# Apache Flink

Contenido

[Apache Flink 1](#_Toc212105251)

[1. Introducción y Arquitectura Fundamental 1](#_Toc212105252)

[El desafío del streaming: Por qué Flink 1](#_Toc212105253)

[Arquitectura Flink 3](#_Toc212105254)

[Flujo de datos: Grafo de tareas (DAG), Particiones de stream y Shuffle 4](#_Toc212105255)

[Tipos de despliegue y el ciclo de vida del job 6](#_Toc212105256)

[2. Gestión de Estado y Tolerancia a Fallos 7](#_Toc212105257)

[Conceptos de Estado y Memoria 7](#_Toc212105258)

[Checkpointing y Snapshots 8](#_Toc212105259)

[Garantía Exactly Once (El Protocolo Two-Phase Commit) 10](#_Toc212105260)

[State Backends Avanzados 10](#_Toc212105261)

[3. Dominando el Tiempo y la Corrección del Stream 10](#_Toc212105262)

[Control del Tiempo: Processing Time vs. Event Time 10](#_Toc212105263)

[Gestión del Desorden: Watermarks (Marcas de Agua) 11](#_Toc212105264)tumblin

[Ventanas de Procesamiento y Tolerancia a la Latencia 12](#_Toc212105265)

[Timers y Lógica Avanzada 12](#_Toc212105266)

[4. Integración y Patrones de Transformación 13](#_Toc212105267)

[Conexión con Kafka (Serialización y Formatos) 13](#_Toc212105268)

[La API Unificada: DataStream vs. Table API/SQL 14](#_Toc212105269)

[Patrones de Join en Streaming (Casos de Uso) 15](#_Toc212105270)

[La Dualidad de Almacenamiento: Integración con Delta Lake 15](#_Toc212105271)

[Caso práctico de integración: kafka a postgresql 16](#_Toc212105272)

[5. Testing, Operaciones y Optimización 19](#_Toc212105273)

## Introducción y Arquitectura Fundamental

### El desafío del streaming: Por qué Flink

El principal problema en el procesamiento de datos a gran escala y en tiempo real no es la velocidad, sino la **consistencia y la corrección** del resultado. Apache Flink nace para resolver el dilema que las generaciones anteriores de motores de Big Data no lograron superar.

Históricamente, los motores de *streaming* se vieron forzados a comprometer la precisión por la velocidad, cayendo en uno de estos tres modos:

|  |  |  |
| --- | --- | --- |
| Motor | Modo de procesamiento | Fallo principal |
| Hadoop/MapReduce | Batch | Latencia alta: Incapaz de procesar datos en tiempo real |
| Spark Streaming (primeras versiones) | Micro-batch | Resultados incorrectos: no gestionaba correctamente el tiempo de evento real |
| Alternativas simples | At Least Once | Duplicación de datos: en caso de fallo, los datos se reprocesan y se duplican |

Ningún motor ofrecía una solución que fuera rápida y con una garantía estricta de precisión.

Apache Flink fue diseñado desde cero como un verdadero motor de streaming evento-por-evento, atacando los problemas de inconsistencia en tres frentes:

1. Problema de precisión (**exactly once**)
   1. Objetivo: garantizar que cada evento sea procesado exactamente una vez, sin pérdidas ni duplicados, incluso ante fallos de hardware o red
   2. Mecanismos de Flink:
      1. **Checkpoints** asíncronos: Flink toma instantáneas globales y atómicas del estado de la aplicación
      2. **Two-Phase Commit** (2PC): Coordina estas instantáneas con los destinos de datos, asegurando que los resultados solo se hacen permanentes y visibles cuando el checkpoint es globalmente exitoso.
2. Problema de desorden (**corrección temporal**)
   1. Objetivo: producir resultados correctos sin importar si los eventos llegan a Flink tarde o fuera de orden.
   2. Mecanismo de Flink:
      1. **Event Time**: Flink utiliza el timestamp real incluido en el dato, no el reloj de su servidor.
      2. **Watermarks**: estas señales de progreso le indican a Flink cuándo debe cerrar una ventana de tiempo. Esto permite al sistema “esperar” un tiempo razonable a que lleguen los eventos desordenados, garantizando la corrección de las ventanas.
3. Problema de la **escalabilidad del estado**
   1. Objetivo: poder mantener la “memoria” de la aplicación (el estado) a escala de terabytes sin sacrificar la baja latencia.
   2. Mecanismo de Flink:
      1. **Keyed State Local**: el estado se mantiene en el nodo que procesa la clave, permitiendo un acceso a velocidad de memoria RAM.
      2. Gestión de memoria independiente: Flink utiliza memoria Off-Heap (fuera del control del garbage collector de java) y State Backends avanzados como RocksDB para gestionar grandes volúmenes de estado en disco sin sufrir pausas de latencia

### Arquitectura Flink

Los dos componentes principales de un clúster de apache Flink son:

* **Job Manager**: coordinador. Coordina y envía trabajos a los Task Managers además de controlar que realicen bien los trabajos. También coordina el estado del sistema a través de snapshots. Periódicamente ejecuta checkpoints y los registra con los snapshots. Estos snapshots se almacenan fuera de flink.
* **Task Manager**: ejecutores. Generalmente se ejecuta un Task Manager por cada nodo del clúster. Una vez iniciados, los task managers se registran en el job manager y ofrecen su capacidad computacional para ejecutar aplicaciones de Flink.
* Además de estos dos componentes cabe destacar un concepto clave para entender la ejecución paralela en Flink, los **Task Slots.** Un Task Slot es la unidad de concurrencia más pequeña que puede configurarse en Flink. Representa una porción fija de recursos (principalmente memoria) que cada Task Manager reserva para la ejecución de subtareas. Los Task Managers publicitan su capacidad de trabajo a través de Task Slots.

