- **Raft:**

El nuevo timonel de Kafka Raft es un algoritmo de consenso distribuido diseñado para gestionar de forma eficiente y robusta la replicación de datos en sistemas distribuidos. En el contexto de Kafka, Raft ha reemplazado al antiguo algoritmo ZooKeeper para coordinar los *brokers* y garantizar la consistencia del estado del clúster.

¿Cuál es la función principal de Raft en Kafka?

- Elección de líder: Raft selecciona de forma democrática un líder entre los *brokers* del clúster. Este líder es el responsable de coordinar la replicación de los datos y de tomar decisiones sobre el estado del clúster.
- Replicación de *logs:* el líder replica los *logs* (o registros) de las particiones a los seguidores. Esto garantiza que todos los *brokers* tengan una copia actualizada de los datos.
- Tolerancia a fallos: Raft puede tolerar fallos en varios *brokers* sin comprometer la disponibilidad del sistema. Si el líder falla, se elige un nuevo líder rápidamente.

Características clave de Raft

- Simplicidad: Raft es más sencillo de entender e implementar que otros algoritmos de consenso, como Paxos.
- Eficiencia: Raft es altamente eficiente en términos de comunicación y procesamiento.
- Seguridad: Raft garantiza la seguridad de los datos mediante la replicación y la elección de un líder legítimo.
- Tolerancia a fallos: Raft puede tolerar fallos en varios nodos sin afectar a la disponibilidad del sistema.

Ejemplos de uso de Raft en Kafka

- Elección del líder: cuando un *broker* se une al clúster o un *broker* existente falla, Raft se encarga de elegir un nuevo líder de forma rápida y segura.
- Replicación de *logs:* Raft garantiza que todos los *brokers* tengan una copia actualizada de los *logs* de las particiones, lo que es fundamental para la durabilidad de los datos.
- Coordinación de la configuración del clúster: Raft se utiliza para coordinar cambios en la configuración del clúster, como la adición o eliminación de *topics.*

Ventajas de utilizar Raft en Kafka

- Mayor rendimiento: Raft ha demostrado ser más eficiente que ZooKeeper en términos de rendimiento y escalabilidad.
- Mayor simplicidad: la implementación de Raft es más sencilla, lo que facilita la comprensión y el mantenimiento del sistema.
- Mayor seguridad: Raft ofrece garantías más fuertes de seguridad y consistencia.

***Producers* (productores)**

- Función: Los *producers* son las aplicaciones o servicios que envían (producen) mensajes a Kafka. Estos mensajes se agrupan en categorías llamadas *topics.* Un productor puede enviar datos a un *topic* específico o a varios, según la configuración.

- Características:

  - Un *producer* puede elegir a qué partición de un *topic* enviar los mensajes. Si no se especifica una partición, Kafka puede asignar el mensaje a una partición utilizando una política predeterminada (por ejemplo, *round-robin* o basada en el valor de la clave del mensaje).

  - Los mensajes que un *producer* envía son inmutables una vez que se almacenan en Kafka, lo que significa que no se pueden modificar después de su publicación.

- Ejemplo: un sistema de procesamiento de órdenes de un comercio electrónico puede actuar como un *producer,* enviando información sobre cada nueva compra realizada (como el ID del cliente, el producto y el monto) a un *topic* en Kafka.

***Consumers* (consumidores)**

- Función: los *consumers* son las aplicaciones o servicios que leen (consumen) los mensajes de los *topics* en Kafka. Los consumidores se suscriben a uno o más *topics* y leen los mensajes de las particiones correspondientes.
- Características:
  - Los consumidores pueden leer los mensajes de manera secuencial desde las particiones.
  - Cada consumidor mantiene un *offset* para saber qué mensaje leer a continuación dentro de una partición. Este *offset* se guarda para garantizar que no se procesen mensajes repetidos.
  - Kafka permite agrupar consumidores en grupos de consumidores. En un grupo de consumidores, solo un miembro lee cada partición, lo que garantiza que los mensajes se procesen una sola vez de forma eficiente.
- Ejemplo: un sistema de análisis de ventas podría ser un consumidor que lea los mensajes de un *topic* que contiene datos de compras realizadas, procesando y almacenando la información en una base de datos.

