UD 7 - Apache Kafka Stream

1. Introducción

Kafka Streams es una biblioteca cliente para construir aplicaciones y microservicios, donde los datos de entrada y salida se almacenan en clústers de Kafka. Combina la simplicidad de escribir y desplegar aplicaciones Java y Scala estándar en el lado del cliente con los beneficios de la tecnología de clústeres del lado del servidor de Kafka.

Las aplicaciones desarrolladas con Kafka Streams podrán realizar procesamiento en streaming. De esta forma, se procesa de manera secuencial sobre flujos de datos sin límites temporales. Se basa en la mensajería de Kafka para permitir procesar datos en tiempo real. Pero mientras un productor Kafka sólo publica datos en un topic, y un consumidor únicamente consume datos de topics, las aplicaciones Kafka Streams pueden utilizar uno o varios topics como entrada, realizar algún tipo de transformación o procesado de esos datos y dejar el resultado como salida en otro u otros topics.

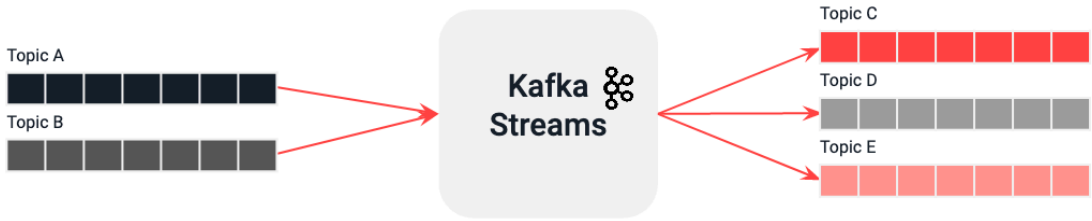


Figura 7.1_Kafka Stream: Topics y Kafka Streams (Fuente: paradigmadigital.com)

Además, Kafka Streams gestiona de forma transparente el equilibrio de carga de varias instancias de la misma aplicación aprovechando el modelo de paralelismo de Kafka.

2. Concepto

El concepto de kafka stream se basa en una topología, denominada **Stream Processing Topology**. Esta topología de procesamiento de streams es el esquema o diseño de cómo se procesan los streams en una aplicación Kafka Streams. Se compone de **stream processor** o nodos de procesador que son entidades lógicas que procesan los datos. Cada nodo en la topología está vinculado a otros nodos para formar un grafo dirigido acíclico (DAG).

2.1 Stream Processing Topology

Una topología de procesamiento de streams es el esquema o diseño de cómo se procesan los streams en una aplicación Kafka Streams. Se compone de nodos de procesador que son entidades lógicas que procesan los datos. Cada nodo en la topología está vinculado a otros nodos para formar un grafo dirigido acíclico (DAG).

