# Detección de Anomalías con Apache Flink

Ricardo Andre Ulloa Vega
Jose Joaquin Tello León

Universidad Nacional de Ingeniería

5 de julio de 2025

# Índice

# Introducción al procesamiento de flujos con estado

Consideremos un flujo de origen que emite eventos con el esquema $e = [event_{id} : int, event_{value} : int]$

**Sin estado**

Objetivo: `event_value`.

Canalización: *source-map-sink*.

Función: `map`

**Con estado**

Función: `map` con estado.

Canalización: necesita el valor del evento anterior.

Requiere `keyBy` y estado por clave.

# Estado en Apache Flink

- Flink se descompone lógicamente en un grafo de operadores (DAG).

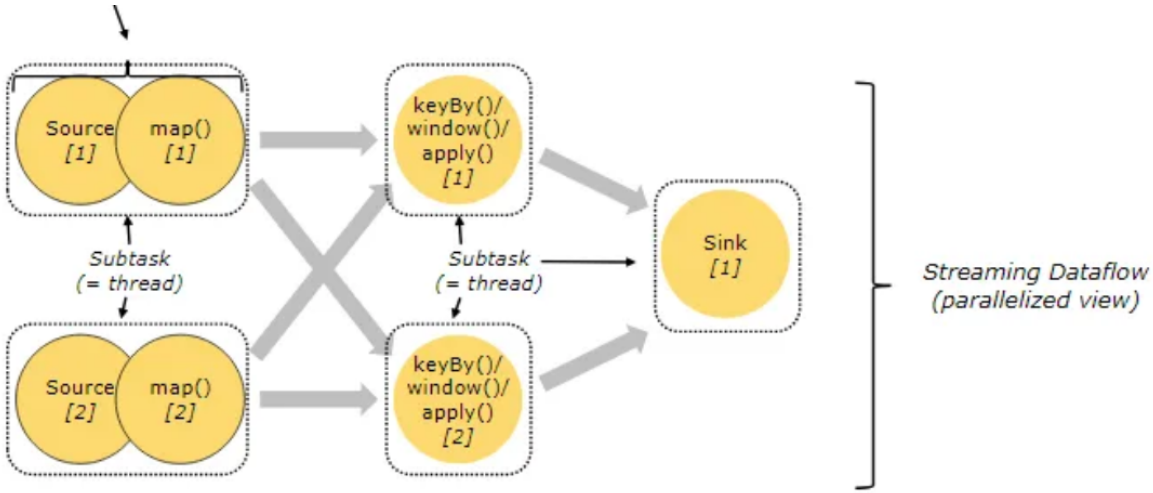



Figura: DAG de los pipelines de Flink

- Cada nodo es un operador (‘map‘, ‘keyBy‘, ‘window‘, etc.).
- Las aritas son los flujos de datos. Las aristas lógicas en el grafo de operadores del trabajo (verticalmente).
- Job Manager çerebro”de un cluster Flink.

# Estado en Apache Flink

- Recordemos que Flink sigue una arquitectura shared-nothing, por lo que no existe comunicación directa entre instancias paralelas durante la ejecución del job (job runtime).
- Para el procesamiento de flujos con estado de Flink, diferenciamos dos tipos de estado: estado del operador y estado con clave
- Reescalado con estado, considerando :
  - consistente
  - significativa
  - redistribucion

# Reasignacion de Estado con Reescalado

- Flink tiene un mecanismo que garantiza la integridad de los datos y el sistema **puntos de control** se busca consistentes del estado.

