores que crean los datos y los que los usan son casi siempre diferentes. Los distintos actores tienen distintos requisitos, por lo que queremos separar estas preocupaciones en diferentes servicios autónomos.

Tradicionalmente nos referimos a las bases de datos que usamos para estos escenarios como bases de datos OLTP o almacenes de datos operacionales. Definimos explícitamente estos almacenes de datos para permitir que los actores realicen sus tareas de manera eficiente. Queremos aprovechar al máximo esta característica y optimizar estos almacenes de datos operacionales para sus escenarios específicos.

Los almacenes de clave-valor, los motores de búsqueda y el almacenamiento de objetos son una excelente opción para estos almacenes de datos operacionales.

Fase de análisis

En el Capítulo 2, Definición de límites y cesión del control, analizamos cómo los sistemas orientados a eventos incorporan de forma implícita analítica de negocio y observabilidad, porque los eventos representan hechos, y la analítica trabaja con hechos.

Todas las técnicas y tecnologías de analítica tradicionales siguen siendo aplicables. Por ejemplo, los data marts y los data warehouses pueden consumir eventos ascendentes y usarlos para poblar las tablas de hechos en sus esquemas de estrella.

Almacenar datos por columna crea efectivamente un índice por columna. Las bases de datos de series temporales, como InfluxDB y AWS Timestream, y los motores de búsqueda, como Elasticsearch, son alternativas al uso de bases de datos relacionales para la analítica.

Uno de los avances más significativos en la fase de análisis del ciclo de vida de los datos es el event lake. El event lake consume todos los eventos y almacena estos hechos a perpetuidad.

Esto nos permite diferir el compromiso sobre cómo queremos utilizar la tecnología de analítica. Por ejemplo, los equipos pueden experimentar con diferentes hipótesis reproduciendo eventos históricos del lake en un almacén de datos y validando si el enfoque produce pronósticos precisos.

Otro avance significativo es la analítica accionable con big data y machine learning. Los servicios consumen eventos de interés y los usan para entrenar modelos de machine learning. Los modelos son luego utilizados por los servicios descendentes para identificar condiciones, como en la detección de anomalías, y controlar el comportamiento del sistema.

Data mesh es un tema emergente en la fase de análisis. Su objetivo principal es eliminar el cuello de botella de los equipos centralizados de ciencia de datos y analítica, sobrecargados de trabajo, mediante la descentralización y federación de las actividades de preparación de datos de analítica y su disponibilidad.

En definitiva, la fase de análisis desempeña un papel importante en impulsar la innovación porque proporciona los conocimientos que ayudan a las organizaciones a aprender y adaptarse.

Fase de archivo

Tarde o temprano, el costo de mantener un dato fácilmente disponible se volverá más elevado que el valor que proporciona.

En la sección Datos ligeros, cubrimos la técnica de time-to-live. Es una técnica sencilla en la que cada servicio decide cuánto tiempo necesita usar un dato. Después de ese tiempo, la base de datos elimina el registro para que el almacén de datos pueda mantener un tamaño predecible y manejable.

Estos servicios no se preocupan por los requisitos de archivo, ya que los requisitos de archivo corresponden a actores diferentes. Creamos servicios de archivo descendentes que consumen eventos y archivan los datos de forma proactiva. En otras palabras, creamos los archivos continuamente a medida que creamos los datos. Esto libera a los servicios operacionales para eliminar datos cuando quieran. Los servicios de archivo gestionan los requisitos de gestión de registros en cuanto a qué datos debemos retener y durante cuánto tiempo.

A medida que los datos envejecen, pueden transicionarse a almacenamiento a largo plazo, como AWS Glacier. Normalmente organizamos estos archivos por actor y fecha.

Es probable que un archivo soporte la rehidratación de datos para que los servicios ascendentes puedan reconstituir los datos que han eliminado.

Inversión de la base de datos

Estamos acostumbrados a depender de la base de datos para hacer gran parte del trabajo pesado por nosotros, pero ya no tenemos una base de datos homogénea en el núcleo de nuestros sistemas. Ahora tenemos muchas bases de datos heterogéneas, y es nuestra responsabilidad hacer que todas funcionen juntas.

Debemos superar la preocupación de asumir esta responsabilidad desde la base de datos, porque tomar el control de la sincronización de datos es lo que lleva a escapar completamente de la gravedad de los datos. Abordemos esta preocupación mirando hacia adentro para ver cómo las bases de datos usan un transaction log para gestionar (es decir, sincronizar) los datos derivados.

El transaction log

En el corazón de una base de datos está el transaction log. Es la fuente de verdad para la información sobre cada sentencia de inserción, actualización y eliminación que se ha ejecutado contra una base de datos.