***Topics* (temas)**

- Función: un *topic* en Kafka es un canal o categoría donde se organizan los mensajes. Los *topics* permiten que los productores publiquen mensajes y que los consumidores los lean posteriormente. Los mensajes dentro de un *topic* se almacenan de forma ordenada y secuencial.
- Características:
  - Un *topic* puede dividirse en particiones para permitir una mayor paralelización del procesamiento de los datos.
  - Cada *topic* tiene un nombre único para identificarlo.
  - Los *topics* permiten que múltiples productores y consumidores interactúen con el mismo conjunto de datos.
- Ejemplo: Un *topic* llamado «nuevas-ordenes» podría ser utilizado para almacenar todas las órdenes realizadas por los clientes.

***Partitions* (particiones)**

- Función: las particiones son la unidad de paralelización dentro de un *topic.* Un *topic* puede tener varias particiones, lo que permite que los datos sean distribuidos entre diferentes *brokers,* aumentando así la capacidad de procesamiento y mejorando el rendimiento.

- Características:
  - Las particiones permiten a Kafka distribuir los datos a través de múltiples *brokers* en el clúster, lo que ayuda a escalar Kafka horizontalmente.
  - Cada partición se identifica mediante un número de partición y cada mensaje dentro de una partición tiene un *offset* único, que actúa como un identificador dentro de la partición.
  - La orden de los mensajes dentro de una partición se mantiene, pero no necesariamente entre particiones.
- Ejemplo: si un *topic* tiene tres particiones, los mensajes de ese *topic* se dividen en tres conjuntos y cada conjunto se almacena en una partición diferente. Esto permite que diferentes consumidores puedan leer datos de distintas particiones simultáneamente.

Replicación

- Función: Kafka utiliza la replicación para garantizar la disponibilidad de los datos y la tolerancia a fallos. Cada partición de un *topic* puede tener réplicas almacenadas en diferentes *brokers.* Esto asegura que, si un *broker* falla, otro *broker* con una réplica de los datos pueda tomar el control.
- Características:
  - Cada partición tiene un líder que maneja las solicitudes de lectura y escritura para esa partición.
  - Los demás *brokers* que contienen réplicas de la partición son conocidos como seguidores.
  - Si el líder de una partición falla, uno de los seguidores se convierte en el nuevo líder.