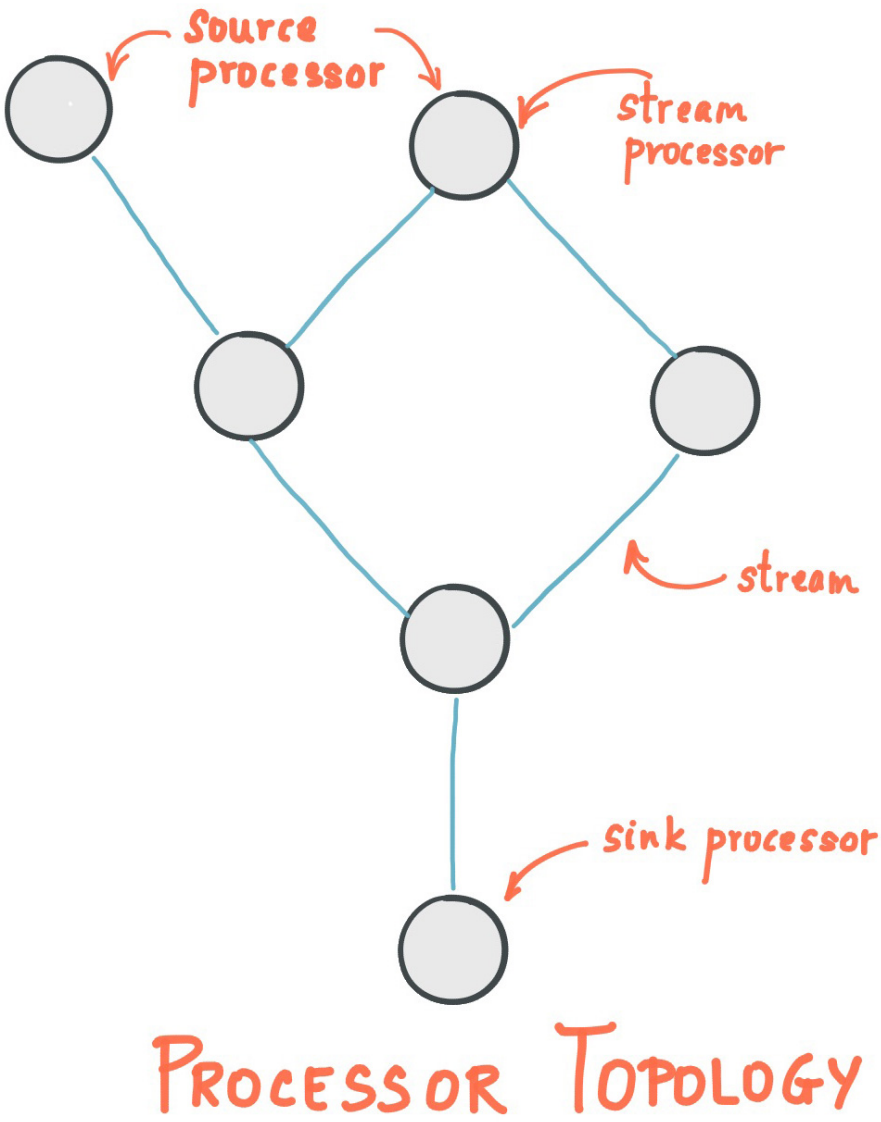


Figura 7.2_Kafka Streams: Stream Processing Topology (Fuente: kafka.apache.org)

Estos nodos pueden ser de 2 tipos

- **Source Processor**: Es un tipo especial de source processor que no tienen otros processor anteriores. Por tanto, producen un stream de entrada a su topología a partir de uno o

varios topics Kafka consumiendo registros de estos topics y reenviándolos a sus procesadores de stream descendente

- **Sink Processor**: Es un tipo especial de stream processor que no tiene procesadores de stream descendente. Envía todos los registros recibidos de sus strem process ascendente a un topic Kafka especificado.

2.2 Construcción de la Topología

La topología de procesamiento se define programáticamente usando la **Streams DSL** o la **Processor API**:

- Streams DSL: Proporciona abstracciones de alto nivel, como KStream, KTable, y GlobalKTable, que permiten definir transformaciones complejas con poco código, como `map`, `filter`, `join` and `aggregations`
- **Processor API**: Ofrece control granular sobre el procesamiento de streams y permite definir operaciones personalizadas, gestionar el estado y conectarse con otros sistemas.

2.3 KStream vs KTable

En Kafka Streams, KStream y KTable son dos abstracciones fundamentales que representan diferentes formas de ver los datos en un stream de Apache Kafka. Cada uno tiene características y usos específicos dependiendo de la naturaleza de los datos y el tipo de operaciones de procesamiento de streams que necesitas realizar.

KStream

Representa un stream de registros de datos donde cada registro en el stream es considerado un evento independiente. Se utilizan para modelar datos, donde cada registro es una pieza autocontenida de los datos dentro de un conjunto de datos sin consolidar. Esto debe entenderse como un flujo de registros, donde los nuevos registros no reemplazarán una parte de los datos existentes con un nuevo valor. Los streams contienen una historia o una secuencia de los datos.