Por ejemplo, cuando ejecutamos una sentencia de actualización contra una tabla específica, la base de datos encuentra el registro deseado, aplica los cambios y luego almacena el estado anterior y el nuevo estado en el transaction log.

La solución es el event hub y el event lake, como cubrimos en el Capítulo 4, Confianza en los hechos y la consistencia eventual. Esto es exactamente lo que logran. Proporcionan la fuente de verdad para la información sobre cada comando que los servicios autónomos han realizado dentro del sistema.

Datos derivados

El transaction log es el corazón de la base de datos porque se usa para propagar datos hacia las demás partes de la base de datos.

Estas otras partes de la base de datos son las copias derivadas de los datos, como índices, vistas materializadas e instancias replicadas de la base de datos.

La base de datos usa el log para replicar la base de datos a otras instancias para failover y para crear más copias en forma de read replicas que soporten más y más lectores. Mantiene índices, que son copias de los datos organizadas en disco en diferentes órdenes de clasificación para soportar lecturas más rápidas. Calcula vistas materializadas, y cada una es otra copia de los datos que contiene los resultados precalculados de una consulta demasiado costosa de ejecutar una y otra vez en tiempo real.

La consecuencia inevitable es que las aplicaciones añaden su propia caché de lectura directa (read-through cache) para proteger a sus usuarios de esta competencia con otras aplicaciones. Esta es otra copia derivada de los datos. Sin embargo, **esta vez la copia no es gestionada por la base de datos sino por los desarrolladores de la aplicación.**

Disección del patrón CQRS

El patrón Command Query Responsibility Segregation (CQRS) proporciona la base para invertir nuestras bases de datos monolíticas y escapar de la gravedad de los datos.

Propone el uso de dos modelos de dominio diferentes: un modelo de comandos optimizado para escritura y un modelo de consultas optimizado para lectura.

Desafortunadamente, el patrón CQRS tiene reputación de hacer los sistemas más complejos. Esta preocupación no carece de fundamento; sin embargo, con las modificaciones adecuadas, esta segregación produce sistemas que son en realidad más directos y flexibles.

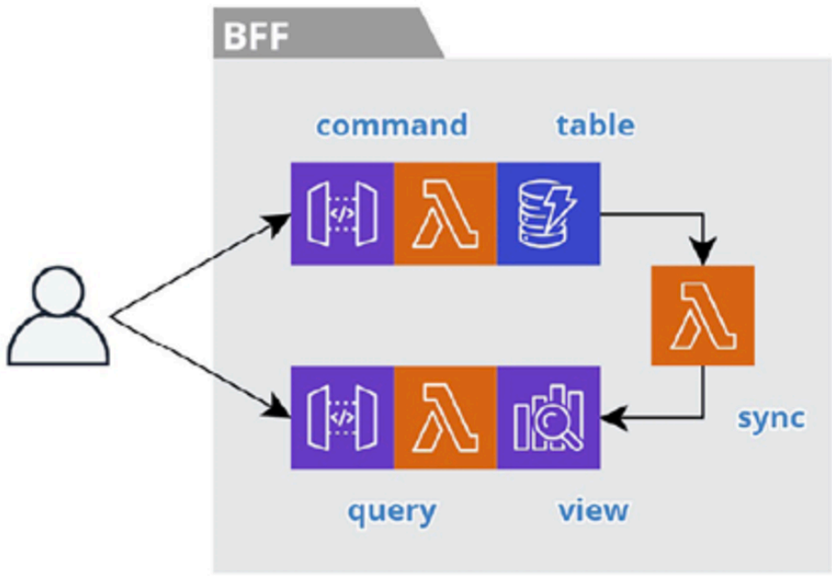

Figure 5.3: Traditional CQRS

Esto es obviamente más difícil de implementar porque hay más piezas. Hay una tabla usada por la función de comandos, una vista usada por la función de consultas, y una función de sincronización para sincronizar los datos de la tabla a la vista. **Si hay un beneficio claro en separar los modelos de dominio, como usar polyglot persistence, entonces puede valer la pena dedicar el esfuerzo adicional.**

CQRS a escala de sistema

Volviendo a la sección Adopción del ciclo de vida de los datos, vimos que a lo largo del ciclo de vida de los datos, diferentes actores usan los mismos datos lógicos con distintos requisitos de acceso y almacenamiento.

Propusimos usar las fases del ciclo de vida de los datos para definir los límites entre diferentes almacenes de datos. Como consecuencia, el patrón CQRS debe asumir un rol mucho más estratégico cuando adoptamos esta perspectiva holística.

