4. Confianza en los hechos y la consistencia eventual

Veremos cómo un event hub se sitúa en el núcleo de cada subsistema autónomo y crea una barrera de salida que protege a los servicios autónomos upstream de las interrupciones downstream.

Vivir en un mundo de consistencia eventual

Las bases de datos relacionales y las integraciones en tiempo real se suponía que resolverían estos problemas de consistencia de datos, pero en cambio, produjeron sistemas altamente acoplados, difíciles de modificar y propensos a fallos en cascada.

Los datos son el activo más importante de un sistema de información. Deben ser precisos. Sin embargo, en lugar de controlar los datos de manera demasiado estricta, **queremos aceptar el hecho de que vivimos en un mundo de consistencia eventual.**

Procesamiento por etapas

Staged Event-Driven Architecture (SEDA). Hemos dividido el proceso en un conjunto de etapas, y cada etapa tiene una cola de entrada. Las colas mejoran la tasa de rendimiento al minimizar la necesidad de que una etapa espere a otra. Las etapas upstream no necesitan esperar a las etapas downstream, y las etapas downstream nunca están inactivas salvo que no haya trabajo.

A nivel de servicio, el event hub proporciona las colas (es decir, los streams) y los servicios autónomos realizan el trabajo en las distintas etapas de un proceso.

A nivel de función, las funciones individuales del procesador de streams dentro de los servicios autónomos implementarán sus propios flujos de procesamiento por etapas. Aprovecharán la **E/S asíncrona sin bloqueo** para proporcionar las colas que permiten la multitarea, y la **Programación Reactiva Funcional (FRP)** para definir las etapas de trabajo.

Cooperación

La noción de programación cooperativa también está presente aquí, tanto a nivel de función como de servicio. **La cantidad de trabajo realizado en cada etapa debe ser similar.** De lo contrario, se crea un flujo de proceso desbalanceado, en el que una etapa rápida puede saturar a la siguiente, o una etapa lenta puede privar de trabajo a la siguiente. No existe una manera particular de imponer esto, más allá del **diseño intencional y la cooperación entre equipos**, de ahí el término cooperativo.

Atomicidad

Cada etapa también debe realizar una unidad de trabajo atómica. En otras palabras, cada etapa debe actualizar un único recurso. **Esto nos permitirá maximizar el paralelismo.** También minimizará el desperdicio cuando algo salga mal, ya que no será necesario repetir etapas exitosas. En última instancia, **encadenaremos una serie de acciones atómicas (es decir, etapas) para lograr la consistencia eventual.**

Consistencia

Todos los participantes acuerdan cómo funciona el proceso y comprenden que una pequeña inconsistencia produce el mejor resultado para todos los involucrados. **El proceso funciona porque contamos con transparencia, hechos y una reacción en cadena.**

Transparencia

La transparencia es esencial en un sistema de consistencia eventual.

- Los usuarios necesitan entender el proceso para poder establecer sus expectativas.
- También necesitan conocer el estado del sistema para poder tomar decisiones informadas.

Hechos

Los hechos son el elemento vital de un sistema de consistencia eventual. Cada etapa produce un evento para registrar el hecho de que completó una unidad de trabajo atómica. **Estos hechos no son efímeros. Los conservamos indefinidamente. Representan datos en movimiento.**

En conjunto, estos hechos representan el estado actual del sistema. Los servicios downstream pueden estar temporalmente inconsistentes, pero **los hechos contienen toda la información necesaria para hacerlos consistentes.**

Reacción en cadena

Logramos la consistencia eventual encadenando acciones atómicas. Cada acción atómica produce un hecho (es decir, un evento), y la siguiente acción atómica en la cadena reacciona y produce su propio hecho. Esta reacción en cadena continúa hasta que se alcanza el resultado deseado. **Diseñamos específicamente la reacción en cadena para asegurar que el sistema llegue a ser consistente.**

En cualquier momento dado, podemos revisar los hechos, evaluar el estado actual del sistema y determinar si hay trabajo pendiente por completar. **Cuando todo funciona correctamente, una reacción en cadena ocurrirá en tiempo casi real.** Sin embargo, las cosas saldrán mal, y debemos estar preparados.

Concurrencia y particiones

El teorema CAP nos muestra que, en caso de una partición del sistema, debemos elegir entre consistencia y disponibilidad. No podemos tener ambas. Los usuarios claramente prefieren la disponibilidad y esperan que garanticemos que

todo eventualmente converja de manera ordenada. Esto es suficientemente sencillo cuando ningún otro usuario opera sobre los mismos datos lógicos. Sin embargo, debemos considerar la concurrencia. Debemos tener en cuenta qué sucede cuando los usuarios interactúan con los mismos datos lógicos de forma concurrente.

Existen muchas oportunidades para que ocurran particiones en nuestros sistemas.

- El modo sin conexión es un ejemplo obvio.
- Los servicios downstream experimentan una breve partición mientras los eventos upstream avanzan a través de las reacciones en cadena, y estas particiones pueden prolongarse bajo condiciones de error.
- Otro ejemplo ocurre cuando dos servicios upstream operan sobre instancias separadas de los mismos datos lógicos.

Múltiples usuarios pueden operar simultáneamente sobre diferentes instancias de los mismos datos lógicos, pero registramos el resultado de cada acción como un hecho para no perder información.

El cambio de un usuario no necesariamente tiene que sobrescribir el del otro. Downstream, podemos diseñar el modelo de datos para admitir ambos cambios, combinarlos o descartar uno u otro. En cualquier caso, conservamos los hechos como registro de auditoría.

Tolerancia al orden e idempotencia

Muchos sistemas de mensajería ofrecen garantías de orden de llegada (FIFO). Cuando todo funciona correctamente, esto puede ayudar a mejorar la eficiencia del sistema. Sin embargo, no podemos depender de estas garantías. **Los mensajes pueden llegar al sistema en el orden incorrecto, otros fallarán y serán devueltos a la cola, y múltiples canales pueden avanzar a distintos ritmos.**

Paralelismo

Ya hemos mejorado el rendimiento al dividir el proceso en etapas y permitir que distintos ítems de una orden se distribuyan hacia diferentes etapas. Sin embargo, podemos hacerlo mejor, mucho mejor, añadiendo otra instancia del servicio. Esto es la teoría clásica de colas en acción.

Nuestros sistemas de consistencia eventual se basan en colas, lo que hace que el paralelismo sea una solución natural.

La realidad es que nuestros sistemas viven en un mundo de consistencia eventual. Alguna parte de casi cualquier sistema es eventualmente consistente. Los usuarios móviles necesitan la capacidad de trabajar en modo sin conexión, lo que significa que son eventualmente consistentes. Las integraciones con sistemas legados y externos son eventualmente consistentes, e incluso pueden ser orientadas al procesamiento por lotes.

En lugar de imponer un modelo de consistencia estricta a un mundo de consistencia eventual, es mejor abrazar la consistencia eventual en todos los niveles. La impedancia de cambiar entre distintos modelos en realidad nos ralentiza e impide la innovación.

Publicación en un event hub

La capacidad de publicar eventos es una capacidad fundamental de los servicios autónomos. Los servicios upstream publican eventos a medida que su estado cambia, y los servicios downstream reaccionan a estos eventos. **Este paradigma de publicación y suscripción desacopla a los productores de los consumidores,** pero si no tenemos cuidado, podemos introducir inadvertidamente acoplamiento y complejidad a través de la infraestructura de mensajería.

El flujo general de eventos a través del event hub es el siguiente:

1. Los servicios upstream publican eventos de dominio al hub a través de un bus.
2. El bus enruta los eventos hacia uno o más canales, como un stream.
3. Los servicios downstream consumen eventos del hub a través de un canal específico.