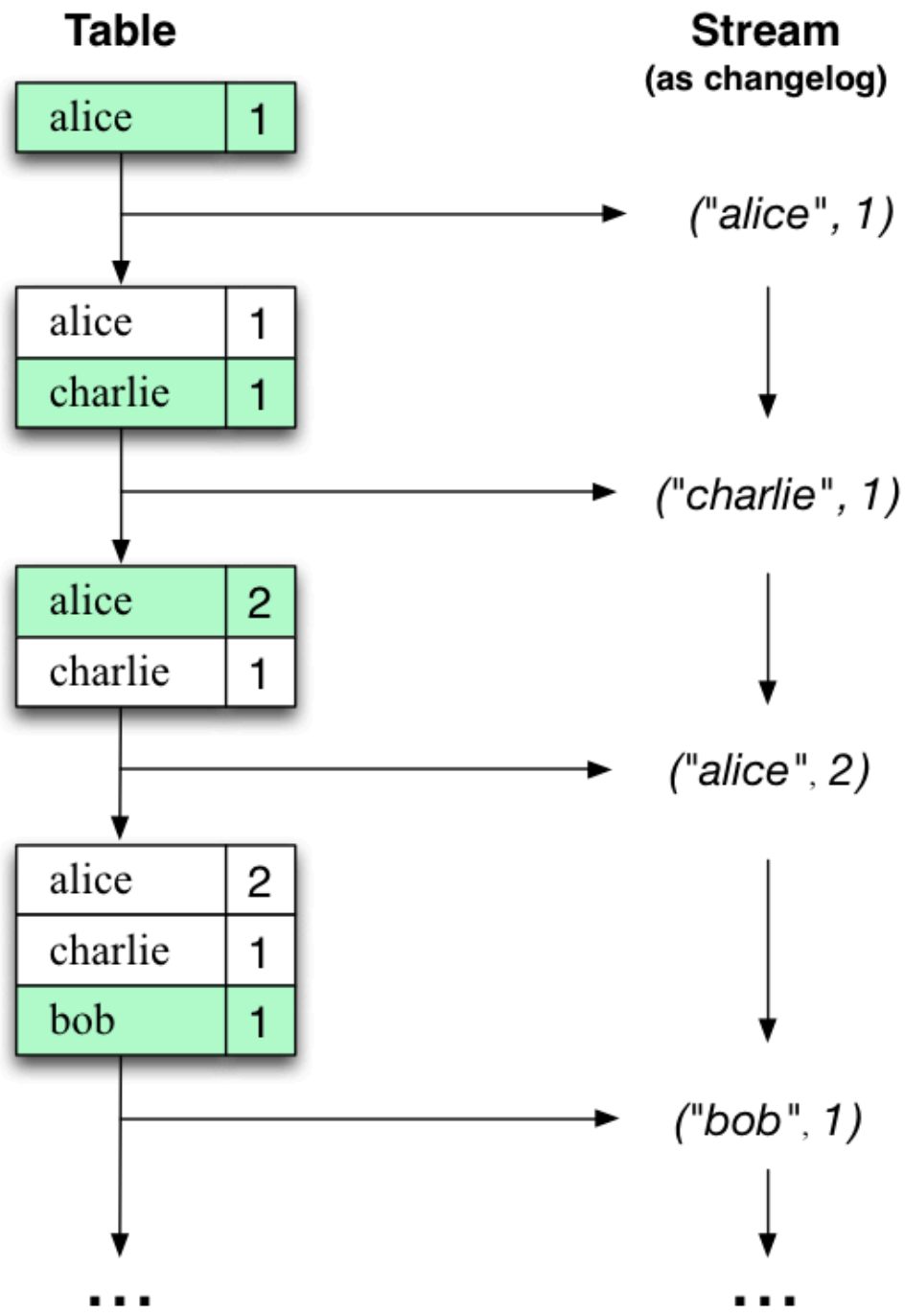


Figura 7.3_Kafka Streams: KTable vs KStream (Fuente: kafka.apache.org)

KTable

Representa una tabla de registros, donde cada registro es considerado un **estado actual** (última actualización) de la clave. Es similar a una base de datos relacional donde cada clave tiene un valor actual que puede ser actualizado o borrado.