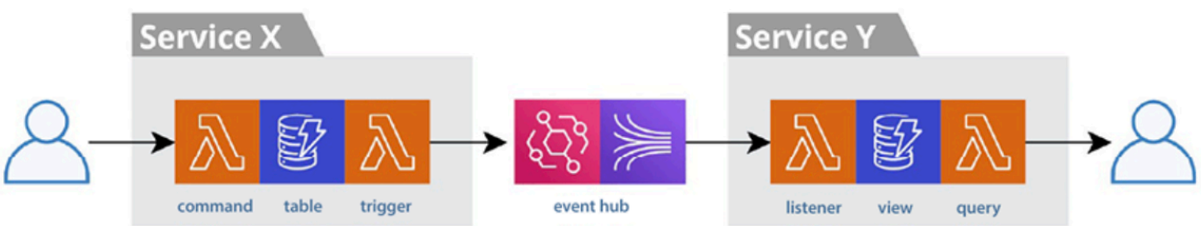

Figure 5.4: System-wide CQRS

El Servicio X se sitúa en la parte ascendente, apoyando al primer actor, y proporciona una función de comandos que crea datos en una tabla de su propiedad. La función trigger produce un evento de dominio al event hub, que proporciona el transaction log a escala de sistema que discutimos en la sección Inversión de la base de datos.

El Servicio Y se sitúa en la parte descendente, apoyando al siguiente actor, y proporciona una función listener que consume eventos y materializa los datos necesarios en una vista diseñada específicamente para soportar los escenarios de acceso de este actor descendente.

Además, la función de consultas y la vista proporcionan al actor descendente recursos dedicados para usar los datos y ayudar a garantizar una experiencia de usuario responsiva y resiliente.

Este último ejemplo muestra cómo el patrón CQRS es un mecanismo clave para crear una arquitectura que habilita el cambio.

Vistas materializadas

El trabajo principal de una vista materializada es mejorar la capacidad de respuesta de la experiencia del usuario. Lo logramos eligiendo el tipo de base de datos más adecuado, creando índices apropiados y luego precalculando joins y almacenando los datos en el formato que necesita el consumidor específico.

No hay razón para calcular el join una y otra vez. Podemos hacerlo una vez cuando recibimos los datos para que el consumidor no pague el precio en cada solicitud.

Barreras de entrada

Las vistas materializadas también realizan el trabajo igualmente importante de proporcionar resiliencia al sistema.

Los servicios que tienen dependencias directas de otros servicios son menos resilientes porque una interrupción en el servicio dependido impactará al servicio dependiente. Para contrarrestar este impulso, debemos fortalecer los límites entre servicios añadiendo barreras que controlen el radio de explosión cuando inevitablemente cometemos un error.

El patrón Circuit Breaker intenta resolver este problema manteniendo el paradigma de comunicación síncrona. Cuando el servicio dependiente detecta que está recibiendo demasiados errores del servicio dependido, el dependiente abrirá un circuito y dejará de llamar al servicio dependido durante un período de tiempo.

Esto resulta en una experiencia de usuario subóptima cuando hay una interrupción, añade sobrecarga a cada solicitud incluso cuando no hay ninguna interrupción, hace la experiencia del desarrollador más complicada y aumenta el costo total de propiedad. Pero para todos los demás escenarios, **el mejor enfoque es eliminar las dependencias eliminando la comunicación síncrona entre servicios** y aprovechando las vistas materializadas.

En el Capítulo 4, Confianza en los hechos y la consistencia eventual, aprendimos a implementar la comunicación asíncrona entre servicios y a usar un event hub como barrera de salida para proteger a los servicios ascendentes de las interrupciones en los servicios descendentes.

El patrón CQRS es empleado por los servicios descendentes para reaccionar a estos eventos y crear las vistas materializadas que actúan como barrera de entrada para proteger a los servicios descendentes de las interrupciones en los servicios ascendentes.

Cuando un servicio ascendente sufre una interrupción, no hay impacto en los servicios descendentes porque pueden continuar respondiendo a las consultas con los datos más recientes en sus vistas materializadas, en lugar de una respuesta predeterminada.

En última instancia, estas barreras hacen que los servicios sean más que resilientes; los hacen autónomos. Estas vistas materializadas son los recursos que los servicios necesitan para continuar operando cuando otros servicios experimentan problemas.

Caché en vivo

Otro beneficio del patrón CQRS es que las vistas materializadas actúan como una caché en vivo, lo que resuelve los problemas inherentes de una read-through cache.