Event bus

El event bus es el punto de entrada al event hub. Proporciona un nivel de indirección para que no acoplemos a los productores con canales de consumo específicos. **Los productores solo necesitan conocer el bus.** Añadiremos reglas para enrutar eventos a canales de consumo específicos sin impactar a los productores.

El event bus también actúa como la barrera de salida que protege a los servicios upstream de los fallos e interrupciones downstream. El bus es un servicio serverless de alta disponibilidad, como AWS EventBridge o Azure Event Grid.

Un servicio upstream puede publicar un evento al bus y olvidarse de él. Puede confiar en que el bus eventualmente entregará el evento a todos los consumidores interesados, a través de los distintos canales.

Eventos de dominio

Los eventos de dominio son el elemento vital de un sistema orientado a eventos. Los usamos para transmitir información entre servicios.

- Un actor interactúa con un servicio para realizar una acción sobre un modelo de dominio.
- El servicio produce un evento de dominio para registrar el hecho de que el actor realizó la acción.
- Luego enviamos el evento de dominio desde el servicio productor a todos los servicios consumidores a través del event hub.

Sobre del evento (envelope)

El sobre (envelope) del evento define un conjunto estándar de campos que todos los eventos deben contener. Esto permite que el bus, cualquier canal y todos los consumidores puedan gestionar cualquier evento. El bus se basa en estos campos para realizar el enrutamiento basado en contenido hacia canales específicos.

Transferencia de estado mediante eventos

Nuestros eventos de dominio son más que simples notificaciones. Representan el hecho de que algo ocurrió y contienen los datos que proveen el contexto del evento. Los servicios downstream usan estos datos para tomar decisiones y crear vistas materializadas. **Nos referimos a esto como el patrón de transferencia de estado mediante eventos.**

Por ejemplo, dos servicios BFF pueden ofrecer a distintos actores la capacidad de trabajar sobre diferentes aspectos de la misma entidad de dominio. Ambos servicios tienen una copia liviana y de solo lectura del resumen de la entidad de dominio, pero cada uno es responsable de mantener diferentes subsecciones del modelo de dominio.

Cuando los BFFs publican eventos, incluyen la información de resumen como contexto, junto con la información detallada que cada uno posee. Sin embargo, ninguno de los dos necesita mantener copias del submodelo de dominio del otro ni incluirlo en sus eventos. En cambio, los servicios downstream combinarán los subdominios en sus propias vistas materializadas según sus necesidades.

Sustitución

Necesitamos la capacidad de cambiar los productores de eventos sin forzar a los consumidores a cambiar. En otras palabras, **un servicio downstream debe poder consumir el mismo evento de dominio independientemente de qué servicio lo produjo.**

Con el tiempo, los productores pueden cambiar y, en cualquier momento, puede haber múltiples fuentes produciendo el mismo evento de dominio.

El contrato sí necesita evolucionar, en particular en las etapas tempranas del desarrollo. Para equilibrar la flexibilidad con la estabilidad, mantendremos eventos de dominio internos y externos.

Internos versus externos

Los eventos de dominio internos definen los contratos dentro de un subsistema, y los eventos de dominio externos definen los contratos entre subsistemas.

Ofrecemos sólidas garantías de compatibilidad hacia atrás a los subsistemas downstream para los eventos de dominio externos. Esto permite a los equipos experimentar y modificar la definición de los eventos de dominio internos con mayor libertad.

Enrutamiento y topología de canales

El enrutamiento es la razón de ser del event hub. El enrutamiento ayuda a desacoplar a los productores de los consumidores. Los productores envían eventos al bus, que es responsable de enrutarlos hacia los distintos canales de consumo.

Un consumidor generalmente se suscribe a un único canal. Muchos consumidores pueden compartir un canal. El hub típicamente posee los canales compartidos y las reglas de enrutamiento asociadas, mientras que **un servicio poseerá un canal dedicado y añadirá sus propias reglas de enrutamiento al bus.**

Un subsistema puede necesitar una topología de múltiples canales para soportar el flujo de eventos hacia los consumidores downstream. Hay una variedad de aspectos a considerar al definir una topología de canales, tales como:

- aislamiento,
- tamaño de los mensajes,
- prioridad,
- volumen,
- y número de consumidores.

Puede ser necesario aislar canales específicos para asegurar que no interfieran entre sí.

El tamaño de los eventos es importante. La regla de enrutamiento que definimos en los párrafos anteriores filtra los eventos de fallo. Este es un ejemplo de exclusión de un tipo de evento específico basado en el potencial tamaño del mensaje. Los canales tienen límites de capacidad, y los eventos grandes pueden impactar el rendimiento de otros eventos. Estos eventos grandes pueden justificar un canal dedicado.

La prioridad de los eventos es otra consideración. Podemos definir un canal de alta prioridad para uno o más tipos de eventos, a fin de evitar la competencia con tipos de eventos de baja prioridad.

El volumen de eventos también es una consideración; puede tener sentido tener un canal separado para tipos de eventos de bajo volumen.

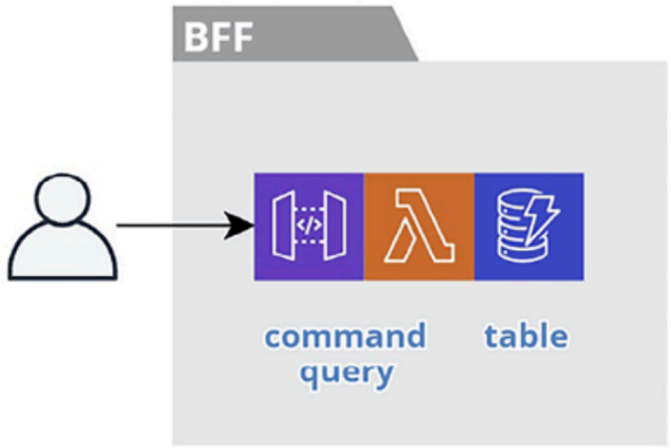
El número de consumidores también puede tener impacto en el rendimiento. Más consumidores en un único canal generan competencia por la capacidad de lectura, lo que puede resultar en throttling de lectura. En este caso, si no podemos añadir más capacidad al canal, podemos crear canales duplicados y distribuir los consumidores entre ellos.

Análisis del patrón Event Sourcing

El patrón Event Sourcing proporciona la base para la consistencia eventual y la flexibilidad posterior que nos permite crear sistemas reactivos y evolutivos.

Esencialmente propone convertir los eventos en hechos, en lugar de tratarlos como simples mensajes efímeros. Lo logramos conservando todos los eventos y almacenándolos de forma permanente. Los eventos sirven como registro de auditoría de la actividad histórica para que no perdamos información.

Lamentablemente, **el patrón Event Sourcing tiene la reputación de hacer los sistemas más complejos.** Esta preocupación no carece de fundamento. Sin embargo, con las modificaciones adecuadas, comunicarse mediante eventos y tratarlos como hechos produce en realidad sistemas más directos y flexibles.



Account	Date	Event	Amount
1	2021/01/31	debit	30.00
1	2020/12/31	credit	10.00
1	2020/11/30	debit	30.00
2	2021/02/12	debit	20.00
3	2021/03/13	debit	30.00

Figure 4.4: Traditional event sourcing

En este caso, no perdemos información porque almacenamos un registro separado para cada transacción. El comando sigue siendo directo porque simplemente escribe un nuevo registro con el monto del cambio y un indicador de crédito o débito. Sin embargo, la consulta se ha vuelto mucho más compleja.