♔: e2..f1
♖: f6..f7
♖: f4..h4
♚: g8..h7
♘: c5..e6
♖: f7..e7
♘: e6..f4
♟: a7..a6
♖: h4..h5
♖: e7..f7
♘: f4..h3
♝: c6..e8
♖: h5..h4
♖: b8..d8
♘: h3..g5
♟: h6..g5 {♘}

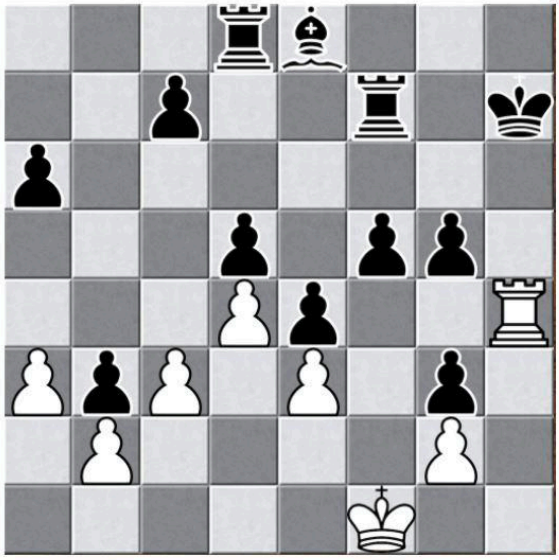


Figura 7.4_Kafka Streams: KTable vs KStream (Fuente: paradigmadigital.com)

2.3. Manejo del tiempo

El manejo del tiempo es crucial en Kafka Streams, y se distinguen tres tipos principales:

- **Event time:** Momento en el que se produjo un evento o registro de datos, es decir, cuando se creó originalmente "en la fuente". Ejemplo: Si el evento es un cambio de geo-localización reportado por un sensor GPS en un coche, entonces el evento-tiempo asociado sería el momento en que el sensor GPS capturó el cambio de localización
- **Processing time**: El momento en que el evento o registro de datos es procesado por la aplicación, es decir, cuando el registro está siendo consumido. El tiempo de procesamiento puede ser milisegundos, horas o días, etc. más tarde que el tiempo del evento original. Ejemplo: Imaginemos una aplicación analítica que lee y procesa los datos de geolocalización notificados por los sensores de los coches para presentarlos en un cuadro de mandos de gestión de flotas. En este caso, el tiempo de procesamiento en la aplicación analítica podría ser de milisegundos o segundos (por ejemplo, para canalizaciones en tiempo real basadas en Apache Kafka y Kafka Streams) o de horas (por ejemplo, para canalizaciones por lotes basadas en Apache Hadoop o Apache Spark) después del momento del evento.
- **Ingestion time**: El punto en el tiempo cuando un evento o registro de datos es almacenado en una partición de topic por un broker Kafka. La diferencia con el tiempo de evento es que esta marca de tiempo de ingestión se genera cuando el broker Kafka añade el registro al topic de destino, no cuando el registro se crea "en el origen". La diferencia con el tiempo de procesamiento es que el tiempo de procesamiento es cuando la aplicación de procesamiento de stream procesa el registro. Por ejemplo, si un registro nunca se procesa, no hay noción de tiempo de procesamiento para él, pero sigue teniendo un tiempo de

ingestión. Podríamos resumirlo como el momento en que el evento es añadido al log de Kafka.

3. Ejemplo 5. Demo KStreams

Como hemos comentado, Kafka Streams combina la simplicidad de escribir y desplegar aplicaciones Java y Scala estándar en el lado del cliente con los beneficios de la tecnología de clúster del lado del servidor de Kafka para hacer estas aplicaciones altamente escalables, elásticas, tolerantes a fallos, distribuidas y mucho más.

Realizaremos una demo con la aplicación `WordCount` (convertido para utilizar expresiones Java 8 lambda para facilitar la lectura)

```
// Serializers/deserializers (serde) for String and Long types
final Serde<String> stringSerde = Serdes.String();
final Serde<Long> longSerde = Serdes.Long();

// Construct a `KStream` from the input topic "streams-plaintext-input", where message values
// represent lines of text (for the sake of this example, we ignore whatever may be stored
// in the message keys).
KStream<String, String> textLines = builder.stream(
      "streams-plaintext-input",
      Consumed.with(stringSerde, stringSerde)
    );

KTable<String, Long> wordCounts = textLines
    // Split each text line, by whitespace, into words.
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

    // Group the text words as message keys
    .groupBy((key, value) -> value)

    // Count the occurrences of each word (message key).
    .count();

// Store the running counts as a changelog stream to the output topic.
wordCounts.toStream().to("streams-wordcount-output", Produced.with(Serdes.String(), Serdes.Long()));
```