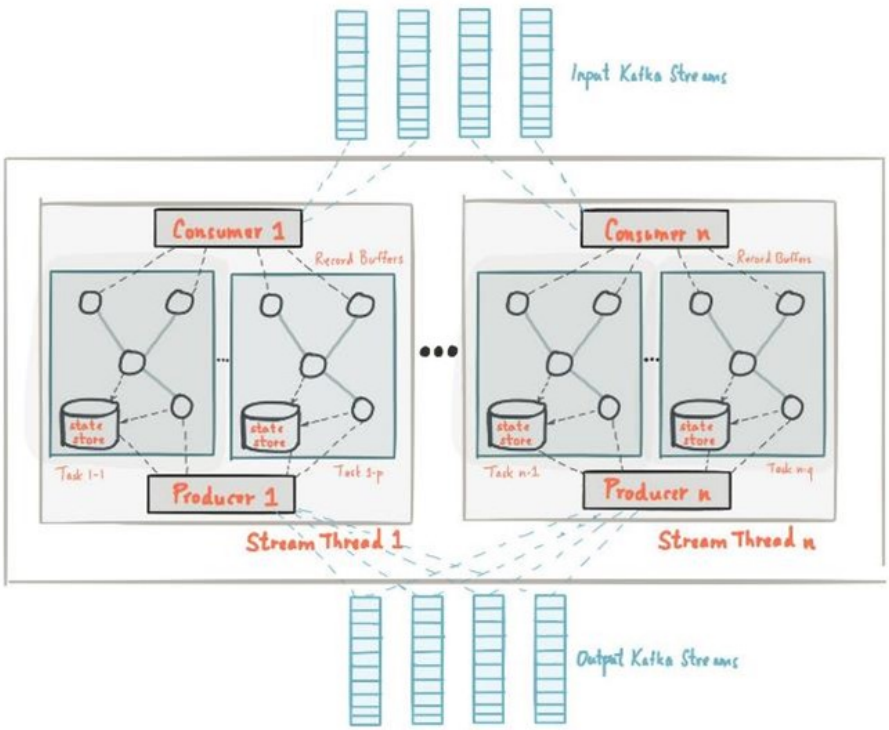

Figura 9. Diagrama de funcionamiento de replicación en Apache Kafka. Fuente: Kafka, s. f.

En el siguiente enlace, tienes la documentación oficial arquitectura:

https://kafka.apache.org/

Flujo de datos en Kafka: de los *producers* a los *consumers*

El proceso de manejo de los datos en Kafka sigue un flujo claro:

1. **Producción de datos.** El *producer* envía datos a Kafka, publicándolos en un *topic* específico. Cada mensaje tiene un *offset,* que es un número único que identifica el mensaje dentro de la partición.

2. **Almacenamiento de datos.** El *broker* recibe los datos y los almacena en la partición correspondiente dentro del *topic.* Kafka es altamente distribuido, lo que significa que los datos de un *topic* pueden ser replicados en diferentes *brokers* para asegurar que no se pierdan.

3. **Consumo de datos.** Los *consumers* se suscriben a uno o varios *topics* y leen los datos desde las particiones. Los consumidores mantienen un *offset* que les indica el punto exacto donde deben leer. Si un consumidor se detiene y vuelve más tarde, puede continuar desde donde lo dejó.

4. **Replicación de datos.** Para garantizar la alta disponibilidad y la tolerancia a fallos, Kafka replica las particiones de los *topics* en diferentes *brokers.* Si un *broker* falla, Kafka puede seguir operando gracias a la réplica de los datos en otros *brokers.*

Conceptos de escalabilidad y tolerancia a fallos

La arquitectura distribuida de Kafka le permite escalar y ser resiliente ante fallos, lo que lo convierte en una solución muy robusta para el manejo de datos en tiempo real:

- **Escalabilidad horizontal:** Kafka puede añadir más *brokers* para aumentar su capacidad de almacenamiento y procesamiento de datos. Las particiones pueden distribuirse entre diferentes *brokers* y los *producers* pueden enviar datos a cualquier *broker* disponible.

- **Tolerancia a fallos:** gracias a la replicación de particiones entre varios *brokers,* Kafka puede continuar funcionando incluso si uno o más *brokers* fallan. Cuando un *broker* cae, otro *broker* con una copia de los datos tomará su lugar.

- **Balanceo de carga:** Kafka distribuye las particiones de los *topics* entre los *brokers* de manera eficiente, lo que permite un balanceo de carga entre ellos. Esto significa que no hay un solo *broker* encargado de manejar todo el trabajo, sino que se distribuye entre varios.

> Ejemplo: si un *broker* con una partición activa se cae, el sistema puede seguir funcionando, utilizando las réplicas de esa partición en otros *brokers,* sin perder ningún dato.

Instalación de Apache Kafka

Apache Kafka es una plataforma distribuida de *streaming* de datos, que se usa ampliamente para procesar grandes volúmenes de datos en tiempo real. Para empezar a usar Kafka, primero necesitas instalarlo y configurarlo correctamente. En este apartado, veremos cómo hacerlo paso a paso e instalaremos y configuraremos Apache Kafka en tu entorno local o de desarrollo.