Como discutimos anteriormente en la sección Inversión de la base de datos, los desarrolladores recurrirán en última instancia a añadir una capa de caché a su stack tecnológico para compensar la mayor latencia resultante de la competencia por recursos escasos de base de datos. Sin embargo, como hemos visto, **una read-through cache solo aborda los síntomas en lugar de la causa raíz, e introduce sus propios problemas también.**

El primer problema son los cache misses. Cuando falta un valor en la caché, necesitamos hacer una llamada adicional a la base de datos de origen para recuperar los datos, de ahí el nombre read-through. Luego añadimos los datos a la caché antes de devolverlos. **El caso extremo es cuando la caché está fría (es decir, vacía) y todas las solicitudes son cache misses.**

El siguiente problema son los datos obsoletos. Cuando hay un cache hit, devolvemos los datos de la caché en lugar de la base de datos de origen. Este es el propósito de una caché, pero introduce la posibilidad de devolver datos obsoletos si estos han cambiado en la base de datos de origen.

El truco está en encontrar el equilibrio correcto entre la probabilidad de devolver datos obsoletos y tener demasiados cache misses.

El problema final es la complejidad. Una read-through cache añade otra capa de tecnología al call stack.

El patrón CQRS y las vistas materializadas proporcionan una mejor solución porque mantenemos los datos actualizados en tiempo casi real. Esto simplifica la experiencia del desarrollador porque el código para materializar datos está separado del código para recuperarlos.

Capacidad por lector, por consulta

Como hemos visto, poner todos los datos en la misma base de datos genera competencia por recursos escasos y crea un único punto de fallo. El patrón CQRS proporciona una solución completa dando a cada lector control total sobre sus propios datos derivados. Estamos, en esencia, asignando capacidad por lector, por consulta.

Cada servicio soporta un lector diferente (es decir, un actor) y crea y mantiene sus propias vistas materializadas e índices, en lugar de depender de una base de datos centralizada y compartida.

Datos ligeros

Los volúmenes ingobernables de datos, como ya hemos visto, son una de las causas de la gravedad de los datos. El gran volumen de datos en un sistema puede obstaculizar su evolución porque el tiempo y esfuerzo involucrado en remodelar los datos es un poderoso factor disuasorio.

Mover estos datos derivados a los almacenes de datos de los servicios que los usan hace que los almacenes de datos de origen sean aún más ligeros.

Proyecciones

La duplicación de datos es una de las preocupaciones generales del patrón CQRS.

Cuando creamos vistas materializadas a partir de eventos ascendentes, no necesitamos retener todos los datos que contiene un evento. El event lake actúa como el transaction log a escala de sistema y el registro de verdad para que otros servicios no tengan que serlo. Esto significa que los servicios descendentes pueden proyectar la cantidad mínima de datos en sus vistas que sea necesaria para soportar a sus usuarios finales.

Mantener las vistas materializadas ligeras obviamente ayuda a reducir los costos de almacenamiento, pero también mejora el rendimiento y reduce los costos generales. La cantidad de capacidad que consume una consulta depende de la cantidad de datos que recupera. Por lo tanto, no queremos procesar más datos de los necesarios.

Mantener los datos ligeros también mejora el rendimiento de las consultas, lo que reduce el tiempo que tardan en ejecutarse las funciones y, por lo tanto, reduce aún más los costos.

Time to Live (TTL)

Si bien las proyecciones son importantes para mantener los registros individuales ligeros, también necesitamos abordar el número total de registros. Con demasiada frecuencia dejamos que las tablas crezcan y crezcan aunque solo usemos los datos más recientes en el día a día.

En el contexto de un servicio específico, los datos tienen una vida útil por consulta. A medida que materializamos datos, sabemos cuánto tiempo necesitamos conservarlos para cada consulta. **Por lo tanto, al insertar y actualizar datos, debemos establecer un time-to-live (TTL) para cada registro.**

```
toUpdateRequest: (uow) => ({
    Key: {
      pk: uow.event.thing.id,
      sk: 'Thing',
    },
    ...updateExpression({
      name: uow.event.thing.name,
      ttl: uow.event.timestamp + (60*60*24*33) // 33 días
    }),
  }),
```

Para soportar tablas ligeras, simplemente necesitamos añadir un campo `ttl` a cada registro. En este caso, el valor se calcula en base a `uow.event.timestamp`, que indica cuándo ocurrió el evento de negocio.

Al ejecutar comandos para crear y modificar una entidad de dominio, es mejor usar datos dentro de la entidad para calcular el TTL, como la fecha de cierre esperada de una solicitud de hipoteca.