La consulta necesita leer todas las transacciones de una cuenta y calcular el saldo en cada solicitud. Esto no es un gran problema cuando hay pocas transacciones, pero **a medida que crece el número de transacciones, el rendimiento se verá afectado.**

Tomar snapshots es una solución común a este problema de rendimiento, pero **los snapshots añaden complejidad** porque incorporan un componente adicional al diseño.

Event sourcing a nivel de sistema

Volviendo a la sección "Vivir en un mundo de consistencia eventual", vimos que un sistema de consistencia eventual actúa como una serie de etapas atómicas que intercambian hechos. Esto simplifica el sistema en su conjunto porque diseñamos cada etapa para centrarse en una única tarea, y luego encadenamos estos bloques constructivos. Además, una etapa no retiene los hechos; los transmite para poder concentrarse en realizar su trabajo.

La figura 4.5 muestra dos servicios BFF colaborando para soportar múltiples etapas en un sistema de consistencia eventual. El Servicio X proporciona el comando para actualizar el saldo de cuenta, y el Servicio Y proporciona la consulta para leer el saldo de cuenta:

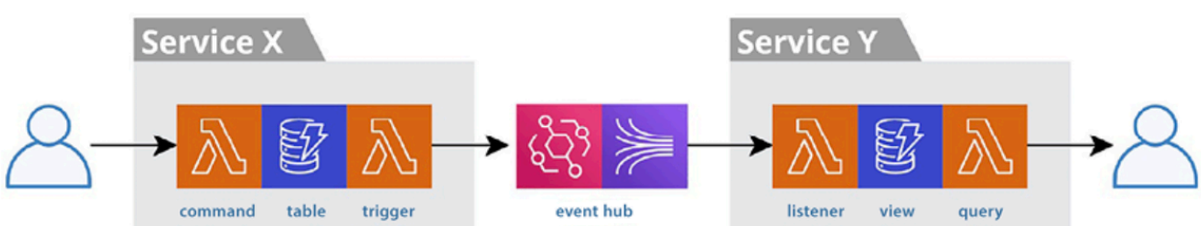


Figure 4.5: System-wide event sourcing

La función trigger en el Servicio X implementa la mejora. La implementación de su funcionalidad de comando permanece directa porque solo almacena el último saldo. Simplemente añadimos la función trigger para consumir los cambios del stream Change Data Capture (CDC) de la tabla y enviar estos hechos al event hub.

La mejora con la función trigger es un cambio aditivo directo. Hemos mantenido la simplicidad, el enfoque y la naturaleza atómica del Servicio X sin perder información, porque estamos enviando los hechos (es decir, los eventos) downstream.

Ahora, el Servicio Y y cualquier otro servicio pueden posicionarse downstream y reaccionar a los hechos que emiten los servicios upstream. Estamos ganando flexibilidad al tiempo que conservamos la información y la simplicidad. **Esto es la inversión de responsabilidad.**

Los servicios upstream simplemente emiten eventos para registrar el hecho de que completaron una acción atómica, sin preocuparse por lo que sucede después. Luego, añadimos servicios downstream según sea necesario. En otras palabras, **el event sourcing es un mecanismo clave para crear una arquitectura que habilita el cambio, porque permite modificar el sistema añadiendo servicios sin necesidad de modificar otros.**

Event lake

El event lake es responsable de convertir los eventos en hechos. Lo logra capturando todos los eventos y almacenándolos de forma permanente. Es el registro definitivo de verdad sobre las acciones que han tenido lugar dentro del sistema. Es un registro de auditoría completo que contiene todos los hechos.

En cualquier momento, podemos explorar el event lake para encontrar la verdad sobre cada cambio de estado.

El event lake mantiene sus propios canales y añade sus propias reglas de enrutamiento al event hub, asegurando que reciba todos los eventos publicados dentro del subsistema o consumidos desde subsistemas upstream.

Almacenamiento perpetuo

La durabilidad de los eventos dentro del lake es de la mayor importancia. Por esta razón, el almacenamiento de objetos es la mejor opción para persistir eventos en el lake.

Organizamos los eventos en el almacenamiento de objetos según su timestamp. Podemos explorar los eventos por año, mes y día, pero empaquetamos múltiples eventos en un único objeto para un rendimiento y almacenamiento óptimos.

La seguridad de los eventos en el event lake es crucial. Mantener un lake separado por subsistema ayuda a limitar el acceso con base en el principio de mínimo privilegio. También redactamos datos sensibles en el lake mediante cifrado de sobre.

Indexación de eventos

El event lake es una valiosa fuente de información. Conservar los eventos como hechos en almacenamiento duradero es crucial, pero **también necesitamos la capacidad de encontrar fácilmente los eventos de interés.**

La indexación de eventos nos permite buscar fácilmente eventos en múltiples dimensiones. Hay muchas formas de indexar eventos. Los eventos son muy adecuados para una base de datos de series temporales porque tienen un timestamp y diversas dimensiones, como tipo y etiquetas. También son muy adecuados para un motor de búsqueda porque tienen contenido rico y de granularidad gruesa.

Elasticsearch es especialmente adecuado para indexar esta información de series temporales de granularidad gruesa. También podemos pensar en el event lake como un archivo de log y elegir indexar los eventos en un sistema de monitoreo de logs y observabilidad de tipo SaaS. Amazon Athena es otra opción cuando almacenamos el lake en S3.

Reproducción de eventos

Conservar los hechos en el event lake nos permite reproducir eventos. Podríamos reproducir eventos para reparar un servicio averiado, o para inicializar un nuevo servicio o una nueva versión de un servicio.

Es importante señalar que no republicamos los eventos, porque esto los transmitiría a todos los consumidores. En cambio, enviamos los eventos a una función listener específica. Esta es la misma función listener que consume de un canal, como un stream. **Por tanto, la utilidad de reproducción necesita envolver los eventos en el formato del canal específico.**

No necesitamos filtrar hasta el conjunto exacto de eventos cuando los reproducimos hacia una función. Como veremos en la sección "Procesamiento de event streams", **una función listener es un procesador de streams que ya filtra los tipos de eventos de interés.** Por eso podemos especificar un rango de fechas y dejar que la función descarte los eventos no deseados.

Cuando reproducimos un rango de eventos, es probable que la función listener ya haya procesado algunos de ellos cuando fueron publicados originalmente. Por tanto, inevitablemente habrá duplicados, y cualquier evento perdido llegará ahora fuera de orden. **Esta es una razón más por la que es importante asegurar que nuestras funciones listener sean idempotentes y tolerantes al orden.**

Event streams

Un event stream es un canal de mensajería que actúa como almacén temporal de eventos para los consumidores downstream. Aprovechamos los streams para todos los canales de comunicación entre servicios. **Actúan como la cola de entrada para las distintas etapas en nuestro sistema de consistencia eventual.**

Los servicios autónomos escuchan eventos en un stream (es decir, un canal) para poder reaccionar en tiempo casi real. Desde su perspectiva, un canal o stream es la fuente de eventos. En otras palabras, **los streams proporcionan event sourcing temporal a los servicios downstream.**

Almacenamiento temporal

Los streams se diferencian de otros canales de mensajería porque también actúan como un almacén de datos de solo adición. Mientras que la mayoría de los canales de mensajería tratan los eventos como mensajes efímeros y los descartan una vez reconocidos, un stream conserva los eventos durante un período prescrito, como días, semanas o meses.

La simple noción de almacenar eventos los convierte en hechos. Comenzamos a tratar los eventos como ciudadanos de primera clase. Los resultados de todas las acciones atómicas que conducen a la consistencia eventual (es decir, los hechos) pasan a

formar parte del modelo de datos. Ya no conservamos únicamente la imagen estática más reciente. Los datos se vuelven dinámicos. Están en movimiento.