Requisitos previos para instalar Apache Kafka

Antes de comenzar con la instalación, asegúrate de tener lo siguiente instalado en tu sistema:

- **Java 8 o superior:** Kafka está basado en Java, por lo que necesitas tener instalado el JDK (Java Development Kit) en tu máquina.
  - Puedes verificar si Java está instalado ejecutando java-version en tu terminal.
  - Si no tienes Java, puedes descargarlo desde Oracle JDK (https://www.oracle.com/java/technologies/downloads/#java11?er=221886) o usar una distribución como OpenJDK (https://kafka.apache.org/).
- **Sistema operativo:** la instalación de Kafka puede realizarse en sistemas Linux, macOS o Windows. A continuación, se detalla cómo hacerlo en sistemas basados en UNIX (Linux/macOS) y en Windows.

Instalación en Linux / macOS

- **Descargar Kafka.** Visita la página oficial de Apache Kafka en https://kafka.apache.org/downloads y descarga la versión más reciente. El archivo estará en formato comprimido (.tgz). Sustituye x.y.z por la versión más reciente.

```
wget https://downloads.apache.org/kafka/x.y.z/kafka_2.13-x.y.z.tgz
```

- **Descomprimir el archivo.** Una vez descargado el archivo .tgz, descomprímelo en el directorio que prefieras:

```
tar -xvf kafka_2.13-x.y.z.tgz
```

```
cd kafka_2.13-x.y.z
```

- **Iniciar Zookeeper** (si es necesario). Kafka necesita Zookeeper para su funcionamiento en versiones anteriores a la 2.8.0. Para iniciar Zookeeper, puedes usar el script proporcionado en la distribución de Kafka. Esto iniciará Zookeeper en el puerto por defecto (2181):

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

- **Iniciar Kafka.** Una vez que Zookeeper esté corriendo, puedes iniciar el servidor de Kafka. Kafka utiliza su propio archivo de configuración (server.properties), pero puedes modificarlo si es necesario. Esto iniciará Kafka en el puerto 9092 por defecto:

```
bin/kafka-server-start.sh config/server.properties
```

La documentación de instalación la encuentrarás en el siguiente enlace:

https://kafka.apache.org/quickstart

Instalación en Windows

- **Descargar Kafka.** Visita la página oficial de Apache Kafka en su página oficial y descarga la versión más reciente. El archivo estará en formato comprimido (.tgz).
- **Instalar herramientas de descompresión.** Si no tienes un programa para descomprimir archivos .tgz, puedes usar 7-Zip o cualquier otra herramienta que soporte este formato.
- **Descomprimir y mover los archivos.** Extrae el archivo .tgz y mueve los archivos a un directorio de tu preferencia.
- **Instalar Zookeeper.** Kafka requiere Zookeeper para su funcionamiento. En sistemas Windows, puedes usar Zookeeper en el mismo directorio de Kafka. Asegúrate de haber descomprimido Kafka y de que estás en el directorio correcto. Esto iniciará Zookeeper en el puerto 2181:

```
.\bin\windows\zookeeper-server-start.bat config\zookeeper.properties
```

- **Iniciar Kafka.** Después de que Zookeeper esté en funcionamiento, inicia el servidor Kafka con el siguiente código (Kafka debería iniciarse en el puerto 9092 por defecto.).

```
.\bin\windows\kafka-server-start.bat config\server.properties
```

Guía de instalación para Windows: https://hevodata.com/learn/install-kafka-on-windows/

Configuración básica de Apache Kafka

Aunque Kafka viene con una configuración predeterminada, puedes personalizar su comportamiento modificando el archivo «server.properties». Algunos parámetros importantes que puedes configurar son:

- **listeners:** configura el puerto donde Kafka escucha las conexiones. Esto permite que Kafka escuche en todas las interfaces de red de la máquina en el puerto 9092:

```
properties

listeners=PLAINTEXT://0.0.0.0:9092
```

- **log.dirs:** define el directorio donde se almacenan los *logs* y los datos de Kafka. Asegúrate de que el disco tenga suficiente espacio.

```
properties

log.dirs=/tmp/kafka-logs
```

- **zookeeper.connect:** en versiones anteriores de Kafka, debes especificar la dirección de Zookeeper, pero en versiones más recientes que no dependen de Zookeeper, este parámetro ya no es necesario.

```
properties

zookeeper.connect=localhost:2181
```

- **num.partitions:** define el número de particiones predeterminadas por *topic.*

```
properties

num.partitions=3
```

Verificación de la instalación

Para verificar que Kafka y Zookeeper están funcionando correctamente, puedes realizar algunas pruebas básicas.