Eliminar datos más antiguos hace las tablas ligeras al mantener el número de registros relativamente constante. También ayuda a reducir el impacto de los cambios en los servicios ascendentes. Como acabamos de cubrir en la sección Proyecciones, si un cambio ascendente afecta a uno de los campos proyectados, necesitamos hacer un cambio para todos los nuevos registros. **Sin embargo, puede que no necesitemos convertir los registros existentes si el TTL es suficientemente corto.**

Establecer un TTL para los registros desempeña un papel importante en la lucha contra la gravedad de los datos al controlar el volumen de datos. Esto, a su vez, facilita la evolución de los sistemas. Sin embargo, **debemos tener cuidado de no eliminar datos antiguos de forma demasiado agresiva, ya que esto puede tener un impacto en la idempotencia y la tolerancia al orden del sistema.**

Implementación de idempotencia y tolerancia al orden

En el Capítulo 4, Confianza en los hechos y la consistencia eventual, aprendimos que la entrega exactamente una vez de los mensajes es poco realista. Para dar cuenta de la realidad de la entrega al-menos-una-vez, debemos diseñar nuestros sistemas para que sean idempotentes. En otras palabras, **no importa cuántas veces recibamos y procesemos un evento o solicitud, solo debe actualizar el sistema una vez.**

También aprendimos que entregar mensajes en orden puede ser problemático. Un stream ciertamente entregará los eventos en el orden en que los recibió, pero puede que no los reciba en el orden correcto. **Para dar cuenta de esta realidad, debemos diseñar nuestros sistemas para que sean tolerantes al orden.**

Identificadores deterministas

Es una práctica muy común delegar la generación de identificadores al lado del servidor, como usando un número de secuencia auto-incremental generado por la base de datos. **Este es un enfoque conveniente, pero no es idempotente.**

La solución a este problema es que el cliente genere el identificador y envíe una solicitud `HTTP PUT` en su lugar. Esto garantizará que la solicitud sea idempotente y solo creará y actualizará un registro sin importar cuántas veces el cliente reenvíe la solicitud.

Hay una variedad de técnicas alternativas para generar un identificador en el lado del cliente. Una opción aparentemente simple es generar un Universally Unique Identifier (UUID).

Para las entidades de dominio, un UUID V4 es el más apropiado porque se generan con números aleatorios que ayudan a evitar particiones calientes (hot partitions).

Para los eventos de dominio, un UUID V1 es el más apropiado porque se generan en base al tiempo, lo que permite que se ordenen de forma natural.

Sin embargo, hay una limitación en el enfoque de UUID. Se desmorona si el cliente olvida el UUID que acaba de generar. Para resolver este problema en un cliente SPA, podemos aprovechar la consistencia de sesión (session consistency), como discutimos en el Capítulo 3, Dominio de la capa de presentación. Con este enfoque, el cliente mantiene todo en el almacenamiento de sesión durante la duración de la sesión del usuario.

Otra opción es crear un identificador concatenando una combinación única de campos del contenido de la entidad de dominio. Una gran ventaja de este enfoque es que el identificador tiene significado de negocio.

Hay un escenario en el que tiene sentido usar un identificador que genere la base de datos. Como discutiremos en breve en la sección Uso de Change Data Capture, podemos crear stream processors que procesen el stream de eventos de cambio que un almacén de datos emite a medida que insertamos, actualizamos y eliminamos registros. **Estos eventos tendrán identificadores que la base de datos generó. Es seguro reutilizarlos como identificadores de eventos de dominio porque estos valores no cambiarán, sin importar cuántas veces un stream processor reintente.**

Y finalmente, otro enfoque es generar el hash de la entidad de dominio o del evento de dominio completo. Es un enfoque simple, pero existe la posibilidad de que genere colisiones. **Esto dependerá de la complejidad y variabilidad de la entidad de dominio, así que considere esta opción con cuidado.**

Bloqueo optimista inverso

Desde la perspectiva del frontend, usamos una **técnica de bloqueo optimista tradicional** para evitar que múltiples usuarios actualicen el mismo registro de forma concurrente. Almacenamos un timestamp o contador en un campo `oplock` y solo permitimos que un usuario actualice el registro si el valor del oplock no ha cambiado desde que el usuario recuperó los datos. Si otro usuario ha actualizado el registro y por lo tanto el valor del `oplock`, lanzamos una excepción cuando el usuario actual intenta actualizar el registro y obligamos al usuario a recuperar los datos de nuevo antes de proceder con la actualización.