Persistir los eventos nos da la confianza de que los datos fluirán hacia los servicios downstream y el sistema eventualmente llegará a ser consistente. Podemos ajustar un canal determinado para que retenga los eventos el tiempo suficiente para asegurar que sus consumidores tengan el tiempo necesario para reaccionar. Esto libera a los servicios upstream de la responsabilidad de garantizar la consistencia.

Event sourcing con prioridad en el stream (Stream-First)

Las transacciones distribuidas han caído en desuso porque el ciclo de commit en dos fases no escala. Esto se evidencia en el hecho de que muchos servicios cloud-native no soportan transacciones distribuidas. **Esto significa que no podemos guardar datos de forma confiable en más de un almacén de datos a la vez.**

En cambio, creamos sistemas de consistencia eventual encadenando una serie de etapas atómicas. Cada etapa es atómica porque solo actualiza un único almacén de datos. Pero esto significa que debemos elegir entre escribir datos en una base de datos o emitir un hecho al event hub. No podemos hacer ambas cosas como una acción atómica.

Aquí es donde entran en juego las variantes stream-first y database-first del patrón event sourcing. Necesitamos convertir los eventos en hechos. Un servicio puede enviar primero un evento al event hub y confiar en que los servicios downstream reaccionarán y almacenarán los datos; o un servicio puede almacenar primero los datos y confiar en que su propia función trigger reaccionará y publicará el evento.

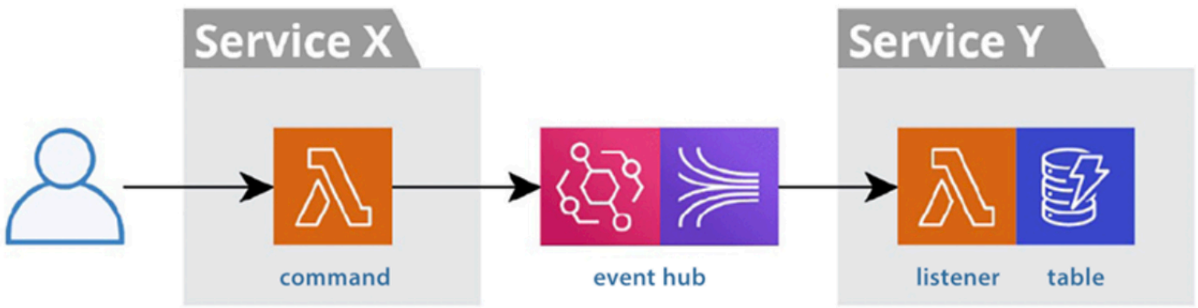


Figure 4.6: Stream-first event sourcing

Ya hemos visto que un stream es un almacén de datos temporal. Cuando un `command` envía un evento al `event hub`, podemos confiar en que el event hub lo enrutará hacia uno o más canales (es decir, streams) para su almacenamiento y consumo downstream.

Las funciones `listener` downstream consumen de estos canales y almacenan datos para su uso dentro de sus servicios.

La elección entre las variantes stream-first y database-first depende en gran medida de las capacidades de la base de datos en uso, además de la necesidad del usuario de interactuar con los datos de inmediato.

Si los usuarios necesitan interactuar con los datos que crean, entonces nos inclinaremos hacia la variante database-first. Sin embargo, esta variante requiere una base de datos que soporte CDC, como los streams de DynamoDB. Ahora bien, si la base de datos no soporta CDC, entonces los comandos del BFF pueden usar la variante stream-first y el BFF puede consumir sus propios eventos y persistir los datos.

Control de concurrencia

El control de concurrencia es otra razón por la que prefiero insertar un canal basado en streams entre el bus y los consumidores. Por ejemplo, Amazon EventBridge tiene la opción de enrutar eventos directamente desde el bus hacia funciones AWS Lambda. **Esta opción es adecuada para eventos de muy bajo volumen, pero puede tener impactos negativos en eventos de alto volumen.**

En primer lugar, este enrutamiento directo típicamente entrega un único evento a la vez a la función. **Esto incrementará el número de invocaciones de la función con el correspondiente aumento en el costo.** Esto también elimina las eficiencias que podemos obtener mediante el procesamiento por lotes de eventos, como veremos en la sección "Procesamiento de event streams". Estas ineficiencias también incrementarán el costo.

Además, este enrutamiento directo entrega eventos a las funciones tan rápido como es posible, sin considerar los límites de concurrencia de las funciones. Esto resultará en throttling y provocará que el bus reintente la entrega hasta que tenga éxito.

Podemos evitar todos estos problemas simplemente insertando un canal basado en streams. Configuramos la capacidad del stream para que pueda recibir eventos del bus sin throttling. Luego, la función procesadora del stream puede consumir del stream de manera ordenada.

Micro-almacenes de eventos

Un micro-almacén de eventos contiene un subconjunto de hechos que son importantes para un servicio específico. Mientras que el event lake es responsable de almacenar todos los eventos y un event stream almacena los eventos más

recientes, un servicio puede definir su propio micro-almacén de eventos para proporcionar acceso dedicado a un subconjunto de eventos.

El servicio recopila eventos de interés y los correlaciona en un micro-almacén de eventos, como una tabla de DynamoDB. Esto le da al servicio acceso optimizado para recuperar estos eventos relacionados. Retiene estos hechos solo el tiempo necesario estableciendo un time to live en cada registro.

Procesamiento de event streams

A primera vista, el procesamiento de streams puede parecer un cambio de paradigma extremo. Y para ser honestos, si hay que hacerlo todo desde cero, no es para los pusilánimes. Las ofertas serverless de función-como-servicio, como AWS Lambda, proporcionan esta capacidad a solo el costo de invocar las funciones para ejecutar la lógica de negocio, encargándose de todo el trabajo pesado.

Aun así, el procesamiento de streams es diferente, pero de manera positiva. La diferencia comienza con el tamaño del lote, lo que nos lleva a la programación reactiva funcional. Esto, a su vez, abre todo un ámbito de posibilidades, como filtrado, mapeo, reducción, contrapresión y mucho más.

Micro procesamiento por lotes

La firma de entrada de una función es una de las primeras cosas que debemos considerar al implementar un servicio con función-como-servicio. La estructura de los eventos de entrada depende del tipo de canal de mensajería que alimenta la función.

Una función AWS Lambda que consume de un stream Amazon Kinesis recibe un lote de registros como el siguiente:

```
{
  "Records": [{
      "kinesis": {
        "partitionKey": "1",
        "sequenceNumber": "49590338...88898",
        "data": "SGVsbG8s...N0Lg==",
      },...
    },{
      "kinesis": {
        "partitionKey": "1",
        "sequenceNumber": "49590338...5618",
        "data": "VghpcyBp...zdC4=",
      },...
    }]
}
```

La función solo recibirá un registro si hay un único registro en el stream, o hasta el límite del tamaño de lote especificado para la función cuando hay muchos registros en el stream. **Podemos pensar en esto como micro procesamiento por lotes, donde cada invocación procesa el siguiente micro-lote de eventos en el stream a nivel de servicio.**

El siguiente fragmento de serverless.yml, del template de servicio de ejemplo, crea una función listener y la configura para consumir del stream Amazon Kinesis que definimos como canal predeterminado en el event hub:

```
functions:
  ...
  listener:
    handler: src/listener/index.handle
    events:
      - stream:
          type: kinesis
          arn: ${cf:event-hub-${opt:stage}.stream1Arn}
          batchSize: 100
```

Tenemos control sobre el `batchSize`. En este caso, lo establecemos en 100. Esto es significativo porque abre la oportunidad para las muchas optimizaciones que cubriremos en este capítulo, como la contrapresión y la E/S asíncrona sin bloqueo.