- **Verificar el estado de Zookeeper.** En el terminal, puedes usar el siguiente comando para ver si Zookeeper está corriendo (A) o también puedes intentar conectarte a Zookeeper usando la línea de comandos de Zookeeper (B):

A

```
bash

ps aux | grep zookeeper
```

B

```
bash

bin/zookeeper-client.sh
```

- **Verificar el estado de Kafka.** Para verificar que Kafka está corriendo, usa el siguiente comando:

```
bash

ps aux | grep Kafka
```

- **Crear un *topic* de prueba.** Puedes crear un *topic* de prueba con el siguiente comando en Kafka (este comando creará un *topic* llamado «test-topic» con tres particiones y un factor de replicación.

```
bash

bin/kafka-topics.sh --create --topic test-topic --partitions 3 --
replication-factor 1 --bootstrap-server localhost:9092
```

- **Enviar mensajes al *topic.*** Para enviar mensajes al *topic,* usa el productor de Kafka (después de ejecutar este comando, puedes escribir mensajes en la terminal y presionar «Enter» para enviarlos al topic):

```
bash

bin/kafka-console-producer.sh --topic test-topic --bootstrap-server
localhost:9092
```

- **Leer los mensajes del *topic.*** Usa el consumidor de Kafka para leer los mensajes del *topic.* Este comando leerá todos los mensajes del *topic* «test-topic»:

```
bash

bin/kafka-console-consumer.sh --topic test-topic --bootstrap-server
localhost:9092 --from-beginning
```

- **Desinstalación de Kafka** (si es necesario). Si alguna vez necesitas desinstalar Kafka, simplemente elimina el directorio donde lo descomprimiste. Además, si usaste Zookeeper, detén ambos servicios:

  - Detener Kafka: bash // bin/kafka-server-stop.sh
  - Detener Zookeeper: bash // bin/zookeeper-server-stop.sh

Principales operaciones en Apache Kafka

Como hemos visto, Apache Kafka es una plataforma robusta para manejar flujos de datos en tiempo real. En este apartado, exploraremos las principales operaciones que puedes realizar con Kafka.

Estas operaciones incluyen la creación de *topics,* la publicación de mensajes *(producers)* y la suscripción a esos mensajes *(consumers)*. A lo largo de esta explicación, veremos ejemplos de código para que puedas entender cómo interactuar con Kafka a nivel básico.

Creación de un *topic* en Kafka

Un *topic* en Kafka es como un canal de comunicación donde los datos son enviados y desde donde son consumidos. Los productores envían mensajes a un *topic* y los consumidores leen esos mensajes. La creación de un *topic* es un paso fundamental al trabajar con Kafka.

- **Comando para crear un *topic.*** Kafka proporciona un comando en su línea de comandos para crear *topics.* En la terminal, puedes usar el siguiente comando (este comando creará un topic en Kafka y estará disponible para que los productores envíen datos y los consumidores los lean):

```
bash

bin/kafka-topics.sh --create --topic nombre-del-topic --partitions 3 --
replication-factor 1 --bootstrap-server localhost:9092
```

- --topic nombre-del-topic. Define el nombre del *topic.*
- --partitions 3. Especifica el número de particiones para el *topic.* Las particiones permiten distribuir los mensajes entre varios servidores.
- --replication-factor 1. Especifica cuántas réplicas de cada partición deben existir. En este caso, se establece a 1, lo que significa que no hay réplicas.
- --bootstrap-server localhost:9092. Especifica el servidor de Kafka donde se creará el topic.