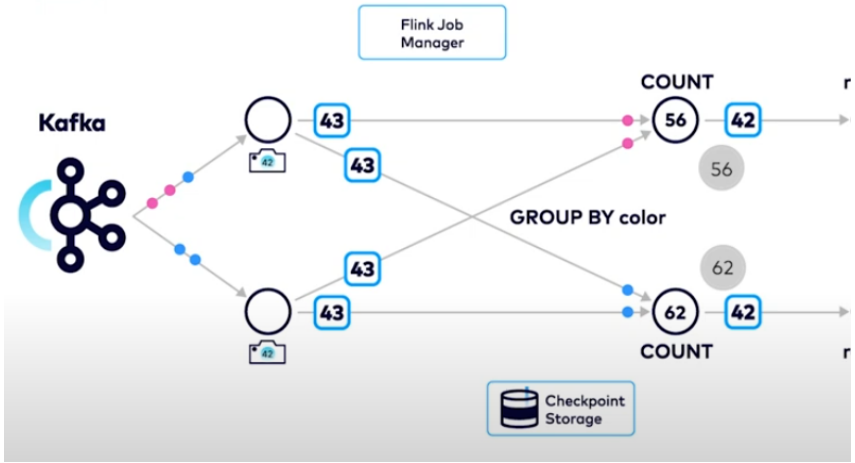



Figura: Inyeccion de checkpoints

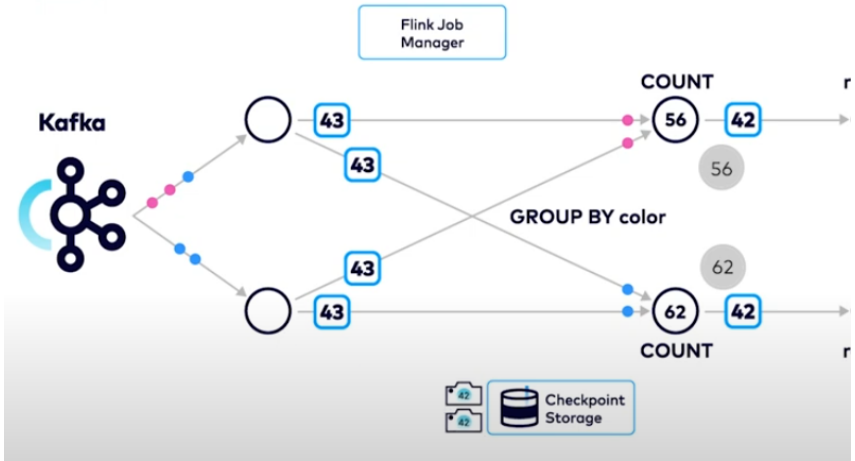
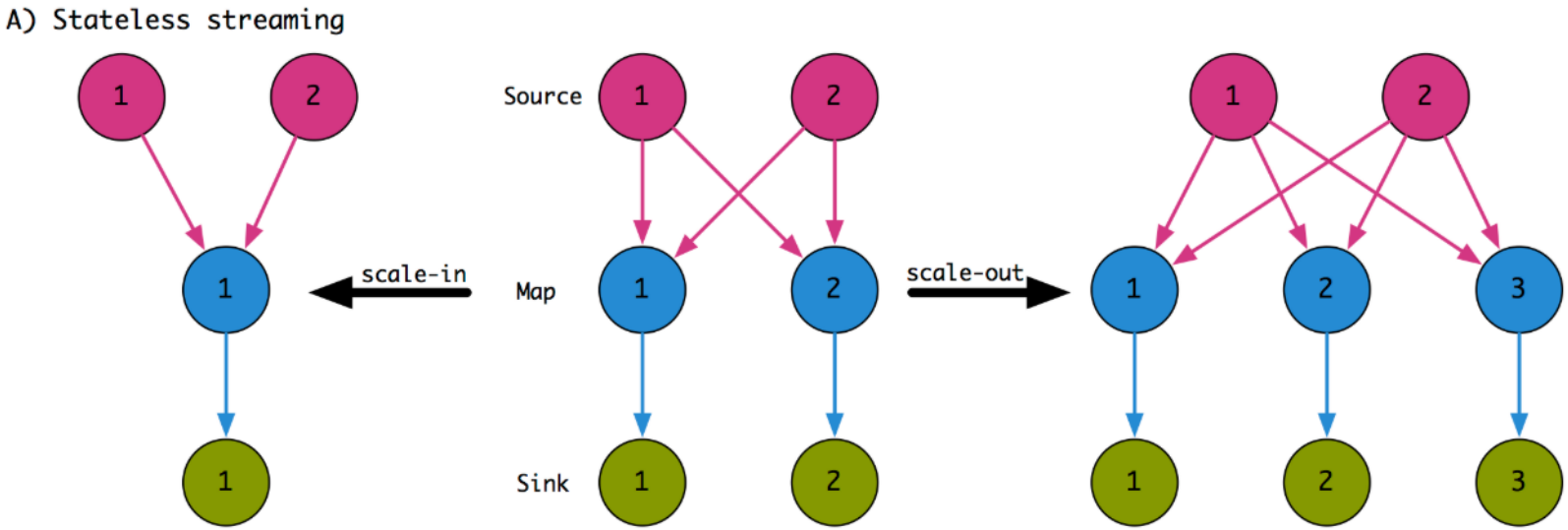