La cantidad total de Task Slots en el clúster define el **paralelismo máximo** que un *job* de Flink puede alcanzar. Por ejemplo, si un clúster tiene 4 Task Managers con 2 slots cada uno, el paralelismo máximo es 8.

Flink optimiza significativamente el rendimiento gracias al **Task Chaining** (encadenamiento de tareas). Este mecanismo ejecuta múltiples operadores consecutivos (como source →map → filter) con un flujo de datos directo ("*one-to-one*") en el **mismo Task** Slot y dentro del **mismo hilo de procesamiento**.

Esto elimina la serialización y la transferencia de datos a través de la red entre procesos, lo que reduce la latencia.

El encadenamiento se mantiene mientras el flujo de datos sea local. La **causa principal y más común para romper una cadena de tareas es una operación de *shuffle***. Un *shuffle* (como el que se produce tras una operación de **keyBy()** o groupBy() requiere que los datos sean repartidos y reorganizados entre los Task Slots, obligando a una transferencia de datos a través de la red y, por lo tanto, forzando la **creación de un nuevo *Task* y la ruptura del *chaining***.

Un solo Task Slot puede ser responsable de ejecutar una cadena completa de operadores siempre que el flujo de datos entre ellos no requiera este costoso intercambio de datos entre Tasks.

Para la comunicación entre el Job Manager y los Task Managers, Flink utiliza internamente el sistema de actores Akka. Este sistema está ordenado jerárquicamente. El Job Manager se comunica con el Task Manager y este con el TaskSlot. Toda esta comunicación se hace en llamadas RPC.

Este sistema de actores es ideal para Flink porque es asíncrono:

* El JobManager no se bloquea. Envía Х órdenes a У TaskManagers a la vez.
* No tiene que esperar la respuesta de uno para enviar la orden al siguiente
* Cuando un TaskManaager termina envía un mensaje de tarea completada
* Si se cae un TaskManager, el JobManager se da cuenta porque no responde a los mensajes (timeouts) y puede reasignar esa tarea a otro TaskManager, haciéndolo muy tolerante a fallos.

compone una jerarquía de procesos comunicados por mensajes en llamadas RPC. El estado y el comportamiento de los actores viene determinado por su estado y por los mensajes que ha recibido

Estos componentes (JobManaager y TaskManager) de Flink son procesos JVM que ejecutan las tareas en uno o más hilos de procesamiento.

Diagrama

El contenido generado por IA puede ser incorrecto.

### Flujo de datos: Grafo de tareas (DAG), Particiones de stream y Shuffle

Este capítulo detalla cómo el código de tu aplicación se convierte en un plan ejecutable y cómo los datos se distribuyen a través del clúster.

**1. El Flujo de Datos: El Grafo Dirigido Acíclico (DAG)**

Una aplicación de Flink no se ejecuta como un único programa, sino como una secuencia de pasos interconectados.

* Definición**:** El Grafo Dirigido Acíclico **(DAG)** es la representación lógica de tu aplicación. Muestra las transformaciones de datos como nodos (operadores) y el flujo de datos como flechas dirigidas (de la Fuente al Destino).
* Transformación**:** El JobManager toma tu código (por ejemplo, una cadena de filter().map().keyBy().window()) y lo convierte en este grafo ejecutable.
* **Acíclico:** Es crucial que sea "Acíclico" (sin ciclos), lo que significa que el flujo de datos siempre va hacia adelante; no puede volver a un operador anterior.

**2. Particiones de Stream y Subtareas**

Para que el procesamiento sea paralelo, Flink divide tanto el *stream* como los operadores:

* **Particiones de Stream:** El *stream* de datos se divide en una o más particiones lógicas. Estas particiones se consumen de forma independiente por las subtareas.
  + *Ejemplo:* Si lees un *topic* de Kafka con 4 particiones, tu *stream* tendrá al menos 4 particiones lógicas.
* **Subtareas de Operador:** Cada operador de tu DAG (el map, keyBy, sink) se divide en una o más subtareas que se ejecutan en los Task Slots del *clúster*.
  + El número de subtareas está definido por el **paralelismo** del operador.

**3. El Movimiento de Datos: Tipos de Conexión**

El movimiento de datos entre un operador (aguas arriba) y el siguiente (aguas abajo) se realiza a través de dos patrones de conexión principales:

**A. Conexión One-to-One (Forwarding) ⚡**

Esta es la forma más rápida y se usa para optimizar operaciones ligeras.