Enviar mensajes *(producer)*

En Kafka, un *producer* (productor) es el componente responsable de enviar mensajes a un *topic.* Los productores pueden enviar mensajes de manera asíncrona, lo que permite a Kafka manejar grandes volúmenes de datos de forma eficiente.

- **Ejemplo de *Producer* en Kafka (Usando Java):**

Para interactuar con Kafka, es común usar el cliente de Kafka en Java. Aquí tienes un ejemplo de código para enviar un mensaje a un *topic:*

- **Paso 1.** Agregar dependencias (si usas Maven). Primero, agrega la dependencia de Kafka en tu archivo pom.xml (si usas Maven):

```
xml

<dependency>

    <groupId>org.apache.kafka</groupId>

    <artifactId>kafka-clients</artifactId>

    <version>2.8.0</version>

</dependency>
```

- **Paso 2.** Código para enviar mensajes *(producer)* (este código enviará el mensaje «¡Hola, Kafka!» al *topic* llamado «nombre-del-topic». El productor envía el mensaje de forma asíncrona):

  - BOOTSTRAP_SERVERS_CONFIG. La dirección del servidor Kafka (en este caso, localhost:9092).
  - KEY_SERIALIZER_CLASS_CONFIG y VALUE_SERIALIZER_CLASS_CONFIG. Estos parámetros definen cómo Kafka serializa las claves y los valores de los mensajes. Usamos StringSerializer en este ejemplo.

```
java

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;
```

```
public class ProducerExample {

    public static void main(String[] args) {

        // Configuración del productor

        Properties props = new Properties();

        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

        // Crear el productor

        KafkaProducer<String, String> producer = new KafkaProducer(props);

        // Enviar un mensaje

        String topic = "nombre-del-topic";

        String mensaje = "¡Hola, Kafka!";

        producer.send(new ProducerRecord(topic, "clave1", mensaje));

        // Cerrar el productor

        producer.close();

        System.out.println("Mensaje enviado al topic " + topic);

    }

}
```

Instalar la librería kafka-python:

- **Paso 1.** Si aún no tienes instalada la librería, puedes instalarla usando pip:

```
bash

pip install kafka-python
```

- **Paso 2:** Código para enviar un mensaje (anotamos antes, la explicación del código):

  - KafkaProducer. Este es el productor de Kafka que se utiliza para enviar mensajes.
  - bootstrap_servers='localhost:9092'. Especifica la dirección de tu servidor Kafka. En este caso, estamos usando *localhost* en el puerto 9092.
  - value_serializer=lambda v: json.dumps(v).encode('utf-8'). Serializa el mensaje en formato JSON antes de enviarlo. Kafka solo puede manejar datos en formato de *bytes,* por lo que serializamos el mensaje y lo convertimos a *bytes* usando encode('utf-8').
  - producer.send(topic, value=mensaje). Envía el mensaje al *topic* especificado.
  - producer.flush(). Asegura que el mensaje se haya enviado antes de cerrar el productor. Esto es importante porque el envío de mensajes es asíncrono.
  - producer.close(). Cierra el productor de Kafka una vez que el mensaje ha sido enviado.

```
python

from kafka import KafkaProducer

import json

# Crear el productor de Kafka

producer = KafkaProducer(

    bootstrap_servers='localhost:9092', # Dirección del servidor de Kafka

    value_serializer=lambda v: json.dumps(v).encode('utf-8') # Serializar
el mensaje en formato JSON

)

# El nombre del topic donde se enviarán los mensajes

topic = 'nombre-del-topic'

# El mensaje a enviar

mensaje = {"mensaje": "¡Hola, Kafka desde Python!"}

# Enviar el mensaje al topic

producer.send(topic, value=mensaje)

# Asegurarse de que el mensaje ha sido enviado

producer.flush()

# Cerrar el productor

producer.close()

print(f"Mensaje enviado al topic '{topic}'")
```

Leer mensajes *(consumer)*

U n *consumer* (consumidor) en Kafka es responsable de leer mensajes desde un *topic.* Los consumidores pueden leer desde cualquier punto del *topic* y recibir los mensajes enviados en tiempo real.