Por el contrario, usamos la **técnica de bloqueo optimista inverso** en los stream processors asíncronos descendentes para proporcionar tanto idempotencia como tolerancia al orden. Garantiza que los eventos más antiguos o duplicados no sobreescriban los datos más recientes. En lugar de forzar a una transacción a reintentarse cuando falla el oplock, simplemente hacemos lo inverso; descartamos los eventos entrantes si el timestamp es más antiguo o igual al valor oplock actual en el almacén de datos.

```
export const toUpdateRequest = (uow) => ({
  Key: {
    pk: uow.event.thing.id,
    sk: 'thing',
  },
  ExpressionAttributeNames: {
    '#name': 'name',
    '#oplock': 'timestamp',
  },
  ExpressionAttributeValues: {
    ':name': 'Thing One',
    ':timestamp': uow.event.timestamp,
  },
  UpdateExpression: 'SET #name = :name,
                      #oplock = :timestamp',
  ConditionExpression: 'attribute_not_exists(#oplock)
      OR :timestamp > #oplock',
});
```

Lo primero que hay que notar en este ejemplo que es diferente con un `inverse oplock` es que la comparación es mayor que en lugar de igual a. Si el valor es igual al valor anterior, entonces esto indica que ya hemos procesado este evento.

Si el valor es menor que el valor anterior, entonces esto indica que estamos procesando un evento más antiguo fuera de orden. Cuando este valor es mayor que el valor anterior, sabemos que tenemos un evento más reciente, así que podemos proceder con la actualización.

La última pieza del rompecabezas es manejar el valor de respuesta devuelto por la llamada para actualizar la base de datos.

```
db.update(params).promise()
  .catch((err) => {
    if (err.code !== 'ConditionalCheckFailedException') {
      throw err;
    }
  });
```

Cuando ignoramos una excepción, es una buena práctica registrar una métrica para que podamos rastrear con qué frecuencia ocurre esto y evaluar si hay una causa raíz que debamos abordar.

Al usar la técnica de bloqueo optimista inverso, **es importante asegurarse de que no estamos descartando datos importantes cuando ignoramos eventos más antiguos**. Esto podría ocurrir si estamos usando el mismo campo oplock para diferentes tipos de eventos.

Si el evento más antiguo es de un tipo diferente, entonces probablemente tiene diferentes campos que queremos capturar en el mismo registro, por lo que no queremos descartar este evento más antiguo.

Esencialmente estamos materializando un registro de join a partir de múltiples tipos de eventos. Los diferentes tipos de eventos llevan el mismo valor que estamos usando como identificador del registro de join, más campos adicionales por tipo de evento que queremos unir. **Si usáramos el mismo campo oplock para todos estos tipos de eventos, tendríamos que recibirlos en el orden correcto.**

La solución para unir múltiples tipos de eventos en el mismo registro de join es usar un campo `oplock` diferente para cada tipo de evento. Podemos referirnos a esto como un **Inverse Oplock Join**.

Esto nos permite construir incrementalmente estos registros de join, un evento a la vez, en cualquier orden. El primer evento que llegue creará el esqueleto del registro con la información que porta. Cada evento adicional completará el registro con la información adicional que porta. Dentro de cada tipo de evento, podemos descartar de forma segura los eventos más antiguos o duplicados.

Disparadores de eventos inmutables

Una de las ventajas de la técnica de bloqueo optimista inverso es que no requiere que el stream processor recupere ningún dato para realizar un join. Esto mejora el throughput del stream processor y reduce la posibilidad de errores. Sin embargo, no siempre es posible calcular la respuesta correcta sin realizar una consulta para recuperar información adicional. Para obtener lo mejor de ambos mundos, podemos aprovechar el event sourcing añadiendo un micro almacén de eventos a la ecuación y dividiendo la lógica en dos stream processors separados.

El primer stream processor *recopila* eventos de interés del stream de eventos y los almacena en un micro almacén de eventos.

El event sourcing facilita la idempotencia porque los eventos son inmutables. Podemos poner el mismo evento con el mismo identificador determinista en el micro almacén de eventos una y otra vez, pero solo producirá un registro.

El segundo stream processor *gestiona* el evento después de insertarlo en el micro almacén de eventos. Como discutiremos en breve en la sección Uso de Change Data Capture, podemos crear stream processors que procesen el stream de eventos que un almacén de datos emite a medida que insertamos, actualizamos y eliminamos registros.

Gracias a la inmutabilidad de los eventos, podemos estar seguros de que el almacén de datos solo emitirá el evento al stream una vez cuando creemos el registro por primera vez.

El segundo stream processor también proporciona tolerancia al orden recuperando eventos relacionados del micro almacén de eventos y reaccionando en base a los eventos que se han recibido hasta ese momento. Estos eventos podrían haber llegado en cualquier orden, pero aquí el stream processor puede ordenarlos y actuar en consecuencia.