* Función**:** La subtarea $X$ del operador $A$ **siempre** envía datos a la subtarea $X$ del operador $B$.
* Ventajas**:**
  + Mantiene el Orden**:** El orden de los eventos se mantiene.
  + **Baja Latencia:** Permite el **Task Chaining**, donde ambos operadores se ejecutan en el mismo hilo de procesamiento y en el mismo **Task Slot**. Esto evita la serialización/deserialización y la comunicación de red.
* **Operadores Típicos:** map(), filter(), y flatMap().

**B. Conexión Redistribuida (Shuffle) 🌐**

Esta conexión es más costosa, ya que **mueve datos a través de la red**, pero es fundamental para el procesamiento con estado.

* Función**:** La subtarea $X$ del operador $A$ puede enviar datos a cualquier subtarea del operador $B$.
* ¿Cuándo Ocurre el Shuffle? Se desencadena en operaciones que requieren que los datos se agrupen o mezclen:
  + **keyBy():** La más común. La partición se realiza aplicando un **hash** a la clave (ID de usuario) para garantizar que todos los eventos de esa clave terminen en la misma subtarea de destino.
  + **rebalance():** Redistribuye la carga de trabajo de forma aleatoria para corregir un desequilibrio (*skew*) en los datos.
* Costo**:** El *shuffle* introduce la latencia de red y el uso de CPU para la serialización y deserialización, pero es el precio a pagar para lograr el estado consistente por clave.

**Resumen del Flujo**

Tu aplicación se ejecuta como un **DAG** en paralelo, donde la mayoría de los operadores están **encadenados** (*one-to-one*) para la velocidad, y solo las operaciones críticas con estado (como keyBy()) fuerzan una costosa **redistribución (*shuffle*)** para garantizar que los datos se procesen con la memoria correcta.

### Tipos de despliegue y el ciclo de vida del job

Este capítulo te prepara para mover tu código del entorno de desarrollo a un *clúster* real. Entender el ciclo de vida y los modelos de despliegue es una habilidad fundamental para cualquier *Data Engineer*.

**1. El Ciclo de Vida del Job de Flink**

Antes de ejecutarse, tu aplicación pasa por un proceso de preparación y ejecución:

1. Programa de Usuario**:** Escribes la lógica en la **DataStream API** o **Table API/SQL** (tu código).
2. Traducción a DAG**:** El Job Manager traduce tu código en el **Grafo de Tareas (DAG)** (el plan lógico de tu aplicación).
3. Despliegue y Asignación**:** El *Job Manager* envía el DAG a los Task Managers para su ejecución. Aquí es donde entra en juego el **Modelo de Despliegue**.
4. Ejecución**:** Las subtareas se ejecutan en los Task Slots de los *Task Managers*.

**2. Modelos de Despliegue de Flink**

La elección del modelo de despliegue afecta cómo se asignan los recursos y el nivel de aislamiento entre los diferentes *jobs*.

**A. Session Cluster**

* Definición**:** El modo de despliegue más simple. Un *clúster* de Flink de larga duración (siempre encendido) que aloja múltiples *jobs*. El *Job Manager* y los *Task Managers* se inician una sola vez y son compartidos.
* Aislamiento**:** Bajo. Si un *job* falla o consume demasiada memoria, puede afectar a todos los demás *jobs* en el *clúster*. Esto es porque el peligro no es que los task managers no estén disponibles, sino que todos los Jobs confían en la misma JVM del task manager:
  + Sin Aislamiento de Memoria: Si el Job A tiene una fuga de memoria (memory leak) o un error de código, puede consumir toda la memoria del Task Slot y del Task Manager. Como el Task Manager es un proceso JVM grande y compartido, el fallo de la memoria del Job A hará que el Task Manager se caiga por completo.
  + Efecto Dominó: Si el Task Manager se cae, todos los Jobs B, C y D que se estaban ejecutando en ese mismo Task Manager (incluso si estaban en Task Slots diferentes) fallan instantáneamente y deben recurrir al checkpoint para reiniciarse.
* Uso Ideal**:** Desarrollo, pruebas rápidas (*testing*), o para ejecutar muchos *jobs* pequeños con baja latencia de inicio. Es barato y rápido de usar.

**B. Job Cluster (El Cluster Dedicado)**

* Definición**:** Un *clúster* se crea **específicamente para un único *job*** y se destruye cuando el *job* termina.
* Aislamiento: Alto. Cada *job* tiene su propio *Job Manager* y *Task Managers* dedicados, lo que garantiza que los fallos no afecten a otros *jobs*.
* Uso Ideal**:** Producción y *jobs* de misión crítica. Proporciona recursos garantizados y el máximo aislamiento.

**3. Integración con Gestores de Recursos**

En producción, Flink no gestiona el *hardware* directamente; se integra con gestores de recursos para orquestar los *Task Managers*.

* **Kubernetes (K8s) y YARN:** El *Job Manager* actúa como un cliente que solicita los contenedores (o recursos YARN) necesarios para sus *Task Managers*. Esto permite:
  + Escalado Dinámico**:** Añadir o eliminar *Task Managers* según la carga de trabajo de forma eficiente.
  + Densidad**:** Optimizar el uso de recursos al asignar los *Task Slots* solo cuando se necesitan.

**4. Alta Disponibilidad (HA) del Job Manager**

Dado que el Job Manager es el coordinador del *checkpointing* y la recuperación, si este falla, todo el *job* se detiene.