Figura: Guardamos estados consistentes en una BD

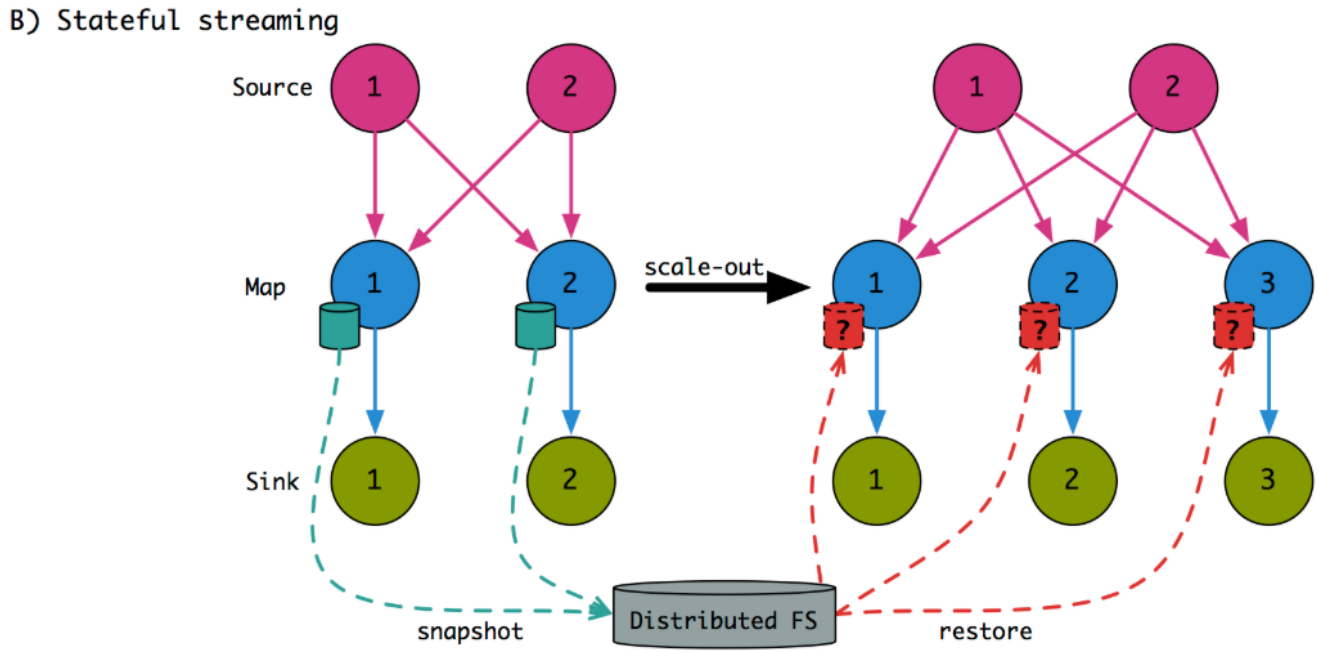
- JobManager inicia Checkpoint Coordinator, inyecta eventos especiales llamados **checkpoint barriers**.
- Guardando el estado en una base de datos distribuida como un HDFS,S3. Para reiniciar los estados ante posibles fallas.

# Reasignacion de Estado con Reescalado




- Solo requiere iniciar o detener instancias paralelas de operadores.

# Reasignacion de Estado con Reescalado



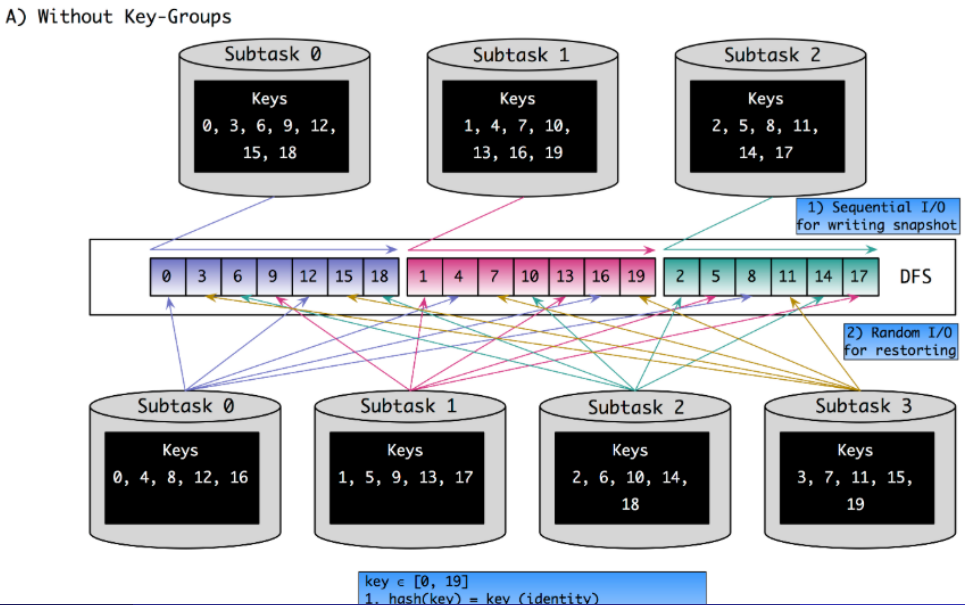

- Por los estados consistentes en la BD resolvemos la redistribucion, pero la consistencia se ve afectada en las instancias en paralelo *3ra instancia*.