Por ejemplo, si falta un evento esperado, el procesador puede no hacer nada y reevaluar cuando llegue el siguiente evento.

Modelado de datos para el rendimiento operacional

Ahora nos centraremos en el modelado de datos para soportar los **patrones de acceso** necesarios durante las fases operacionales del ciclo de vida de los datos.

Nodos, aristas y agregados

Para nuestros almacenes de datos operacionales, usaremos bases de datos NoSQL serverless.

La distinción principal entre los enfoques nuevo y antiguo es que nos centramos más en las filas que en las tablas. **Más específicamente, estos almacenes de datos son sin esquema (schemaless), por lo que nos centramos en los tipos de entidades de dominio que almacenamos en cada fila.**

Los tipos almacenados en cada fila caerán en una de dos categorías: nodos y aristas.

- Un **nodo** representa una entidad de dominio con un identificador único y
- una **arista** representa una asociación entre nodos.

Con esto en mente, tiene sentido usar una notación estilo grafo.

Un **agregado** representa la raíz de un conjunto de objetos de dominio relacionados. Proporcionan perspectiva en el contexto del modelo de datos más amplio. Definen diferentes puntos de partida para navegar a través de los nodos y aristas del modelo.

Sharding y claves de partición

La unidad de disco impone el límite definitivo sobre cuán rápido podemos escribir y leer datos y cuántas solicitudes concurrentes podemos soportar.

Sharding es el proceso de particionar horizontalmente una base de datos en muchas unidades de disco. Esto permite que una base de datos soporte más solicitudes concurrentes. La base de datos realiza un seguimiento de qué datos están en qué disco y distribuye los datos de manera uniforme entre los discos para ayudar a equilibrar la carga.

En el lado de la aplicación, necesitamos seleccionar y usar correctamente las claves de partición para usar eficazmente un almacén de datos con sharding. La base de datos usa una **clave de partición (pk)** para asociar un registro a una partición.

Esto significa que podemos optimizar el rendimiento modelando nuestros datos operacionales de modo que no necesitemos hacer joins entre múltiples discos.

También debemos tener cuidado de evitar las claves de partición calientes (hot pks) que no están bien distribuidas entre particiones. Por ejemplo, si usamos la fecha actual como pk, todas las escrituras de un día determinado irán al mismo disco y el sistema no obtendrá ninguna concurrencia adicional del sharding.

Un UUID V4 es una buena opción para las pks porque se generan con números aleatorios que resultan en una buena distribución entre particiones.

Ejemplos de diseño de tabla única

En primer lugar, la técnica de Single Table Design aprovecha la naturaleza sin esquema de estas bases de datos. Esto significa que cada fila de la tabla puede representar un tipo diferente de entidad de dominio. Siguiendo nuestro modelo de datos lógico orientado a grafos, podemos pensar en cada fila como un nodo o una arista. Un nodo representa una entidad de dominio con un identificador único y tiene un conjunto de atributos. Una arista representa una asociación entre nodos y también puede contener un conjunto de atributos.

Usamos un valor discriminador para ayudar al código a determinar el tipo de dominio de cada fila. El discriminador puede implementarse como un campo separado o incrustado en la clave de ordenamiento (sk). Los siguientes ejemplos usan un campo discriminador separado para mayor claridad.

El siguiente elemento igualmente importante de la técnica es la pk. Como ya hemos visto, una pk mapea un registro a un shard específico (es decir, disco). Queremos aprovechar este hecho y usar el campo pk para precalcular un join entre diferentes registros almacenándolos todos en el mismo shard.

Es útil pensar en la pk como el identificador único de un agregado de dominio que representa una colección de objetos de dominio relacionados con los que queremos trabajar como un grupo. **Dentro de una pk, usamos un sk para identificar de forma única cada uno de los registros relacionados.**

Uso de Change Data Capture

Hasta este punto en este capítulo, nos hemos centrado principalmente en el lado de consumo y consulta descendente del flujo CPCQ. Ahora, veremos el lado de comando y publicación ascendente del flujo CPCQ y cómo se intersecta con el patrón Event Sourcing a escala de sistema que introdujimos en el Capítulo 4, Confianza en los hechos y la consistencia eventual.

En la sección Inversión de la base de datos, aprendimos que dividir los datos en múltiples bases de datos crea la necesidad de un transaction log a escala de sistema. **Satisfacemos esta necesidad con el patrón Event Sourcing y el event lake.**

Cada subsistema autónomo desempeña su papel proporcionando un servicio de event hub para recopilar eventos y un servicio de event lake para almacenarlos.