* Función**:** El *Job Manager* se configura en modo de **Alta Disponibilidad (HA)**. Un *master* actúa como Líder, y los demás como Seguidores.
* Mecanismo**:** Si el Líder falla, un Seguidor es elegido como nuevo Líder.
* Metadatos**:** El *Job Manager* escribe metadatos críticos (como la ubicación de los *Snapshots* más recientes) en una ubicación tolerante a fallos (como ZooKeeper o, en versiones modernas de Kafka, **KRaft**), asegurando que el nuevo Líder pueda restaurar el *job* sin problemas.

## Gestión de Estado y Tolerancia a Fallos

Este módulo detalla cómo Flink logra ser **Exactly Once**.

### Conceptos de Estado y Memoria

Este capítulo sienta las bases de la memoria de la aplicación, un conocimiento clave para entender la fiabilidad y la baja latencia.

**1. Tipos de Estado en Flink**

El estado es la "memoria" de tu aplicación que se guarda para mantener el contexto entre eventos. Flink lo clasifica en dos tipos principales:

* Estado con Clave **(Keyed State):**
  + Propósito**:** El estado está asociado a una clave específica dentro del *stream* (ej., un user\_id).
  + Mecanismo**:** Solo se puede utilizar en un *stream* que ha sido particionado con keyBy().
  + Ejemplo**:** El contador de clics, la ubicación de un vehículo o el saldo de una cuenta. Flink gestiona millones de estos estados.
* Estado del Operador **(Operator State):**
  + Propósito**:** El estado está asociado a una instancia paralela de una tarea, no a una clave.
  + Uso**:** El caso de uso más común es el **Source Operator** (Conector de Kafka), que utiliza este estado para guardar los **offsets** (posiciones de lectura) de cada partición de Kafka.

**2. El Desafío del Estado Masivo: La Memoria Fuera del Heap**

En aplicaciones de *streaming* real, el estado puede crecer hasta superar la RAM disponible en un solo *Task Manager*.

* El Problema del Heap de la JVM**:** La memoria *Heap* estándar de Java está sujeta al *Garbage Collector* (GC). Si el estado es demasiado grande, el GC causaría pausas (***stop-the-world pauses***) que interrumpirían el flujo de datos y arruinarían la baja latencia.
* Memoria Fuera del Heap **(*Off-Heap Memory*):**
  + Solución**:** Flink asigna y gestiona grandes bloques de memoria fuera del *Heap* de la JVM.
  + Beneficio**:** Como esta memoria no está bajo el control del GC, Flink evita las pausas y garantiza una **latencia predecible y baja**, incluso con grandes volúmenes de estado.

### Checkpointing y Snapshots

Este capítulo explica el mecanismo central que protege el estado de los fallos del *hardware* y garantiza el procesamiento consistente.

**1. Checkpoint vs. Snapshot:** Acción vs. Resultado

* Checkpoint**:** Es la **operación** coordinada por el *Job Manager* que toma una instantánea atómica y global del estado. Es el **mecanismo de la operación**.
* Snapshot**:** Es el **resultado físico** del proceso de *checkpointing*. Es el conjunto de archivos guardados en un almacenamiento tolerante a fallos (S3, HDFS). Es el **punto de restauración**.

**2. El Mecanismo de las Barreras de Checkpoint**

El *Job Manager* utiliza las Barreras de Checkpoint para tomar una instantánea consistente de una aplicación que está en movimiento.

* Inyección en la Fuente**:** El *Job Manager* inyecta una barrera (un evento de control) en el *stream* de entrada.
* Separación de Datos**:** La barrera sirve como un delimitador: los datos antes de la barrera pertenecen a la instantánea actual; los datos después pertenecen a la siguiente.
* Sincronización**:** La barrera viaja a través de todos los operadores.
  + Source**:** Registra el ***Offset* de Kafka** y permite que la barrera pase.
  + Operador con Estado**:** Cuando la barrera llega, el operador guarda su estado local en el *State Backend* y la propaga.
  + Alineación**:** En operadores con múltiples entradas (*joins*), el operador espera a recibir la barrera de todas las entradas antes de propagarla. Mientras espera, amortigua **(*buffer*)** los datos que llegaron temprano, asegurando una instantánea globalmente consistente.

Escala de tiempo

El contenido generado por IA puede ser incorrecto.

**3. Recuperación Ante Fallos**

Si un *Task Manager* falla:

1. El *Job Manager* **descarta** la ejecución actual.
2. Lee el último **Snapshot** exitoso.
3. Reinicia los *Task Managers* y les ordena: **restaurar el estado** (los contadores) y **reiniciar la lectura** de la fuente de datos (Kafka) desde los *offsets* guardados en el *Snapshot*.

### Garantía Exactly Once (El Protocolo Two-Phase Commit)

Este capítulo se centra en el protocolo de transacción que permite a Flink llevar la garantía de consistencia hasta el destino final.