1. Aprovecharemos nuestra configuración de un cluster Kafka realizada en el ejemplo 2, pero en esta ocasión cambiaremos los puertos de escucha del controller(9094) y del broker1(9092) y broker2(9093). *Puedes usar el ejemplo de un sólo nodo si lo crees conveniente*
2. Creamos los directorios necesarios para nuestro `ejemplo5`

```
mkdir -p /opt/kafka/ejemplo5/config
```

```
mkdir -p /opt/kafka/ejemplo5/logs
```

3. Hacemos 2 y 1 copia de los ficheros correspondientes de configuración para cada uno

```
cp config/kraft/controller.properties
/opt/kafka/ejemplo5/config/controller1.properties
cp config/kraft/broker.properties
/opt/kafka/ejemplo5/config/broker1.properties
cp config/kraft/broker.properties
/opt/kafka/ejemplo5/config/broker2.properties
```

4. Asignamos la configuración al controller

controller1.properties

```
# Server Basics
process.roles=controller
node.id=1
controller.quorum.voters=1@localhost:9094
# Socket Server Settings
listeners=CONTROLLER://localhost:9094
controller.listener.names=CONTROLLER
# Log Basics
log.dirs=/opt/kafka/ejemplo5/logs/controller1
```

5. Asignamos la siguiente configuración para cada broker

broker1

broker1.properties

```
# Server Basics
process.roles=broker
node.id=2
controller.quorum.voters=1@localhost:9094
# Socket Server Settings
listeners=PLAINTEXT://localhost:9092
advertised.listeners=PLAINTEXT://localhost:9092
# Log Basics
log.dirs=/opt/kafka/ejemplo5/logs/broker1
```

broker2

broker2.properties

```
# Server Basics
process.roles=broker
node.id=3
controller.quorum.voters=1@localhost:9094
# Socket Server Settings
listeners=PLAINTEXT://localhost:9093
advertised.listeners=PLAINTEXT://localhost:9093
```

```
# Log Basics
log.dirs=/opt/kafka/ejemplo5/logs/broker2
```

6. Iniciamos Kafka con KRaft

```
Genera un cluster UUID
KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
echo $KAFKA_CLUSTER_ID

#Formateamos los directorios de log
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
/opt/kafka/ejemplo5/config/controller1.properties
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
/opt/kafka/ejemplo5/config/broker1.properties
bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c
/opt/kafka/ejemplo5/config/broker2.properties
```

7. Iniciamos los server(1 controller y 2 brokers) cada uno en una terminal

```
#Ejecuta el servidor Kafka
bin/kafka-server-start.sh
/opt/kafka/ejemplo5/config/controller1.properties
bin/kafka-server-start.sh /opt/kafka/ejemplo5/config/broker1.properties
bin/kafka-server-start.sh /opt/kafka/ejemplo5/config/broker2.properties
```

8. Topic de entrada `streams-plaintext-input`

```
bin/kafka-topics.sh --create \
    --bootstrap-server localhost:9092 \
    --replication-factor 2 \
    --partitions 2 \
    --topic streams-plaintext-input
```

9. Topic de salida `streams-wordcount-output`

```
bin/kafka-topics.sh --create \
    --bootstrap-server localhost:9092 \
    --replication-factor 2 \
    --partitions 2 \
    --topic streams-wordcount-output \
    --config cleanup.policy=compact
```

10. Vemos la descripción de los topics:

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe

Topic: streams-wordcount-output TopicId: QWTyxqwaSQ2KRhB0UF5OgA
PartitionCount: 2   ReplicationFactor: 2    Configs:
cleanup.policy=compact,segment.bytes=1073741824
    Topic: streams-wordcount-output Partition: 0    Leader: 2   Replicas:
2,3   Isr: 2,3    Elr:    LastKnownElr:
```