Los servicios autónomos individuales desempeñan su parte publicando eventos al hub a medida que su estado cambia.

Cada base de datos individual todavía tiene su propio transaction log, y lo aprovechamos como una herramienta poderosa. **Change Data Capture (CDC)** es una técnica que lee el transaction log de una base de datos en tiempo real y captura estos eventos de cambio a nivel de base de datos, exponiéndolos a consumidores externos como un stream para que puedan reaccionar a medida que los datos cambian.

Event sourcing con base de datos primero

Creamos sistemas eventualmente consistentes encadenando una serie de acciones atómicas que cada una actualiza un único almacén de datos. El enfoque que cubrimos para iniciar esta reacción en cadena es la variante stream-first del patrón Event Sourcing.

Con esta variante, primero emitimos (es decir, escribimos) un evento al event hub, donde un stream actúa como almacén de datos temporal para que los servicios descendentes puedan reaccionar y materializar datos ligeros en sus propios almacenes de datos.

El event sourcing stream-first es más adecuado para escenarios de disparar y olvidar donde el productor del evento no necesita interactuar inmediatamente con el resultado del procesamiento.

Los servicios BFF soportan a los usuarios finales, y los usuarios finales típicamente necesitan interactuar con los datos que crean. **Aquí es donde los patrones Event Sourcing y CQRS obtienen su mala reputación.**

Aquí es donde CDC puede ayudar. CDC nos brinda la oportunidad de tener lo mejor de ambos mundos. Podemos tener la experiencia de desarrollador simplificada de escribir y leer el estado actual con un único modelo de datos, al mismo tiempo que producimos atómicamente eventos de dominio para que los servicios descendentes puedan soportar sus propios modelos de lectura.

La función de comandos realiza la única acción atómica de escribir el resultado de su procesamiento en una tabla de su propiedad. **La característica CDC de la base de datos, como Amazon DynamoDB Streams,** hace visible el evento de cambio de la base de datos a la función trigger. La función trigger transforma el evento de cambio y produce un evento de dominio al event hub.

Figure 5.7: Database-first event sourcing

Eliminaciones lógicas

La idempotencia y la tolerancia al orden pueden ser difíciles de lograr cuando realizamos una eliminación física (hard delete). Una eliminación física elimina los datos físicamente, mientras que una eliminación lógica (soft delete) simplemente marca los datos como eliminados.

En el stream CDC, el evento de cambio aparecerá como una actualización. Sin embargo, la función trigger verá que el indicador de eliminado está establecido en verdadero y publicará un evento `thing-deleted` en su lugar. Esta lógica de trigger se implementa como una característica en el patrón del pipeline CDC.

Es importante mantener los datos ligeros, así que también necesitamos decidir cuánto tiempo conservar los registros que marcamos como eliminados. Para los datos lentos con registros limitados, puede tener sentido conservar estos registros indefinidamente. De lo contrario, podemos establecer un TTL como discutimos en la sección Datos ligeros.

Latching (Bloqueo)

Para habilitar un cambio continuo, una arquitectura evolutiva debe soportar la ejecución concurrente de múltiples versiones de una funcionalidad.

Sin embargo, al igual que en una migración de sistemas heredados, hay momentos en que una evolución es más bien una revolución. En estos casos, para mitigar riesgos, lo más conveniente es crear una nueva versión del servicio que se ejecute de forma concurrente con la versión anterior.

Para lograr esto, necesitamos sincronizar los datos entre las dos versiones del servicio. Esto permitirá que se desarrollen varios escenarios posibles:

1. Si los usuarios beta consideran que la nueva versión no dio en el blanco, pueden volver a la versión anterior y continuar donde lo dejaron.
2. Si la funcionalidad necesita migrarse de forma incremental, los usuarios pueden realizar parte del trabajo en la nueva versión y parte en la anterior de manera transparente.
3. Finalmente, cuando se libere la nueva versión a todos los usuarios, los datos ya estarán convertidos y se podrá retirar la versión anterior.

Al implementar una sincronización bidireccional como esta, debemos tener cuidado de no crear un bucle infinito.

El *latching* (bloqueo), al igual que echar llave a una puerta, es la solución a este problema. La función `trigger` no emitirá un evento cuando el *latch* (bloqueo) esté cerrado. La función `listener` cerrará el bloqueo cuando materialice datos, y una función `command` abrirá el bloqueo cuando un usuario del servicio realice una acción.

Esto evita que las actualizaciones de sincronización entren en bucle, pero permite que los cambios de estado iniciados por el usuario sean emitidos.