* Problema **Resuelto:** Evitar la duplicación de datos en el Sink (destino), incluso si el *job* falla.
* Mecanismo**:** Flink utiliza el protocolo ***Two-Phase Commit* (2PC)**, coordinado por el *Job Manager* para sincronizar el estado interno de Flink con la escritura externa en el *Sink*.

| **Fase** | **Acción** | **Resultado** |
| --- | --- | --- |
| **Fase 1: Pre-Commit** | El *Sink* escribe los resultados del *checkpoint* en una **transacción tentativa**. | Los datos están escritos en el destino (ej., Kafka/Delta Lake), pero **no son visibles** para los consumidores externos. |
| **Fase 2: Commit** | El *Job Manager* declara que el *checkpoint* es **globalmente exitoso** y envía la señal de **COMMIT** al *Sink*. | Los datos se hacen **visibles y permanentes**. |
| **Fase de Fallo (Rollback)** | Si falla cualquier parte del *checkpoint*, se envía una señal de **ABORT** (cancelar), y la transacción tentativa se **descarta**. | Se garantiza que los datos **no se duplican** en el destino, asegurando la promesa *Exactly Once*. |

### State Backends Avanzados

*pendiente de desarrollar: justificar por qué y cómo* ***RocksDB*** *permite escalar el estado a terabytes en disco, ya que el Off-Heap no es infinito.*

## Dominando el Tiempo y la Corrección del Stream

Este módulo es crucial porque explica cómo Flink produce resultados fiables, incluso cuando los datos llegan desordenados.

### Control del Tiempo: Processing Time vs. Event Time

Este capítulo define qué reloj está usando Flink, un concepto que determina la precisión de todos tus cálculos.

**1. Tiempo de Procesamiento (*Processing Time*)**

* Definición**:** Utiliza el reloj del sistema operativo (el servidor) en el momento en que el evento es procesado por un operador de Flink.
* Ventajas**:** Es el modo más sencillo y de menor latencia de implementación, ya que Flink no necesita examinar los *timestamps* de los eventos ni preocuparse por el desorden.
* Desventajas**:** Es **impreciso**. El tiempo de procesamiento es volátil; las pausas de red, los cuellos de botella en el *Task Manager* o los reintentos de fallos causan que los eventos se registren con horas incorrectas.
* Uso Ideal**:** Para *jobs* que no requieren una alta precisión y donde la latencia es la máxima prioridad (ej., monitoreo de la salud del sistema).

**2. Tiempo de Evento (*Event Time*)**

* Definición**:** Utiliza la marca de tiempo incluida en el evento por la fuente original (ej., la hora en que el sensor capturó la lectura o la hora en que el usuario hizo el clic).
* Ventajas**:** Garantiza la corrección. Los cálculos son determinísticos; si ejecutas el *job* hoy o dentro de un mes, obtendrás el mismo resultado, ya que se basa en el *timestamp* del evento.
* Desventajas**:** Requiere el mecanismo de **Watermarks** para funcionar y añade una pequeña capa de latencia para gestionar el desorden.
* UsoIdeal**:** **Es el estándar para la Ingeniería de Datos** y para cualquier aplicación que requiera fiabilidad (finanzas, inventario, facturación).

### Gestión del Desorden: Watermarks (Marcas de Agua)

Este es el mecanismo que permite que el **Event Time** funcione. Las *Watermarks* son la solución al problema de los eventos fuera de orden.

**1. Definición y Función de la Watermark**

* Concepto**:** Una *Watermark* es un marcador de tiempo insertado en el *stream* que le dice a Flink: "El tiempo del evento ha avanzado hasta **T**. Asumo que no llegarán más eventos con un *timestamp* anterior a **T**."
* Función Clave**:** La *Watermark* es el disparador lógico que le indica a Flink cuándo debe cerrar una Ventana de Procesamiento y emitir un resultado final.

**2. La Regla del Mínimo (Sincronización Global)**

* Problema**:** En un *clúster* paralelo, las particiones del *stream* (Kafka) llegan a diferentes *Task Managers* con diferentes latencias. Si cada *Task Manager* usara su propio reloj, las ventanas se cerrarían en momentos inconsistentes.
* Mecanismo**:** Flink aplica la **Regla del Mínimo**. La **Watermark Global** que avanza por el *stream* es siempre la **Watermark más baja** de todas las subtareas paralelas.
* Consecuencia**:** La subtarea más lenta marca la velocidad de todo el *job*. El tiempo lógico del *stream* solo avanza cuando la partición más rezagada se pone al día, garantizando que ninguna ventana se cierre prematuramente.

### Ventanas de Procesamiento y Tolerancia a la Latencia

Las ventanas son la forma en que Flink agrupa los eventos para realizar cálculos, y su definición es clave para la lógica de negocio.

**1. Tipos de Ventanas (Window Assigners)**

| **Tipo de Ventana** | **Definición** | **Uso Ideal** |
| --- | --- | --- |
| **Fijas (*Tumbling*)** | Intervalos de tiempo fijos y **no superpuestos**. | Recuento periódico y discreto (ej. ventas totales cada 10 minutos). |
| **Deslizantes (*Sliding*)** | Intervalos de tiempo fijos que se **superponen** (tamaño > deslizamiento). | Cálculos de promedios móviles (ej. la media de los últimos 5 minutos, actualizada cada 1 minuto). |
| **Sesión (*Session*)** | Agrupación basada en la **inactividad** (*gap time*) entre eventos. | Análisis de comportamiento de usuario (la ventana se cierra si el usuario está inactivo por 30 minutos). |

**2. Eventos Tardíos (*Late Events*) y Tolerancia**

* Eventos Tardíos**:** Son eventos que llegan al *Task Manager* después de que la *Watermark* ha pasado y la ventana ya se ha cerrado.
* Tolerancia a la Latencia **(*Allowed Lateness*):** Flink te permite configurar una **ventana de gracia**. Si un evento llega tarde, pero dentro de este periodo de tolerancia, Flink puede **reabrir la ventana** (o mantener el estado) y actualizar el cálculo para incluir el evento.
* Disposición Final**:** Los eventos que llegan después del periodo de tolerancia son enviados a un **Side Output** (Salida Lateral) o son descartados.