# Reasignación de estado clave

- Para redistribuir estados clave son de forma determinista, se utiliza **hashing**:

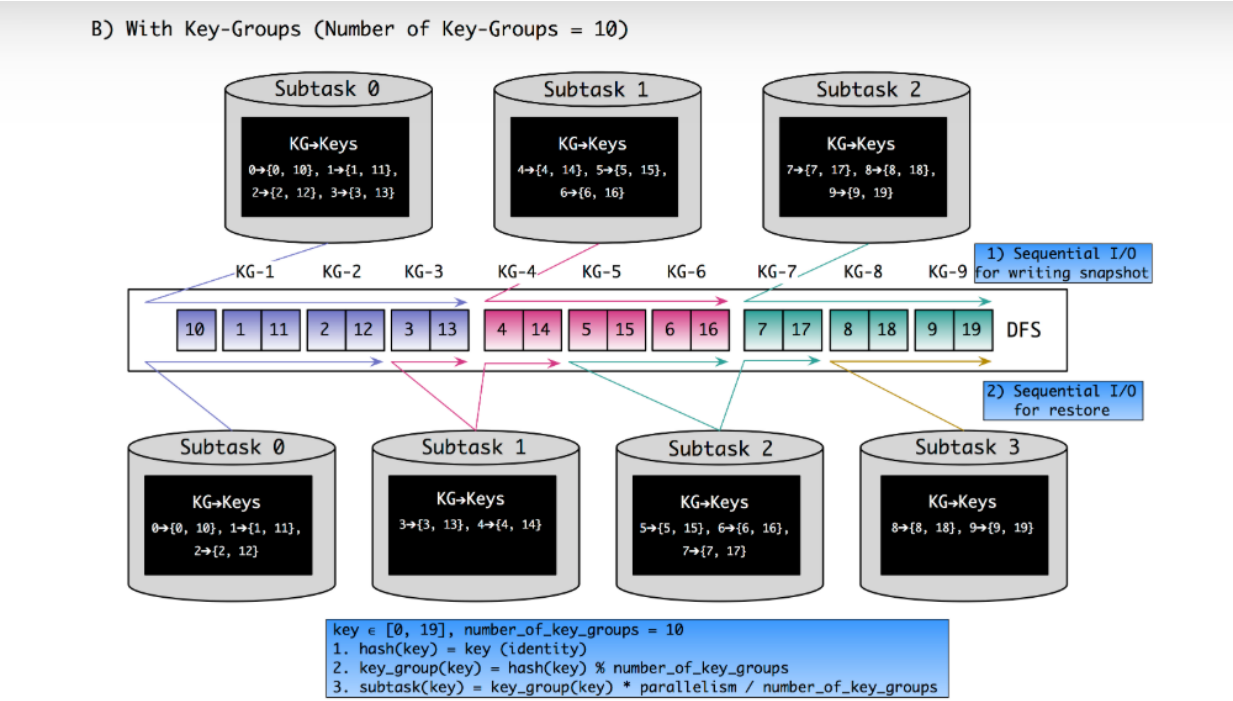
$$subtarea = hash(cliente_{id}) \quad \text{mód } paralelismo$$

- Sin embargo, al cambiar el grado de paralelismo, la asignación depende del número anterior de subtareas, lo que dificulta mantener el estado correctamente.
- Aunque el *hashing consistente* puede mitigar este problema, implica una sobrecarga considerable en red y disco al mover los estados.



# Reasignación de estado clave

- **Motivación:** Flink introduce los *key-groups* para optimizar la redistribución del estado durante el reescalado.



# Reasignación de estado clave

| Operador | Key-Groups (Par = 3) | Key-Groups (Par = 4) |
|---|---|---|
| 0 | 0–3 | 0–2 |
| 1 | 4–6 | 3–5 |
| 2 | 7–9 | 6–8 |
| 3 | – | 9 |

**Ventajas:**

- **Transferencia mínima:** Solo se mueven los key-groups afectados.
- **Lecturas secuenciales:** Flink accede a rangos contiguos, evitando I/O aleatorio.