Esto es lo opuesto a muchos canales de mensajería tradicionales que solo nos permiten procesar un mensaje a la vez, aunque estén realizando agrupamiento y almacenamiento en búfer a nivel inferior.

El tamaño de lote también puede tener un impacto positivo significativo en el costo del cómputo serverless en volúmenes altos, porque pagamos por cada invocación de una función. En lugar de pagar por una invocación por registro, el tamaño de lote actúa como un factor para reducir el número de invocaciones. Y cuando aplicamos las muchas optimizaciones, también mejoramos la eficiencia de cada invocación.

A nivel de servicio, hemos elegido usar tecnología de streaming para nuestros canales de mensajería. A nivel de función, el tamaño de lote nos permite fragmentar los datos que fluyen por los canales y crear streams a nivel de función dentro de nuestros procesadores de streams. Esto, a su vez, significa que necesitamos elegir un modelo de programación más adecuado para el procesamiento de streams.

Elección de un paradigma de programación

Necesitamos decidir qué tipo de paradigma de programación usaremos para procesar nuestro lote (es decir, array) de registros. La programación imperativa es la elección tradicional, pero tiene algunas ineficiencias en este contexto. La programación reactiva funcional (FRP) es un paso en la dirección correcta, pero podemos hacerlo mejor. En última instancia, elegiremos usar el procesamiento de streams, pero veamos cómo llegamos a esa decisión.

Programación imperativa

```
export const handler = async (event) => {
  for (const record of event.Records) {
    if (filterOnEventType(record)) {
     const updateRequest = mapToUpdateRequest(record);
     await update(updateRequest);
    }
  }
  return 'Success';
};
```

En este ejemplo, usamos un bucle for para iterar sobre los registros del array entrante. Luego, usamos una sentencia if para identificar los registros que nos interesan. A continuación, mapeamos los datos al formato deseado y actualizamos el destino.

Necesitamos transformar los datos entrantes a otro formato y luego enviarlos a otro servicio, como un almacén de datos o el event hub. **Esto significa que nuestras funciones están limitadas por E/S. En otras palabras, pasaremos la mayor parte del tiempo de procesamiento esperando que las llamadas de E/S se completen.**

Esto es muy ineficiente. Es razonable esperar que estas funciones operen con una utilización inferior al 10%.

Programación reactiva funcional

Procesar un array de registros es una solución natural para FRP. **La diferencia más obvia entre FRP y la programación imperativa es la ausencia de bucles.** FRP es mucho más declarativa.

```
export const handler = async (event) => {
  await Promise.all(
    event.Records
     .filter(onEventType)
     .map(toUpdateRequest)
     .map(updateRequest => await update(updateRequest)
  );
  return 'Success';
};
```

En este ejemplo, usamos los métodos proporcionados por la clase array de JavaScript, como filter y map, para iterar y manipular los registros. Este código parece mucho más limpio. **Ya no esperamos las llamadas una por una, pero ahora estamos haciendo demasiado a la vez.**

Los métodos filter y map devuelven un nuevo array. Así que aplicamos el filtro a todos los registros, luego mapeamos todos los registros, y finalmente realizamos todas las llamadas de actualización y esperamos a que Promise.all se complete. Esto podría estar bien, pero hay algunos problemas potenciales.

- Primero, podríamos saturar el destino con demasiadas llamadas a la vez.
- Seguimos pasando mucho tiempo sin hacer nada más que esperar E/S.

Procesamiento de streams

Streaming es un término sobrecargado. Lo usamos a nivel de servicio cuando hablamos de canales basados en streams, como AWS Kinesis. También usamos este término a nivel de función cuando necesitamos procesar un stream de datos, como leer o escribir un archivo grande. Estos dos niveles se unen en nuestras funciones de procesamiento de streams, un micro-lote a la vez.

`Highland.js` preserva el estilo FRP con características como filter, map y reduce, y añade características avanzadas como contrapresión, agrupamiento por lotes, procesamiento paralelo y más.

```
import _ from 'highland';
export const handler = async (event) => {
  await _(event.Records)
     .filter(onEventType)
     .map(toUpdateRequest)
     .map((updateRequest) => _(update(updateRequest))
     .parallel(4)
     .toPromise(); // consume
  return 'Success';
}
```

En este ejemplo, **algo clave a destacar es que el código que vemos aquí solo ensambla el pipeline de pasos (es decir, objetos stream) por los que fluirán los datos**. Una buena analogía es la plomería de una casa. Los plomeros ensamblan las tuberías cuando se construye la casa, pero el agua no fluye a través de ellas hasta que abrimos un grifo.

La función constructora del stream de `Highland.js` inicializa el pipeline envolviendo el array de registros en un objeto stream, `_(event.Records)`. Luego, declaramos los pasos restantes que procesarán los datos a medida que fluyan por el pipeline.

Esto incluye el nuevo paso `parallel` que nos permite controlar fácilmente la cantidad de E/S que realizamos a la vez. El paso final, `toPromise`, entonces, y solo entonces, comienza a consumir datos del stream a nivel de función, y los datos empiezan a fluir a través de los pasos de procesamiento. **En otras palabras, comenzamos a extraer datos a través de los pasos después de haber ensamblado el pipeline.**

Esta es una distinción importante. **Cada paso extrae trabajo del paso anterior,** en lugar de recorrer los datos en bucle y enviarlos downstream. Esto nos da control total sobre cómo y cuándo fluyen los datos.

Este es un ejemplo de **Staged Event-Driven Architecture (SEDA) a nivel de función**, como cubrimos en la sección "Vivir en un mundo de consistencia eventual".

- Cada paso representa una etapa en el pipeline de procesamiento.
- Cada etapa tiene un búfer de entrada y salida proporcionado por la biblioteca de streaming de bajo nivel.
- Las etapas upstream pueden continuar extrayendo trabajo cuando las etapas downstream están bloqueadas.

Esto será muy importante cuando hablemos de la E/S asíncrona sin bloqueo en la sección "Optimización del rendimiento".

Creación de un stream

Usaremos esta biblioteca en nuestros ejemplos restantes. Puede encontrar la biblioteca open source aws-lambda-stream aquí: https://github.com/jgilbert01/aws-lambda-stream.

Lo primero que debemos hacer en nuestros procesadores de streams es crear un stream a partir de los registros entrantes. Los distintos canales de streaming y mensajería tienen sus propios formatos. Queremos desacoplar la lógica del procesador de streams de la elección de estas tecnologías.

El primer paso de un procesador de streams transforma los registros entrantes, usando la función `from` apropiada, como `fromKinesis`, `fromDynamodb` y `fromEventBridge`, para normalizar los registros al formato estándar del sobre del evento.

```
import { fromKinesis, ... } from 'aws-lambda-stream';
export const handler = async (event) =>
  fromKinesis(event)
    .filter(onEventType)
    .map(toUpdateRequest)
    .through(update({ parallel: 4 }))
    .through(toPromise);
```

La función `fromKinesis` entiende la estructura de los registros de Kinesis y convierte el array `event.Records` en un stream a nivel de función de eventos. Los pasos restantes no saben que los eventos provienen de Kinesis.

Unidad de trabajo

Más que nunca, no usamos variables globales en los procesadores de streams. En cambio, empleamos estructuras de datos inmutables en FRP y en el procesamiento de streams. **Los datos que pasamos entre los pasos del pipeline deben ser inmutables.** Esto asegura que las distintas etapas no interfieran entre sí.