### Timers y Lógica Avanzada

Los **Timers** son la herramienta de Flink para construir lógica compleja que no encaja en el modelo predefinido de ventanas.

**1. El Timer como Alarma Programable**

* Definición**:** Un **Timer** es un *callback* (una función) que el usuario registra en un operador con estado. Este *callback* se ejecuta cuando el tiempo lógico del *stream* (la *Watermark*) **supera** el valor del *Timer*.
* Relación Clave**:** El *Timer* se guarda como parte del **Estado** de la aplicación, lo que garantiza que, incluso si el *Task Manager* falla, la alarma se disparará después de la recuperación.

**2. Usos Críticos del Timer**

| **Caso de Uso** | **Problema Resuelto** | **Mecanismo** |
| --- | --- | --- |
| **Detección de Timeouts** | Reaccionar a la **ausencia** de un evento (ej. carrito abandonado). | Si un evento no llega, la *Watermark* supera el *Timer* registrado al inicio, y Flink emite una alerta. |
| **Limpieza de Estado** | Evitar que el **Estado** crezca indefinidamente y consuma memoria. | Se registra un *Timer* futuro para cada clave. Si el *Timer* se dispara, el estado de esa clave se **limpia** (clear()). |
| **Emisión Periódica** | Generar resultados intermedios de un cálculo complejo a intervalos fijos. | Se utiliza un *Timer* de *Processing Time* o *Event Time* para forzar una salida. |

Con esto, hemos cubierto toda la sección de tiempo y corrección. El siguiente paso es el **Módulo 2** (Gestión de Estado y Fallos) para ver cómo se protege este estado.

## Integración y Patrones de Transformación

### Conexión con Kafka (Serialización y Formatos)

Este capítulo se enfoca en cómo Flink consume y produce datos de la fuente de *streaming* más común en la industria.

**1. Configuración del Source y el Sink de Kafka**

* Source **(**Consumidor**):** Flink debe configurarse para leer desde un *topic* específico de Kafka. El Estado del Operador de Flink se encarga de guardar los **Offsets** de Kafka para cada partición, garantizando la recuperación de la lectura en caso de fallo.
* Sink **(**Productor**):** Flink escribe los resultados de procesamiento de vuelta a un *topic* de Kafka. Aquí es donde se activa el protocolo ***Two-Phase Commit* (2PC)** para garantizar el ***Exactly Once*** en la escritura.

**2. Serialización: El Puente entre Sistemas**

La serialización es crítica para el rendimiento, ya que los datos deben convertirse de su formato en el *stream* a objetos de Flink (y viceversa).

* **Recomendación de Flink:** Usar formatos binarios, bien tipados y compactos sobre JSON o CSV.
* **POJOs:** Flink prefiere que el dato se ajuste a un **POJO** (Java/Scala) para usar su serializador optimizado (TypeInformation), que minimiza la latencia del *shuffle*.
* **Formatos Binarios Estándar:** Para el intercambio con Kafka, los estándares de la industria son:
  + **Apache Avro:** Incluye el esquema de los datos junto al dato binario, lo que facilita la evolución del esquema sin romper los *pipelines*.
  + **Protocol Buffers (Protobuf):** Extremadamente compacto y rápido, ideal para casos donde la latencia es crítica.

### La API Unificada: DataStream vs. Table API/SQL

Este capítulo diferencia entre la lógica de programación paso a paso y la lógica declarativa, enseñando al ingeniero cuándo usar la herramienta adecuada.

**1. DataStream API (El Código Imperativo)**

* Naturaleza**:** **Imperativa** (dices *cómo* hacer cada paso).
* Uso Ideal**:** Para la lógica de negocio que requiere control de bajo nivel.
  + Integración de sistemas externos o lógica compleja de *if/else*.
  + Implementación de funciones con **Timers** y gestión granular del **Estado con Clave**.
  + Casos de uso avanzados de **ML/AI** que requieren librerías externas.

**2. Table API / Flink SQL (El Lenguaje Declarativo)**

* Naturaleza**:** **Declarativa** (dices *qué* quieres). Los *streams* se ven y se consultan como **tablas dinámicas**.
* Optimización**:** Flink utiliza **Apache Calcite** como optimizador para tomar tu consulta SQL y traducirla a la forma más eficiente de ejecutarla en el *clúster*.
* Uso Ideal**:** Para *joins*, filtros, y agregaciones sencillas (similares a SQL tradicional). Es el lenguaje de preferencia de los analistas de datos.