# Reasignación de estado por operador

- En Kafka, el desplazamiento de las particiones (*offsets*) se maneja de forma independiente para cada partición.
- Idealmente, en Flink también se busca una reasignación de estado granular del tipo:

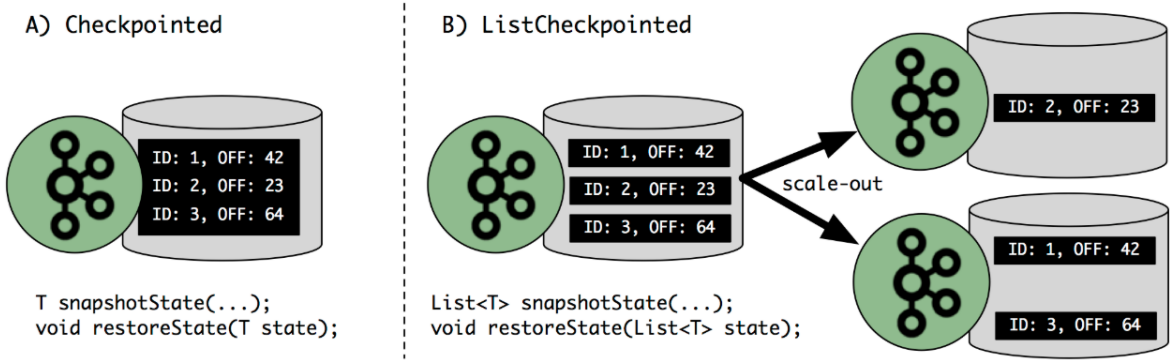
$$\langle \text{PartitionID}, \text{offset} \rangle$$

- El objetivo es contar con una unidad de estado que sea:
  - **Atómica:** indivisible durante el procesamiento.
  - **Redistribuible:** pueda moverse entre operadores sin depender del estado global.

# Reasignación de estado por operador

- La redistribución **round-robin** en `ListCheckpointed` es eficiente para el *Operator State*.
- Permite distribuir las unidades de estado (por ejemplo, offsets de particiones Kafka) de forma circular y equitativa entre subtareas.

*Figure 2*




- La redistribución **round-robin** en `ListCheckpointed` es eficiente para el *Operator State*.
- Permite distribuir las unidades de estado (por ejemplo, offsets de particiones Kafka) de forma circular y equitativa entre subtareas.

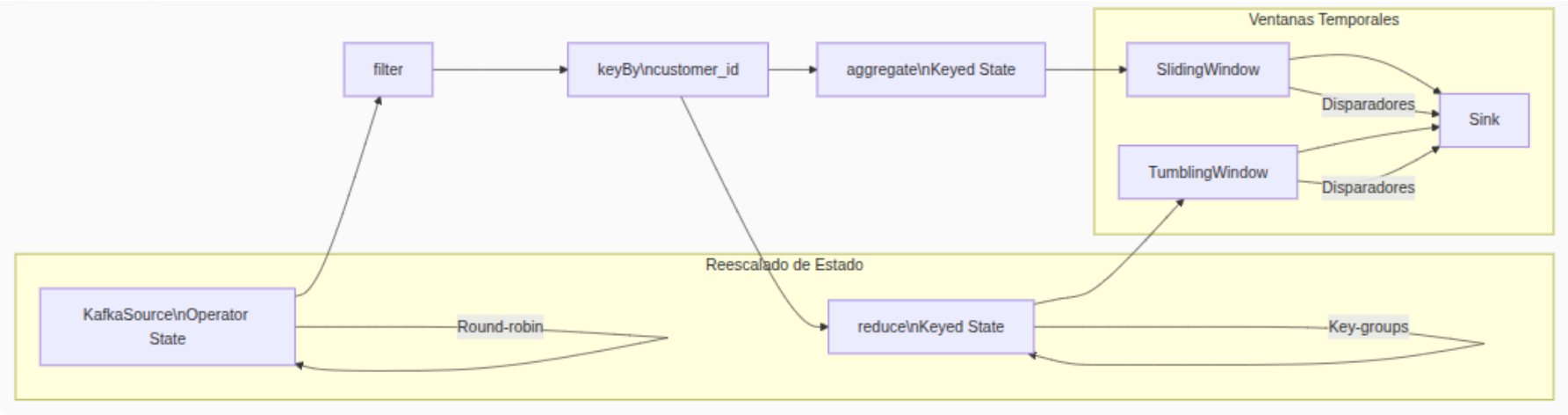
# Reasignación de estado por operador

- Minimiza Transferencia de Estado Cada elemento de estado (ej: una partición Kafka) es independiente.