```
5       Topic: streams-wordcount-output Partition: 1    Leader: 3   Replicas:
3,2   Isr: 3,2    Elr:    LastKnownElr:
6   Topic: streams-plaintext-input  TopicId: KKRkJgoUSha2rqEeAVHzgA
PartitionCount: 2   ReplicationFactor: 2    Configs: segment.bytes=1073741824
7       Topic: streams-plaintext-input  Partition: 0    Leader: 3   Replicas:
3,2   Isr: 3,2    Elr:    LastKnownElr:
8       Topic: streams-plaintext-input  Partition: 1    Leader: 2   Replicas:
2,3   Isr: 2,3    Elr:    LastKnownElr:
```

11. Iniciamos la aplicación

```
1   bin/kafka-run-class.sh
org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

La aplicación leerá los flujos del topic `streams-plaintext-input`, realizará los cálculos del algoritmo WordCount en cada uno de los mensajes leídos, y escribirá continuamente sus resultados actuales en el topic `streams-wordcount-output`. Por lo tanto, no habrá ninguna salida STDOUT excepto las entradas de registro, ya que los resultados se escriben de nuevo en Kafka.

12. Usamos Consumer API para ver como se consume el stream cuando empecemos a producir datos.

```
1   bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
2       --topic streams-wordcount-output \
3       --from-beginning \
4       --formatter org.apache.kafka.tools.consumer.DefaultMessageFormatter \
5       --property print.key=true \
6       --property print.value=true \
7       --property
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
8       --property
value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

13. Usamos Producer API para generar las palabras a contar por `WordCount`

```
1   bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic
streams-plaintext-input
```

Y producimos datos en consola

```
1   >all streams lead to kafka
```

14. Vemos la salida en consumer API

```
all     1
streams 1
lead    1
to      1
kafka   1
```

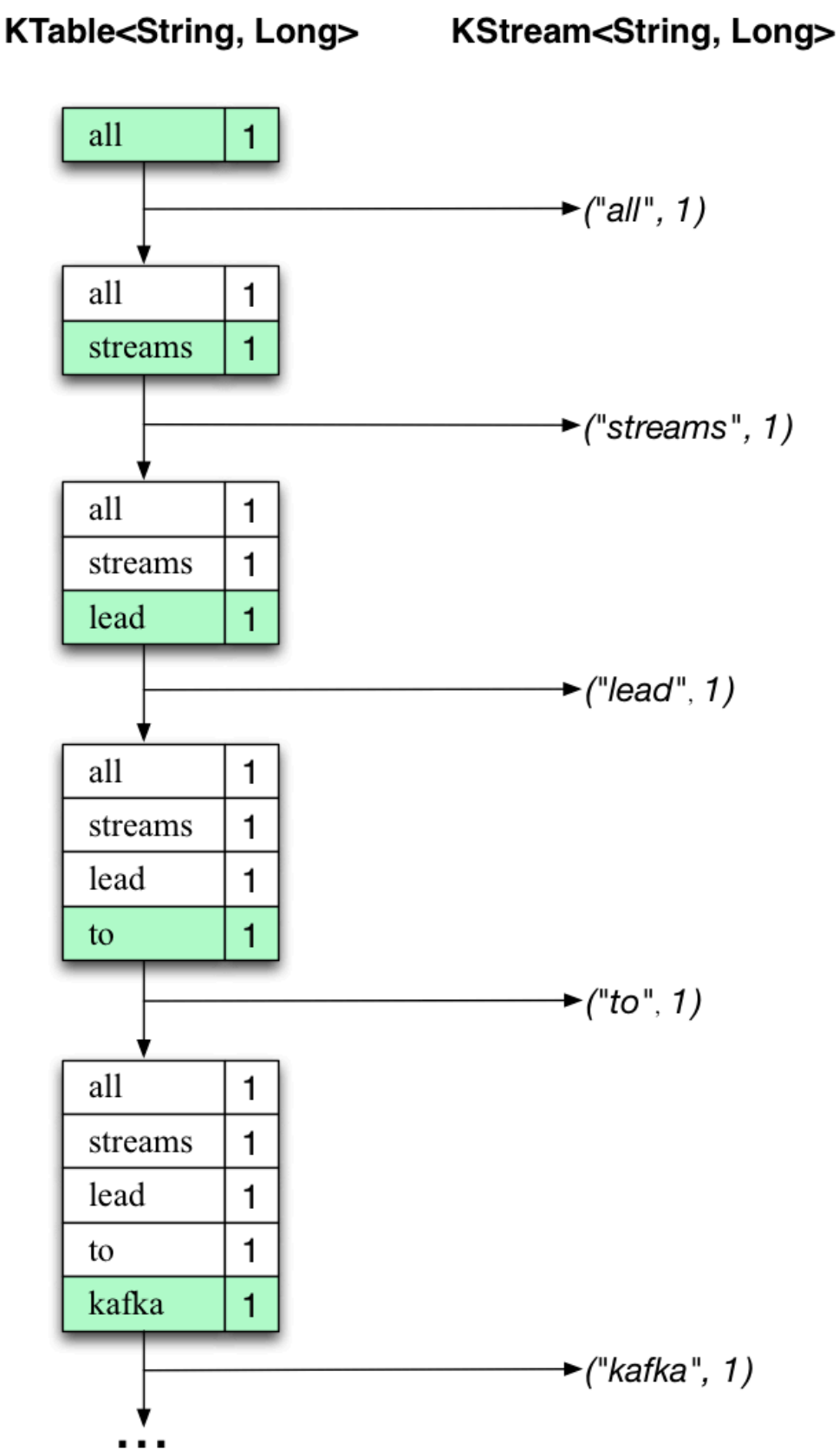


Figura 7.5_Kafka Streams: Ejemplo5. Procesamiento y consumo de datos 1 (Fuente: kafka.apache.org)

15. Seguimos produciendo datos en Producer API.