Para lograr la inmutabilidad, pasamos objetos de unidad de trabajo (uow) a través del pipeline. **Piense en un objeto uow como un objeto inmutable que representa el ámbito léxico de un conjunto de variables que pasan por el stream.** Los pasos del procesador añadirán sus resultados al uow para su uso por pasos downstream.

Las distintas funciones `from` devuelven un stream de objetos de unidad de trabajo.

```
interface UnitOfWork {
  record: any;
  event?: Event;
  batch?: UnitOfWork[];
}
```

Una unidad de trabajo comienza con los siguientes campos:

- El campo `record` contiene el registro original del canal de mensajería.
- El campo `event` contiene el sobre del evento estándar que extrajimos del registro original.
- El campo opcional `batch` contiene un array de unidades de trabajo relacionadas que deben tener éxito o fallar juntas. Veremos cómo usar este campo cuando hablemos del agrupamiento por lotes en la sección "Optimización del rendimiento".

El ámbito que proporciona una unidad de trabajo será crucial cuando aprovechemos las diversas características de procesamiento paralelo y pipeline del framework de procesamiento de streams.

Filtrado y multiplexación

Lograr una topología óptima puede ser un acto de equilibrio. Inevitablemente necesitaremos multiplexar diferentes tipos de eventos a través del mismo canal, como veremos en la sección "Optimización del rendimiento".

El filtrado desacopla un procesador de streams de estas decisiones. Descarta los tipos de eventos no deseados. Especificamos los tipos de eventos de interés, y el procesador ignora los demás.

A medida que la topología del canal de mensajería evoluciona, simplemente reasignamos la función a un canal diferente. **No necesitamos actualizar los filtros ni ninguna otra lógica en el procesador de streams.**

El siguiente bloque de código define el filtro `onEventType` que usamos en nuestra función `listener` de ejemplo. Usa una expresión regular para coincidir con todos los tipos de eventos para la entidad de dominio Thing.

```
const onEventType = uow =>
    uow.event.type.match(/thing-*/);
```

Una función `filter` puede ser arbitrariamente compleja. Es útil crear capas de filtros. El primer filtro descarta los tipos de eventos no deseados, y el siguiente filtro despacha eventos a diferentes pipelines según el contenido del evento. Un pipeline individual puede tener sus propios filtros.

Mapeo

Transformar datos de un formato a otro es el trabajo principal de un procesador de streams.

Una función `listener` consume eventos de dominio upstream y crea vistas materializadas para que las consultas de los usuarios finales no tengan que realizar transformaciones repetidamente.
Una función `trigger` transforma cambios de estado local en eventos de dominio para el consumo downstream.

El siguiente bloque de código muestra la forma básica de un paso de mapeo. El uow debe ser inmutable, por lo que devolvemos un nuevo objeto uow clonando el uow original con el operador spread (…) y añadiendo las variables adicionales:

```
.map((uow) => ({
  ...uow,
  variableName: {
    // lógica de mapeo aquí
  }
}))
```

El siguiente bloque de código define la función de mapeo `toUpdateRequest` que usamos en nuestra función listener de ejemplo. Es responsable de preparar los datos para el paso conector. Transforma el evento de dominio al formato requerido y lo asigna a la variable `updateRequest` que el paso conector downstream espera:

```
import { updateExpression } from 'aws-lambda-stream';
const toUpdateRequest = (uow) => ({
  ...uow,
  updateRequest: {
    Key: { pk: uow.event.thing.id, sk: 'thing' },
    ...updateExpression({
      ...uow.event.thing,
      timestamp: uow.event.timestamp,
    }),
  }
});
```

Hay muchas oportunidades para reutilizar funciones utilitarias, como la función `updateExpression`, que transforma un objeto plano en una expresión de actualización de DynamoDB.

Es importante realizar el mapeo en un paso upstream separado del paso conector que realizará el trabajo de E/S asíncrona sin bloqueo. **Este es un ejemplo de programación cooperativa,** que ayuda a maximizar el potencial de procesamiento concurrente mientras nuestros conectores aguardan una respuesta de recursos externos.

Conectores

Al final de un procesador de streams, hay un paso sink que registra el resultado del procesamiento de una unidad de trabajo. Este paso sink puede persistir datos en una vista materializada, llamar a un servicio externo o publicar un nuevo evento de dominio.

Envolvemos estas llamadas en clases conector delgadas para que podamos crear mocks fácilmente para las pruebas.

El siguiente bloque de código muestra la función `update` reutilizable que estamos usando en nuestra función listener de ejemplo. Envuelve un conector para DynamoDB y llama a la operación `updateItem` para escribir datos en una tabla.

```
import { update, ... } from 'aws-lambda-stream';
export const handler = async (event) =>
...
  .through(update({ parallel: 4 }))
  .through(toPromise);
```

Estas funciones reutilizables aprovechan el currying para sobreescribir configuraciones predeterminadas, como `batchSize` y el número de ejecuciones paralelas.

A continuación se muestra otro ejemplo de una función trigger. Este procesador de streams consume eventos de cambio de un stream de DynamoDB y usa la función reutilizable `publish` para enviar eventos de dominio al event hub:

```
import { publish, ... } from 'aws-lambda-stream';
export const handler = async (event) =>
  fromDynamodb(event)
    .map(toEvent)
    .through(publish({ batchSize: 10 }))
    .through(toPromise);
```

Es importante señalar que los procesadores de streams están realizando acciones atómicas, como discutimos en la sección "Vivir en un mundo de consistencia eventual". Cada uno actualiza un único recurso.

Diseño para la tolerancia a fallos

Tarde o temprano, nuestros procesadores de streams encontrarán problemas.

Contrapresión y limitación de tasa

La contrapresión es una característica importante del procesamiento de streams. **Un procesador de streams no debe sobrecargar su recurso destino.**

- En el mejor caso, el destino lanzará errores de throttling y tendremos bajo rendimiento, mientras el procesador de streams desperdicia tiempo y recursos.
- En el peor caso, podríamos terminar sin rendimiento alguno si la función del procesador agota el tiempo y crea un bucle de reintentos infinito.
- En el peor caso, el procesador de streams podría saturar el sistema destino y causar una interrupción.

La función procesará todos los registros a la vez sin considerar la capacidad de rendimiento del sistema destino. Si enviamos demasiadas solicitudes demasiado rápido, el destino podría sobrecargarse.

El procesamiento de streams, en cambio, proporciona contrapresión natural, porque está orientado a la extracción. En otras palabras, un paso solo extrae la siguiente unidad de trabajo después de haber completado la actual. Esto significa que un recurso destino lento naturalmente frenará el flujo, basándose únicamente en su propia latencia.

Sin embargo, los recursos cloud-native, como AWS DynamoDB, pueden procesar solicitudes con un rendimiento extremadamente alto. **Estos recursos dependen del throttling para restringir la capacidad.** En este caso, la contrapresión natural no es suficiente. Necesitaremos insertar contrapresión artificial.

```
export const handler = async (event) =>
    ...
    .rateLimit(2, 100) // 2 por cada 100ms
    .flatMap(makeSomeAsyncCall)
    ...
```

La característica `rateLimit` de `Highland.js` actúa como un regulador para ralentizar el flujo de trabajo a través del procesador de streams. En este ejemplo, llamamos al destino no más de dos veces cada 100 milisegundos. **Deberá conocer los límites de capacidad del recurso específico para calcular la tasa adecuada.**

Eventos envenenados

Algunos errores son transitorios y se autocorrigen. Por ejemplo, los problemas de red simplemente necesitan un reintento simple. Por otro lado, un error transitorio podría ser algo más grave, como una interrupción en el sistema destino de un procesador de streams. En este caso, el procesador puede dejar que los eventos se acumulen en la cola y reintentar un lote completo hasta que el sistema destino se recupere.