# Comparativas entre estado por clave y operador

| **Aspecto** | **Operator State** | **Keyed State** |
|---|---|---|
| Redistribución | Round-robin simple | Usa `hashing key-groups` |
| Indexación | No requiere metadatos complejos | Key-groups indexados y asignados por rangos |
| Atomicidad | Unidades independientes como `<PartitionID, Offset>` | Estado agrupado por clave, no divisible |
| Operadores que lo usan | KafkaSource, FileSource, ventanas globales | keyBy, reduce, aggregate, ventanas claveadas |

# Diseño de *Pipelines* Escalables en Flink



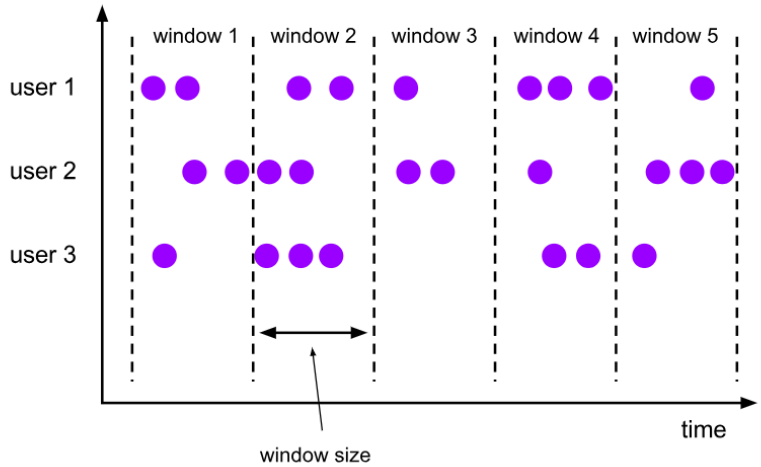
# Ciclo de Vida de Ventanas Temporales

- Una ventana se crea cuando llega el primer elemento que debe pertenecerle.
- El rango de una ventana tumbling está definido por su duración fija $t$ y se aplica en intervalos fijos alineados, como $[n \cdot t,\ (n+1) \cdot t)$ para $n \in \mathbb{N}$. Opcionalmente, puede extenderse con una tolerancia (*allowed lateness*).
- Las ventanas basadas en **Processing Time** se cierran cuando el reloj del sistema supera el final de la ventana.
- Una nueva ventana se crea cuando el **watermark** de un nuevo dato supera el rango.

# Ciclo de Vida de Ventanas Temporales

**Tumbling Windows**

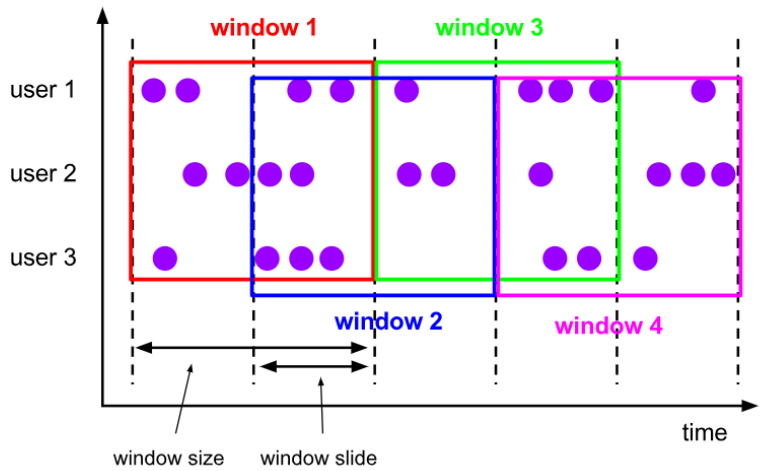
- Ventanas sin solapamiento.
- Cada evento pertenece a una única ventana.
- Útiles para agregaciones periódicas (por ejemplo, cada 5 segundos).



# Ciclo de Vida de Ventanas Temporales

**Sliding Windows**

- Ventanas con solapamiento.
- Cada evento puede estar en varias ventanas.
- Útiles para análisis continuo como promedios móviles.



# Ciclo de Vida de Ventanas Temporales

- Al finalizar una ventana, Flink ejecuta las funciones de agregación definidas sobre los datos agrupados:
  - **reduce**: Realiza una agregación incremental, como suma o conteo, optimizada en memoria.
  - **process**: Permite lógica personalizada, acceso completo a los datos de la ventana y contexto (como timestamp o temporizadores).
- Tras la ejecución, Flink descarta el estado asociado a la ventana, liberando recursos del *state backend* para mantener el sistema escalable.