El siguiente ejemplo muestra cómo crear un consumidor que lee los mensajes desde un *topic* específico en Kafka (usando Java):

- **Código para consumir mensajes *(consumer).*** Este código hará que el consumidor lea mensajes desde el topic "nombre-del-topic" y los imprima en la consola. El ciclo while (true) asegura que el consumidor siga escuchando nuevos mensajes en tiempo real (se añade antes la explicación del código):

  - GROUP_ID_CONFIG. Define el grupo de consumidores. Los consumidores de un mismo grupo comparten el trabajo y se aseguran de que cada mensaje sea leído solo una vez.

  - KEY_DESERIALIZER_CLASS_CONFIG y VALUE_DESERIALIZER_CLASS_CONFIG. Definen cómo Kafka deserializa las claves y los valores de los mensajes.

```
java

import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.kafka.clients.consumer.ConsumerConfig;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import java.util.Properties;

import java.util.Collections;



public class ConsumerExample {

    public static void main(String[] args) {

        // Configuración del consumidor

        Properties props = new Properties();

        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

        props.put(ConsumerConfig.GROUP_ID_CONFIG, "grupo-consumidor");

        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");

        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringDeserializer");
```

```
        // Crear el consumidor

        KafkaConsumer<String, String> consumer = new KafkaConsumer(props);

        consumer.subscribe(Collections.singletonList("nombre-del-topic"));

        // Leer los mensajes

        while (true) {

            for (ConsumerRecord<String, String> record :
consumer.poll(1000)) {

                System.out.println("Mensaje recibido: " + record.value());

            }

        }

    }

}
```

Código para leer mensajes *(consumer)* en Python:

- **Paso 1.** Instalar la librería kafka-python. Si aún no tienes instalada la librería, puedes instalarla usando pip (explicación previa del código):

- KafkaConsumer: este es el consumidor de Kafka que se utiliza para leer mensajes de un *topic.*
- 'nombre-del-topic': el nombre del *topic* desde el cual se desean leer los mensajes.
- bootstrap_servers='localhost:9092': dirección del servidor Kafka. Asegúrate de que la dirección y puerto sean correctos.
- group_id='grupo-consumidor': especifica el grupo de consumidores. Kafka asignará las particiones a los consumidores dentro de un grupo para que cada mensaje se procese solo una vez. Si el group_id es el mismo, todos los consumidores en ese grupo compartirán el trabajo.
- value_deserializer=lambda x: json.loads(x.decode('utf-8')): deserializa el mensaje del formato JSON a un objeto Python. Kafka envía los mensajes como *bytes,* por lo que necesitamos convertirlos a un formato que sea legible y útil en Python.
- for message in consumer: este bucle escucha continuamente los mensajes en el *topic.* Cada vez que se recibe un mensaje, el consumidor lo imprime en la consola.
- message.value: este es el contenido del mensaje recibido, después de ser deserializado.

```
bash

pip install kafka-python

Paso 2: Código para Leer Mensajes

python

from kafka import KafkaConsumer

import json

# Crear el consumidor de Kafka

consumer = KafkaConsumer(

    'nombre-del-topic', # Nombre del topic desde el que leer

    bootstrap_servers='localhost:9092', # Dirección del servidor de Kafka

    group_id='grupo-consumidor', # Grupo de consumidores

    value_deserializer=lambda x: json.loads(x.decode('utf-8')) #
Deserializar el mensaje JSON

)

# Leer los mensajes del topic

print("Esperando mensajes...")

for message in consumer:

    print(f"Mensaje recibido: {message.value}")

    # Aquí puedes realizar cualquier procesamiento con el mensaje

# El consumidor se mantiene escuchando nuevos mensajes en el topic
```

- **Listar *topics:***

Es posible que desees ver los *topics* disponibles en tu clúster de Kafka. Para eso, puedes usar el siguiente comando (este comando devolverá una lista de todos los *topics* disponibles en tu servidor Kafka):

```
bash

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

- **Eliminar un *topic:***

Si deseas eliminar un *topic,* puedes hacerlo con el siguiente comando (este comando eliminará el *topic* especificado de Kafka):