Los eventos envenenados, en cambio, no son recuperables. Un evento envenenado contiene datos que rompen el procesador de streams consumidor. Puede ser un bug en el procesador consumidor o en el productor. Independientemente, **el procesador de streams consumidor continuará reintentando el lote completo a menos que tomemos acciones correctivas.**

Los servicios cloud que aprovechamos pueden ofrecer soporte para estos escenarios. Por ejemplo, AWS Lambda proporciona una característica de bisección que ofrece cierto alivio. Esto reducirá el tamaño del lote a la mitad antes de reintentar.

Desde aquí, podemos enviar el evento envenenado a una Dead Letter Queue (DLQ) para procesamiento especial. Sin embargo, cuando hay un evento envenenado, usualmente hay muchos, por lo que este proceso continúa. Mientras tanto, la latencia adicional hace que más y más eventos buenos se acumulen en el stream.

Podemos hacerlo mejor. **Podemos hacer que la lógica de nuestro procesador de streams sea más proactiva, manejando los eventos envenenados y emitiendo eventos de fallo.**

Eventos de fallo

Los procesadores de streams emiten eventos de fallo cuando encuentran condiciones de error esperadas. **Un fallo es un tipo de evento que contiene un error y el evento o eventos correspondientes que causaron el error.**

Luego, aprovechamos el event hub y todas sus características de enrutamiento como la DLQ, en lugar de crear y mantener algo especial.

Los eventos de fallo fluyen al event lake como cualquier otro. Downstream, un servicio de monitoreo de fallos recopila los fallos, lo que significa que podemos reenviar los eventos afectados después de resolver la causa raíz.

```
export const FAULT_EVENT_TYPE: string = 'fault';
interface FaultEvent extends Event {
  err: {
    name: string;
    message: string;
    stack: string;
  };
  uow: UnitOfWork;
}
```

Cuando un procesador de streams encuentra un error, **el error saltará todos los pasos restantes hasta que sea manejado o llegue al final del stream.** Si llega al final del stream, entonces todo el procesamiento se detiene y la función devuelve el error. En este punto, el lote reintentará, como discutimos en la sección "Eventos envenenados".

En cambio, queremos escribir código proactivo que maneje los eventos envenenados explícitamente. Los apartaremos como eventos de fallo para que los eventos buenos puedan continuar fluyendo. Luego, podemos delegar el procesamiento de los eventos de fallo a otros componentes.

Reenvío

Nuestros procesadores de streams ahora manejan proactivamente los eventos envenenados y producen eventos de fallo para que los eventos buenos sigan fluyendo. **Un servicio de monitoreo de fallos** consume todos los eventos de fallo del event bus y los almacena en almacenamiento de objetos.

Una vez que hayamos resuelto la causa raíz de un fallo, podemos reenviar el evento original al procesador de streams que generó el fallo, asegurando que todos los eventos se procesen exitosamente.

Es importante reconocer que ahora estamos procesando estos eventos fuera de orden, ya que necesitamos apartarlos. **Esta es una razón más por la que es necesario asegurar que la lógica de nuestros procesadores de streams sea tolerante al orden.**

Optimización del rendimiento

El rendimiento es de la mayor importancia para un procesador de streams. ¿Está un procesador de streams específico manteniendo el ritmo de los eventos entrantes o está quedándose atrás? ¿Está el procesador sobrecargado o está pasando demasiado tiempo esperando? ¿Estamos poniendo demasiada carga sobre el recurso destino? Hay muchas formas en que podemos optimizar el rendimiento para asegurar que no estemos ni sobrecargando ni desperdiciando recursos.

Parámetro de tamaño de lote

El tamaño de lote es uno de los principales parámetros que tenemos para ajustar el rendimiento de un procesador de streams. Controla cuánto trabajo entregamos a un procesador en cada invocación.

- ¿Le damos más trabajo y dejamos que se ejecute por más tiempo,
- o le damos menos trabajo e invocamos con más frecuencia?
- ¿Cómo impacta esto el costo del procesador de streams?

A medida que defina y evolucione su topología de canales, necesitará conocer el volumen esperado de tráfico para cada canal. **Para cada procesador de streams, necesitará entender las características del recurso destino.**

Desde aquí, necesitará experimentar con diferentes tamaños de lote.

- A medida que mejore la eficiencia de un procesador, puede ser capaz de aumentar el tamaño de lote.
- Si está multiplexando muchos tipos de eventos a través de un único canal, un tamaño de lote mayor ayuda a asegurar que su procesador tenga trabajo que realizar en cada invocación.
- A medida que cambie el número de shards, es posible que también quiera cambiar el tamaño de lote.
- A medida que aumente el tamaño de lote, es posible que también necesite aumentar el timeout de la función.

Alcanzar la topología de canal y los tamaños de lote del procesador de streams óptimos es un acto de equilibrio iterativo. Tenga en cuenta que la solución óptima también considera el costo.

Es un acto de equilibrio. Me gusta comenzar con un único canal por subsistema y un tamaño de lote predeterminado de 100. Luego, analizo las métricas de antigüedad del iterador y las métricas de invocaciones mientras ajusto la mejor configuración.

E/S asíncrona sin bloqueo

A nivel de servicio, un procesador de streams representa una etapa en un flujo de proceso de consistencia eventual. A nivel de función, un procesador de streams implementa su propio flujo de proceso por etapas.

Para optimizar el rendimiento, necesitamos mantener estos pasos ocupados. No queremos privar de trabajo a los pasos downstream, y no queremos que el resultado de los pasos upstream tenga que esperar. **Aquí es donde la E/S asíncrona sin**

bloqueo juega un papel muy importante.

Como vimos en la sección "Elección de un paradigma de programación", los pasos en un procesador de streams que realizan una llamada externa (es decir, E/S) son típicamente los más costosos en tiempo.

No queremos desperdiciar tiempo bloqueando en estas llamadas. Queremos mantener los otros pasos ocupados mientras esperamos una respuesta. Queremos que los pasos upstream continúen trabajando en el lote y preparando más llamadas externas, y queremos que los pasos downstream continúen con su trabajo.

```
export const handler = async (event) =>
    ...
    .map(makeSomeAsyncCall)
    .parallel(8)
    ...
```

Mientras espera una respuesta, la característica `parallel` de Highland.js continúa extrayendo trabajo a través de los pasos upstream hasta que ha alcanzado el número especificado de llamadas. Por ejemplo, un paso de mapeo upstream puede preparar la entrada para la siguiente solicitud, de modo que podamos hacer inmediatamente otra llamada cuando recibamos una respuesta a una llamada anterior.

La E/S asíncrona sin bloqueo es probablemente la característica más importante para optimizar el rendimiento. **Este es usualmente el primer parámetro que ajusto cuando afino un procesador de streams.**

Pipelines y multiplexación

Para lograr una topología óptima, inevitablemente necesitamos multiplexar diferentes tipos de eventos a través del mismo canal. Esto en realidad juega a nuestro favor porque nos ayuda a mantener nuestros procesadores ocupados mientras esperan las llamadas asíncronas.

Cada servicio es responsable ante un único actor, pero cada uno necesita reaccionar a múltiples tipos de eventos para soportar su funcionalidad. Sin embargo, no es práctico mantener un canal y una función separados por tipo de evento. **Por tanto, necesitamos una forma de mantener una buena separación mientras procesamos múltiples tipos de eventos en el mismo procesador de streams.**

Los pipelines nos proporcionan el mecanismo para lograr esta separación limpia dividiendo la lógica de procesamiento en flujos separados. Bifurcamos el stream maestro en muchos pipelines y los unimos al final. Cada pipeline define un conjunto de pasos relacionados que aplican a un tipo y contenido de evento específico.

```
const pipeline1 = (options) => // inicializar
  (stream) => stream           // ensamblar
    .filter(onEventType)
    .map(toUpdateRequest)
    .through(update({ parallel: 4, ...options }));
...
export default pipeline1;
```

Un pipeline típicamente comienza con uno o más pasos de filtrado que dictan qué tipos de eventos aplican al pipeline y fluirán a través de los pasos restantes. Implementamos cada pipeline en su propio archivo de módulo para crear separación limpia, y para poder probarlos de forma independiente.