# Ciclo de Vida de Ventanas Temporales

- Al finalizar una ventana, Flink ejecuta las funciones de agregación definidas sobre los datos agrupados:
  - **reduce**: Realiza una agregación incremental, como suma o conteo, optimizada en memoria.
  - **process**: Permite lógica personalizada, acceso completo a los datos de la ventana y contexto (como timestamp o temporizadores).
- Tras la ejecución, Flink descarta el estado asociado a la ventana, liberando recursos del *state backend* para mantener el sistema escalable.

# Arquitectura de la Implementación

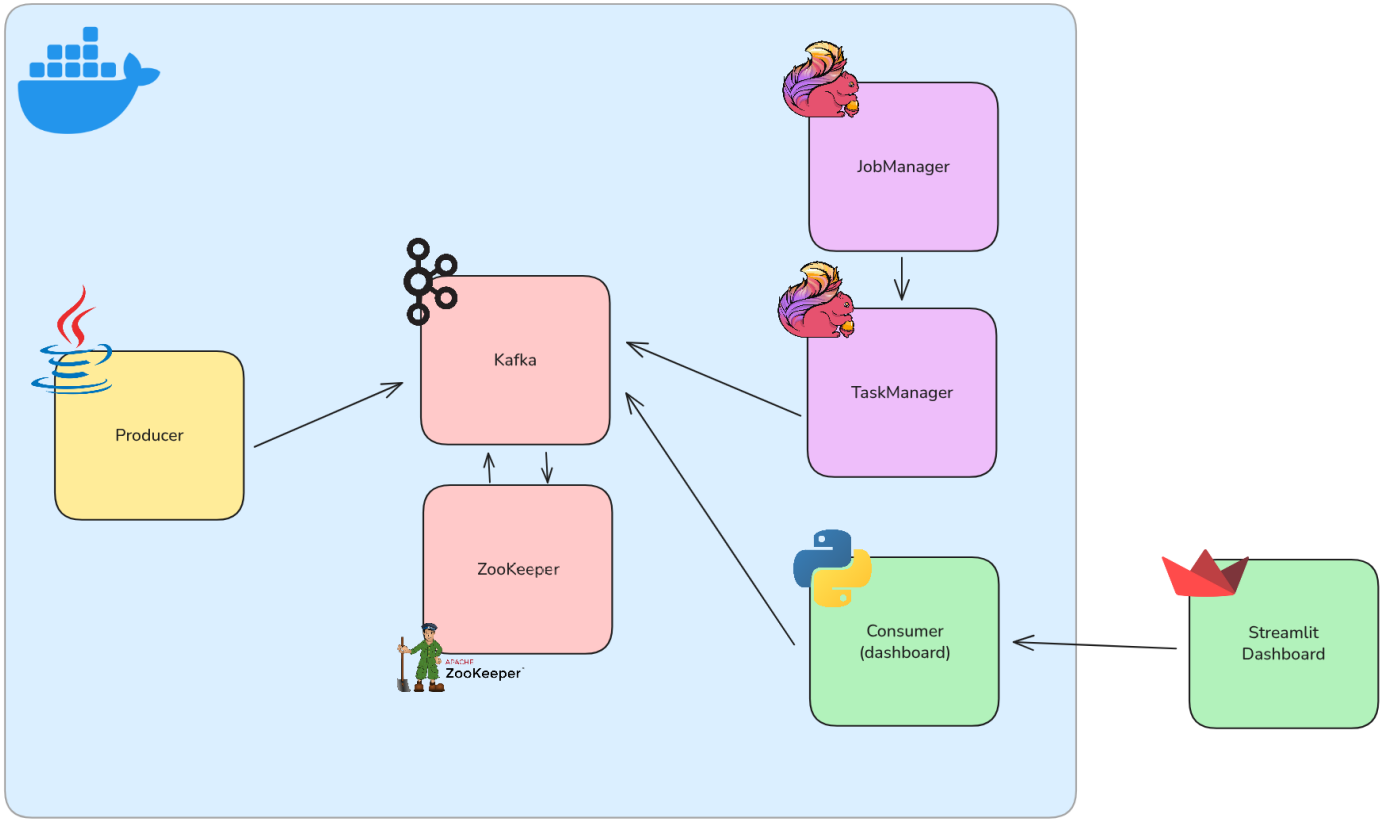



Figura: Arquitectura usada para la solución

# Productor de Transacciones (Java)

Este script Java es el encargado de generar y enviar eventos de transacciones al sistema Kafka. Actúa como el **punto de entrada de nuestros datos**, simulando la actividad de los usuarios.