```
bash

bin/kafka-topics.sh --delete --topic nombre-del-topic --bootstrap-server
localhost:9092
```

Kafka en Confluent: más allá de Apache Kafka

Introducción

En las secciones anteriores, hemos explorado los fundamentos de Apache Kafka y su arquitectura. Ahora, nos centraremos en Confluent, una plataforma empresarial que se construye sobre Apache Kafka y ofrece una serie de características y herramientas adicionales para facilitar la implementación, gestión y escalado de aplicaciones basadas en Kafka.

¿Qué es Confluent?

Confluent es una plataforma de *streaming* de datos de código abierto que se basa en Apache Kafka. Proporciona una distribución empresarial de Kafka, junto con herramientas y servicios adicionales, para simplificar la implementación, gestión y escalado de aplicaciones de *streaming.* Confluent se ha convertido en el estándar de facto para las empresas que buscan construir plataformas de datos en tiempo real.

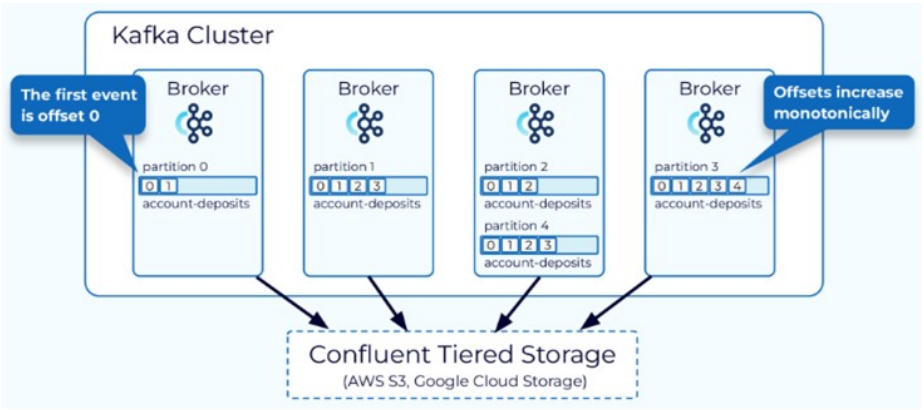

Figura 10. Ejemplo de funcionamiento Kafka en Confluent. Fuente: Rao, s. f.

Site oficial de Confluent Kafka: https://hevodata.com/learn/install-kafka-on-windows/

¿Por qué usar Confluent?

- **Simplicidad.** Confluent simplifica la instalación, configuración y gestión de clústeres de Kafka.
- **Escalabilidad.** Ofrece herramientas para escalar los clústeres de Kafka de manera horizontal y vertical.
- **Alta disponibilidad.** Garantiza una alta disponibilidad de los datos mediante la replicación y la tolerancia a fallos.
- **Seguridad.** Proporciona mecanismos de seguridad robustos para proteger los datos.
- **Integraciones.** Se integra fácilmente con otras tecnologías y herramientas del ecosistema de *big data.*
- **Gestión**. Ofrece una interfaz de usuario y herramientas de línea de comandos para gestionar los clústeres de Kafka.

Características clave de Confluent

- Confluent Platform: es la distribución empresarial de Kafka que incluye herramientas para la gestión, monitoreo y seguridad de los clústeres de Kafka.
- Confluent Cloud: es una versión totalmente gestionada de Confluent Platform, que permite a los usuarios desplegar y gestionar clústeres de Kafka en la nube sin necesidad de administrar la infraestructura subyacente.
- Confluent Control Center: es una interfaz web que proporciona una vista unificada de todos los clústeres de Kafka, permitiendo a los usuarios monitorear el rendimiento, configurar los *topics* y gestionar los usuarios.
- Confluent Schema Registry: es una herramienta para gestionar los esquemas de los mensajes, lo que garantiza la compatibilidad hacia adelante y hacia atrás de las aplicaciones.