**3. Estrategia Senior: La Combinación (Blending)**

* El Patrón**:** Un *Senior Data Engineer* combina ambas APIs:
  1. **DataStream API:** Se usa para la **ingesta**, el *parsing* inicial y la lógica de **Timeouts** o limpieza de estado.
  2. **Conversión:** El *stream* resultante se convierte a una **Table** (Tabla).
  3. **SQL:** Se usa para las **transformaciones complejas** (*joins* de múltiples fuentes, *lookups* y filtrados pesados), aprovechando la optimización de Apache Calcite.

### Patrones de Join en Streaming (Casos de Uso)

El *join* de *streams* es una operación con estado costosa. Este capítulo enseña los dos patrones fundamentales para garantizar la consistencia temporal.

**1. Join de Ventana (*Window Join*)**

* Mecanismo**:** Un evento del *Stream A* solo se une con un evento del *Stream B* si **ambos caen dentro de la misma Ventana de Tiempo** (ej. una ventana fija de 5 segundos) y comparten la misma **clave**.
* Uso Ideal**:** Unir datos que se espera que ocurran casi simultáneamente (ej. unir un evento de Pago con el ID de Sesión que llegó 2 segundos después).
* Desafío**:** Requiere que Flink mantenga el estado de ambos *streams* dentro de los límites de la ventana.

**2. Join por Intervalo (*Interval Join*)**

* Mecanismo**:** Es más flexible. Une un evento del *Stream A* con cualquier evento del *Stream B* que haya ocurrido dentro de un intervalo de tiempo relativo a *A* (ej. entre 10 minutos antes y 5 minutos después del evento *A*).
* Uso Ideal**:** Análisis de atribución o correlación (ej. unir una Compra con todos los Clicks de Navegación que ocurrieron en los 15 minutos *anteriores* a esa compra).

### La Dualidad de Almacenamiento: Integración con Delta Lake

Este capítulo conecta el conocimiento de Flink con la arquitectura de mi empresa, enfocada en la "Dualidad de Almacenamiento".

**1. Delta Lake como Destino Consistente (Sink)**

* Propósito**:** Utilizar Delta Lake (en tu *Data Lake*) como el destino final para almacenar los datos procesados por Flink.
* Mecanismo**:** El Sink de Flink escribe los datos en Delta Lake, aprovechando las capacidades de **ACID** (Atomicidad, Consistencia, Aislamiento, Durabilidad) del formato de tabla.
* Garantía**:** El conector debe integrarse con el protocolo ***Two-Phase Commit*** de Flink para garantizar que el *commit* en la tabla Delta solo se realice si el *Checkpoint* es exitoso.

**2. Delta Lake como Fuente de Stream (Source)**

* Propósito**:** La capacidad de Delta Lake de exponer un **Change Data Feed (CDF)** permite a Flink leer los cambios o inserciones de la tabla como un *stream* continuo.
* Uso**:** Flink puede leer datos de *batch* almacenados en Delta Lake, aplicar una lógica de *streaming* (ventanas, *joins*) y luego escribir el resultado de vuelta en Delta Lake, cerrando el ciclo de procesamiento.

### Caso práctico de integración: kafka a postgresql

Este ejemplo de código Java ilustra el *pipeline* más común: leer de una fuente continua (Kafka), aplicar una transformación con estado simple (contar) y escribir el resultado en una base de datos transaccional (PostgreSQL) con la garantía de ***At Least Once*** (al menos una vez), ya que lograr *Exactly Once* en Postgres requiere conectores transaccionales que son más complejos de ejemplificar, o una tabla idempotente.

Usaremos la sintaxis de la **DataStream API** y una función de *Sink* genérica.

**1. El POJO de Entrada: ClickEvent**

* Concepto: Los objetos ClinckEvent (entrada) y ClickCount (salida) deben ser POJOs (Plain Old Java Objects). De esta manera Flink puede inspeccionar estos objetos (clase pública, constructor vacío, campos públicos) para utilizar su serializador optimizado (TypeInformation). Esta serialización eficiente reduce el tamaño de los datos en tránsito y minimiza la latencia de Shuffle cuando los datos se mueven por la red o se guardan en el estado.

Definimos el objeto que Flink serializará y procesará:

|  |
| --- |
| public class ClickEvent {  public String userId;  public long timestamp;  public String page;  public ClickEvent() {} // Constructor sin argumentos (necesario para POJO)  public ClickEvent(String userId, long timestamp, String page) {  this.userId = userId;  this.timestamp = timestamp;  this.page = page;  }  } |

**2. El Job Principal: Contar y Almacenar**

Este código lee de Kafka, cuenta los clics por usuario y luego escribe el resultado a ctualizado en PostgreSQL.