```
>all streams lead to kafka
>hello kafka streams
```

16. Observamos los datos de nuevo los datos ya procesados y consumidos

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
```

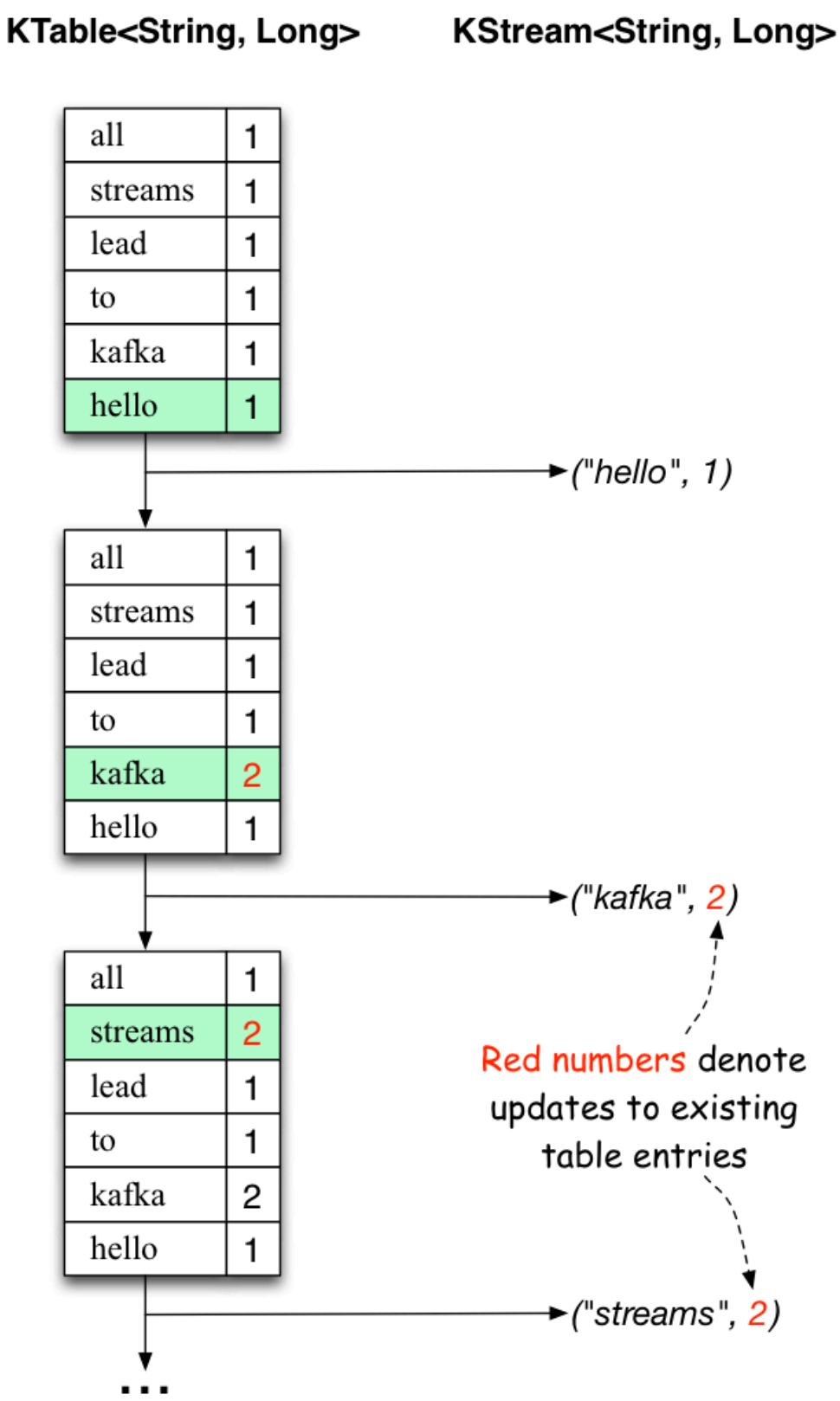


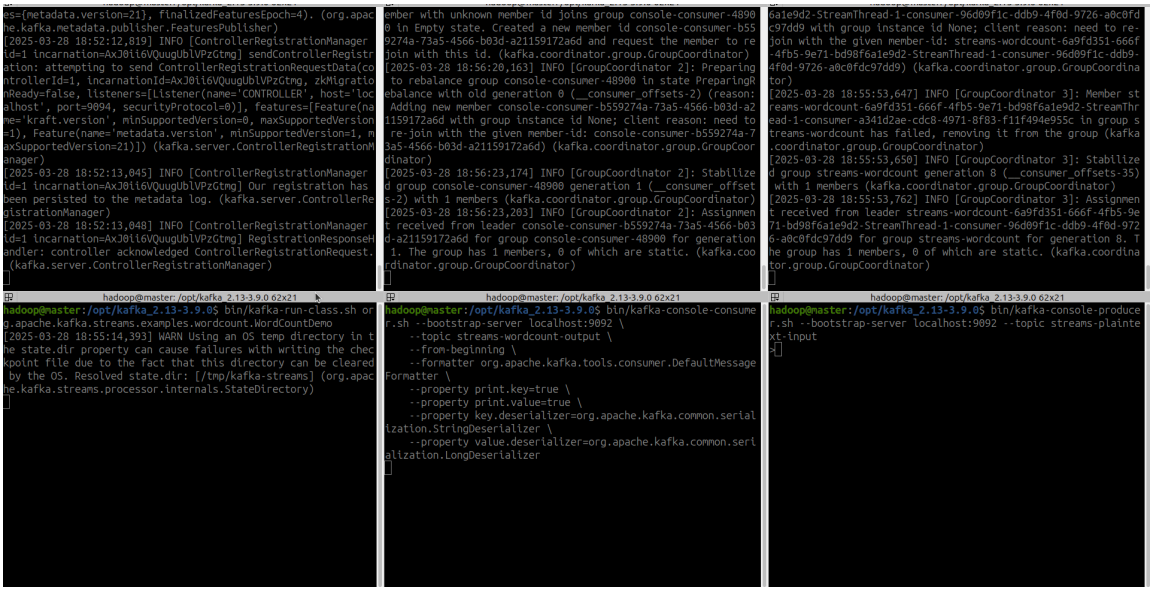
Figura 7.6_Kafka Streams: Ejemplo5. Procesamiento y consumo de datos 2 (Fuente: kafka.apache.org)

17. Más datos a producir

```
>all streams lead to kafka
>hello kafka streams
>join kafka summit
```

18. Datos procesados y consumidos

```
all     1
streams 1
lead    1
to      1
kafka   1
hello   1
kafka   2
streams 2
join    1
kafka   3
summit  1
```



Animación 7.1_Kafka Stream: Ejemplo 5