- **1. Configuración Esencial**
  - **Definición de Conexión**: Se especifica el **tópico de Kafka** (‘transactions‘) al que se enviarán los datos y la **dirección del servidor Kafka** (‘kafka:9092‘).
  - **Propiedades del Productor**: Se configuran los **serializadores** necesarios para convertir los datos (claves y valores) de objetos Java a un formato de bytes que Kafka pueda transmitir eficientemente.
  - **Conversión a JSON**: Se utiliza una herramienta (`ObjectMapper`) para transformar objetos de transacción de Java a **cadenas JSON**. Esto es fundamental para enviar datos estructurados y legibles a Kafka.

# Productor de Transacciones (Java)

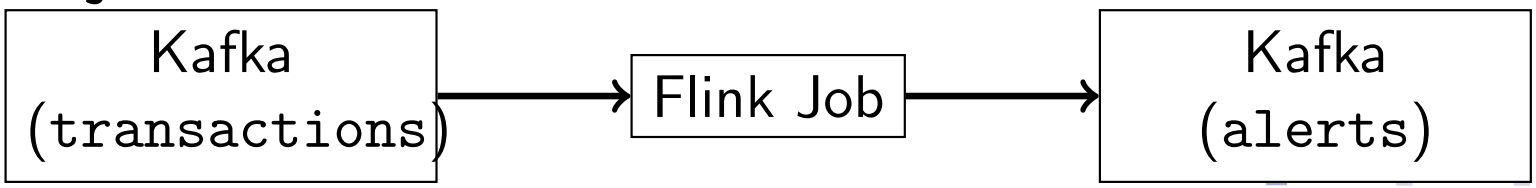
- **2. Generación y Envío de Eventos**
  - **Simulación de Datos**: El script genera transacciones con información variada como ID de usuario, monto, ubicación y dirección IP.
  - **Inyección de Anomalías**:
    - Se introduce aleatoriamente (**15 %** de probabilidad) transacciones con **ubicaciones inusuales** (lejos de la ubicación base del usuario).
    - **20 %** de probabilidad para **montos altos**.
    - **10 %** de probabilidad para **IPs sospechosas**.
    - *Este paso es crítico para probar la capacidad de detección de anomalías del sistema en fases posteriores.*
  - **Preparación del Mensaje**: Cada transacción generada se convierte a un objeto Java, luego se serializa a una cadena **JSON**.
  - **Envío a Kafka**: El mensaje JSON se empaqueta en un registro" de Kafka (incluyendo el tópico y el ID de usuario como clave) y se envía al broker.
  - **Control de Flujo**: Se introduce una pequeña pausa (‘500ms‘) entre envíos para simular un flujo de datos continuo y en tiempo real, evitando saturar el sistema de inmediato.

# Sistema de Detección de Anomalías con Apache Flink

Este componente central del sistema utiliza Apache Flink para procesar flujos de datos en tiempo real, identificar patrones inusuales en las transacciones y generar alertas.

- **1. Configuración y Fuentes de Datos**
  - **Entorno de Ejecución**: Se inicializa el entorno de Flink, que es el "cerebro" que gestiona el procesamiento del flujo.
  - **Origen de Datos (Source)**:
    - Flink se conecta a un tópico de Kafka llamado `"transactions"` para recibir los eventos crudos.
    - Configura cómo se leen los mensajes, asegurando que se procesen desde el inicio (`earliest`).
  - **Destino de Alertas (Sink)**:
    - Se prepara un "sumidero" de Kafka para enviar las alertas de anomalías detectadas.
    - Las alertas se enviarán a un tópico específico llamado `‘alerts‘`.

**Flujo Inicial de Datos:**



- **2. Procesamiento de Eventos y Detección de Anomalías**
  - **Agrupación por Usuario**: Los eventos de transacción se **agrupan por** `userId`. Esto es crucial para analizar el comportamiento de cada usuario de forma independiente.
  - **Ventanas Deslizantes**:
    - Se aplica una **ventana de tiempo deslizante**. En este caso, una ventana de **5 segundos** que "se desliza" **1 segundo** a la vez.
    - Flink procesa los eventos que caen dentro de cada ventana.

# Sistema de Detección de Anomalías con Apache Flink

- **Lógica de Detección de Anomalías (‘AnomalyDetectionFunction‘)**:
  - Se ejecuta sobre cada ventana (por usuario).
  - **Alta Frecuencia**: Detecta si un usuario realiza **más de 5 transacciones** en la ventana de 5 segundos.
  - **Transacciones Individuales**: Revisa cada transacción en la ventana para identificar:
    - **Montos altos**.
    - **Ubicaciones sospechosas**.
    - **Direcciones IP inusuales**.
  - Si se encuentran anomalías, se crea y emite un objeto ‘AnomalyAlert‘.
- **Envío de Alertas**: Las alertas generadas se envían al tópico de Kafka ‘`alerts`‘ a través del ‘Kafka Sink‘ configurado.