|  |
| --- |
| import org.apache.flink.api.common.functions.RichFlatMapFunction;  import org.apache.flink.api.common.serialization.SimpleStringSchema;  import org.apache.flink.api.common.state.ValueState;  import org.apache.flink.api.common.state.ValueStateDescriptor;  import org.apache.flink.configuration.Configuration;  import org.apache.flink.connector.jdbc.JdbcConnectionOptions;  import org.apache.flink.connector.jdbc.JdbcSink;  import org.apache.flink.connector.kafka.source.KafkaSource;  import org.apache.flink.connector.kafka.source.enumerator.topic.Topics;  import org.apache.flink.streaming.api.datastream.DataStream;  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  import org.apache.flink.util.Collector;  public class KafkaToPostgresJob {      private static final String KAFKA\_TOPIC = "click\_events";      private static final String BOOTSTRAP\_SERVERS = "localhost:9092";      public static void main(String[] args) throws Exception {          final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();          // 1. \*\*ACTIVAR CHECKPOINTING\*\* - Obligatorio para cualquier job con estado          env.enableCheckpointing(5000); // Checkpoint cada 5 segundos          // 2. CONFIGURAR KAFKA SOURCE (Lectura de la fuente)          KafkaSource<String> source = KafkaSource.<String>builder()              .setBootstrapServers(BOOTSTRAP\_SERVERS)              .setTopics(Topics.dominate(KAFKA\_TOPIC))              .setGroupId("flink-postgres-group")              .setValueOnlyDeserializer(new SimpleStringSchema())              .build();          DataStream<String> kafkaStream = env.fromSource(source,              WatermarkStrategy.noWatermarks(), // Usaremos NoWatermarks para este ejemplo simple              "Kafka Source");          // 3. DESERIALIZACIÓN Y TRANSFORMACIÓN CON ESTADO (Keyed State)          DataStream<ClickCount> countedClicks = kafkaStream              // Simulación de deserialización (En un caso real usarías un DeserializationSchema para el POJO)              .map(s -> new ClickEvent(s.split(",")[0], System.currentTimeMillis(), s.split(",")[1]))                // KeyBy: Particiona el stream por el usuario              .keyBy(event -> event.userId)                // FlatMap con estado para el conteo acumulado              .flatMap(new CountClicksFunction());          // 4. CONFIGURAR JDBC SINK (Escritura en PostgreSQL)          countedClicks.addSink(              JdbcSink.sink(                  // SQL: El UPDATE intenta actualizar el registro. Si falla (ON CONFLICT), lo INSERTa.                  // Esta lógica se acerca al At Least Once. Para Exactly Once se necesita un conector transaccional.                  "INSERT INTO user\_clicks (user\_id, total\_clicks) VALUES (?, ?) ON CONFLICT (user\_id) DO UPDATE SET total\_clicks = EXCLUDED.total\_clicks",                  (statement, count) -> {                      statement.setString(1, count.userId);                      statement.setLong(2, count.totalClicks);                  },                  JdbcConnectionOptions.builder()                      .withUrl("jdbc:postgresql://localhost:5432/flink\_db")                      .withDriverName("org.postgresql.Driver")                      .withUsername("postgres")                      .withPassword("mypassword")                      .build()              )          ).name("PostgreSQL Sink");          // 5. EJECUTAR EL JOB          env.execute("Kafka to PostgreSQL Click Counter");      }      // Clase auxiliar para la salida (Output)      public static class ClickCount {          public String userId;          public long totalClicks;          public ClickCount(String userId, long totalClicks) {              this.userId = userId;              this.totalClicks = totalClicks;          }      }      // Función que mantiene el estado (contador)      public static class CountClicksFunction extends RichFlatMapFunction<ClickEvent, ClickCount> {          private ValueState<Long> countState;          @Override          public void open(Configuration parameters) {              // Inicialización: se define cómo se almacena el estado (Valor por defecto: 0L)              countState = getRuntimeContext().getState(                  new ValueStateDescriptor<>("userClickCount", Long.class, 0L)              );          }          @Override          public void flatMap(ClickEvent event, Collector<ClickCount> out) throws Exception {              long currentCount = countState.value();              long newCount = currentCount + 1;              countState.update(newCount); // Actualiza el estado local              // Emite el resultado actualizado para que el Sink lo escriba              out.collect(new ClickCount(event.userId, newCount));          }      }  } |

**3. Keyed State: Memoria consistente**

* Concepto: La función CountClicksFunction utiliza un ValueState<Long> countState asociado a la clave (event.userId).

El keyBy(event -> event.userId) garantiza que todos los clicks del mismo usuario lleguen al mismo Task Slot. El estado de ese usuario se mantiene localmente en ese Task Slot.

* Propósito: Permite el cálculo acumulado y distribuido. Cada Task Manager gestiona su propio subconjunto de contadores sin comunicarse, lo que garantiza la velocidad y el paralelismo.

**Nota sobre la Garantía:** El SQL utilizado (ON CONFLICT DO UPDATE) garantiza un modo **At Least Once** efectivo (o **Idempotente**), ya que reprocesar un evento actualizará el conteo al mismo valor sin crear un registro duplicado. Para la garantía **Exactly Once**, necesitarías un conector de PostgreSQL que soporte el protocolo **Two-Phase Commit** (2PC) y usar una tabla de destino que lo permita.

## Testing, Operaciones y Optimización

*Pendiente de desarrollar. Incluir:*

* *Métricas y observavilidad: Es la habilidad de troubleshooting. Añade los conceptos de* ***Backpressure*** *(por qué ocurre un cuello de botella) y cómo leer la* ***Latencia de Watermark*** *para saber si el job se está quedando atrás.*
* *Optimización de estado: State TTL: El cleanup automático es una necesidad de producción. Detalla cómo configurar el* ***Time To Live (TTL)*** *para limpiar el estado antiguo (ej., usuarios inactivos) de forma automática y ahorrar memoria en RocksDB.*