Ahora necesitamos ensamblar un conjunto de pipelines en un procesador de streams.

```
import { initialize, defaultOptions, ... }
  from 'aws-lambda-stream';
import pipeline1 from './pipeline1';
import pipeline2 from './pipeline2';
const PIPELINES = { pipeline1, pipeline2 };
const OPTIONS = { ...defaultOptions, ... };
export const handler = async (event) =>
  initialize(PIPELINES, OPTIONS)
    .assemble(fromKinesis(event))
    .through(toPromise);
```

Desde aquí, los pipelines toman el control y realizan su trabajo. Si un pipeline está esperando llamadas asíncronas, los otros pipelines continuarán extrayendo trabajo hasta que ellos también necesiten esperar. **Este enfoque de programación cooperativa nos ayuda a minimizar el tiempo que un procesador de streams pasa sin hacer nada más que esperar.**

Patrones de pipeline

Es importante ensamblar un conjunto cohesivo de pipelines en una única función.

A medida que construimos servicios autónomos, se hace evidente que tenemos patrones de pipeline repetibles que usamos una y otra vez, con diferencias consistentes. Por ejemplo, una función listener en un servicio BFF es responsable de consumir eventos de dominio y crear las vistas materializadas utilizadas en las consultas.

Todos los pipelines siguen los mismos pasos. Filtran por tipos de eventos específicos, mapean los datos a una solicitud de actualización y guardan los datos en el almacén de datos. Solo los detalles de los pasos de filtrado y mapeo varían entre los pipelines.

Podemos empaquetar estos patrones en pipelines reutilizables y configurarlos con reglas (es decir, opciones).

```
import { materialize } from 'aws-lambda-stream';
import { toThingUpdateRequest } from './model';
const RULES = [{
    id: 'p1',
    pattern: materialize,
    eventType: /thing-*/,
    toUpdateRequest: toThingUpdateRequest,
  },
  {
    id: 'p2',
    pattern: materialize,
    eventType: 'something-else',
    toUpdateRequest: (uow) => ({ ...uow, ... }),
  }];
export default RULES;
```

- El campo `id` contiene un nombre único para la instancia del pipeline.
- El campo `pattern` apunta a la función que implementa el patrón del pipeline. En este caso, usamos la función de pipeline `materialize` proporcionada por la biblioteca.
- El campo `eventType` contiene una expresión regular, cadena o array de cadenas utilizadas para filtrar por tipo de evento.
- El campo `toUpdateRequest` es específico del patrón `materialize`. Este campo apunta a la función de mapeo que el pipeline llamará en el paso de mapeo para formatear una solicitud de actualización de ítem de DynamoDB.

Puede definir la función en línea, pero es mejor implementarla y probarla por separado.

```
import { initializeFrom, ... } from 'aws-lambda-stream';
...
import RULES from './rules;
const PIPELINES = {
  ...initializeFrom(RULES),
  pipeline2,
};
```

La función `initializeFrom` itera sobre las reglas y prepara una instancia de pipeline para cada una, usando la función de pipeline especificada en el campo `pattern`.

Sharding

La Ley de Little nos muestra que el trabajo pasa más tiempo en un sistema a medida que el sistema se acerca a su capacidad máxima. En última instancia, el sistema se quedará atrás cuando la tasa de llegada supere la tasa de servicio. La ley también nos muestra que el paralelismo tiene un impacto significativo en el tiempo que se pasa en el sistema.

Aquí es donde el sharding entra en juego con el procesamiento de streams. El sharding esencialmente divide un canal de entrada (es decir, stream) en múltiples canales de salida (es decir, shards). Una instancia separada de un procesador de streams consume de cada shard, lo que incrementa el paralelismo y el rendimiento del sistema.

El campo `partitionKey` en el sobre del evento juega un papel importante en el sharding. Un algoritmo de hashing sobre el valor de la clave de partición determina por qué shard fluirá un evento. La distribución de las claves de partición dicta qué tan bien utilizamos todos los shards.

Debemos inclinarnos por usar un identificador de entidad de dominio como clave de partición. Estos identificadores usualmente contienen un UUID aleatorio V4, **lo que ayuda a garantizar una distribución uniforme entre los shards**. Esto también asegura que **los eventos relacionados para una instancia de entidad de dominio fluirán a través del mismo shard.**

Agrupamiento por lotes y por grupos

El agrupamiento por lotes y por grupos son similares porque reducen el número de llamadas que un procesador realiza al recurso destino. Esto reduce el tiempo que un procesador pasa esperando la latencia de red y también reduce la cantidad de trabajo que el destino necesita realizar.

En el caso del agrupamiento por grupos, puede reducir significativamente la carga sobre el destino.

Una unidad de trabajo debe tener éxito o fallar en conjunto. En la sección "Diseño para la tolerancia a fallos", también adjuntamos la unidad de trabajo a un evento de fallo para poder reenviar la unidad de trabajo. En general, cada evento entrante representa una unidad de trabajo.

Procesamiento por lotes

Antes de poder implementar el procesamiento por lotes, primero necesitamos determinar si el recurso destino lo soporta. Muchos servicios, como Amazon Kinesis y DynamoDB, proporcionan tanto una interfaz de lotes como una que no lo es.

Añadimos un paso que recopila unidades de trabajo. Cuando alcanza el límite, el paso pone el lote a disposición del siguiente paso para realizar la llamada al recurso destino.

```
import { toBatchUow, ... } from 'aws-lambda-stream';
export const handler = async (event) =>
    ...
    .batch(10)
    .map(toBatchUow)
    .map(makeSomeAsyncCall)
    .parallel(4)
    ...
```

La característica `batch` de Highland.js recopila el lote, y la función utilitaria `toBatchUow` formatea la unidad de trabajo del lote para que podamos generar fácilmente un fallo para el lote completo.

Es importante determinar si la API trata el lote como una única transacción o si las entradas individuales del lote pueden fallar de forma independiente. Si pueden fallar de forma independiente, entonces necesitamos lógica adicional para reenviar las entradas fallidas. Esta lógica puede ser tediosa, **por lo que puede ser mejor determinar primero si el procesamiento por lotes ofrece mejoras suficientes.**

Agrupamiento

Multiplexar diferentes tipos de eventos a través de un canal nos permite agrupar eventos relacionados y reducir el número de llamadas que realizamos contra el recurso destino. Por ejemplo, podríamos agrupar eventos basados en una propiedad común, como un número de cuenta, y simplemente usar el último evento como entrada para la llamada.

```
import { toGroupUow, ... } from 'aws-lambda-stream';
export const handler = async (event) =>
    ...
    .group(uow => uow.event.partitionKey)
    .flatMap(toGroupUows)
    .map(makeSomeAsyncCall)
    .parallel(4)
    ...
```

La característica `group` de Highland.js reduce los datos en grupos basados en el valor de un campo, y la función utilitaria `toGroupUows` formatea cada grupo como una unidad de trabajo para poder generar fácilmente un fallo para el grupo completo. Nuevamente, podemos combinar el agrupamiento con la característica `parallel` para ayudar a asegurar que el procesador no espere estas llamadas.