# KAFKA

Apache Kafka es, en pocas palabras, un middleware de mensajería entre sistemas heterogéneos, el cual, mediante un sistema de colas (topics, para ser concreto) facilita la comunicación asíncrona, desacoplando los flujos de datos de los sistemas que los producen o consumen.

Supongamos que tenemos múltiples generadores de datos, ya sean servidores web, de bases de datos, un servidor de chat y que todos ellos tienen que almacenar sus datos en múltiples destinos, como pueden ser logs, métricas de rendimiento y monitorización, el carrito de la compra o los fallos ocurridos, lo que puede provocar una serie de dependencias de unos con otros. Para evitarlo, Kafka viene al rescate conectando todos los generadores de datos (productores) a Kafka y a su vez, a todos los consumidores de estos datos.

Diagrama

Descripción generada automáticamente

En concreto, se trata de una plataforma open source distribuida de transmisión de eventos/mensajes en tiempo real con almacenamiento duradero y que proporciona de base un alto rendimiento (capaz de manejar billones de peticiones al día, con una latencia inferior a 10ms), tolerancia a fallos, disponibilidad y escalabilidad horizontal (mediante cientos de nodos).

Dentro del vocabulario asociado a arquitecturas asíncronas basadas en productor/consumidor o publicador/suscriptor, se utiliza el mensaje para indicar el dato que viaja desde un punto a otro. En Kafka, además de utilizar el concepto mensaje, se emplea el término evento.

Como sistema de mensajes, sigue un modelo publicador-suscriptor. Su arquitectura tiene dos directivas claras:

* No bloquear los productores (para poder gestionar la [back pressure](https://youtu.be/K3axU2b0dDk), la cual sucede cuando un publicador produce más elementos de los que un suscriptor puede consumir).
* Aislar los productores y los consumidores, de manera que los productores y los consumidores no se conocen.

Hoy en día, Apache Kafka se utiliza como un sistema de mensajería, para ingesta datos, realizar procesado de datos en streaming y analítica de datos en tiempo real, así como en arquitectura de microservicios y sistemas IOT.

## PUBLICADOR / SUSCRIPTOR

Antes de entrar en detalle sobre Kafka, hay que conocer el **modelo publicador/suscriptor o productor/consumidor**.

Hay tres elementos que hay que tener realmente claros:

* Publicador (productor): genera un dato y lo coloca en un topic como un mensaje.
* topic (tema): almacén temporal/duradero que guarda los mensajes funcionando como una cola.
* Suscriptor (consumidor): recibe el mensaje.

Cabe destacar que un productor no se comunica nunca directamente con un consumidor, siempre lo hace a través de un topic:

Diagrama, Escala de tiempo

Descripción generada automáticamente

Antes de seguir adelante realiza el ejercicio 1 de la relación 4\_3Ejercicios

## CASO 0: HOLA KAFKA

Partimos de que tenemos zookeeper y los brokers arrancados y funcionando.

### CREAR UN TOPIC

A continuación, en un nuevo terminal (en cualquiera de los nodos), vamos a crear un topic mediante el comando kafka-topics.sh, utilizando el parámetro --create:

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-topic --bootstrap-server nodo1:9092*

Hemos puesto uno de los nodos, pero podemos poner cualquiera, y ejecutarlo el comando desde cualquiera de los equipos aunque ponga nodo1

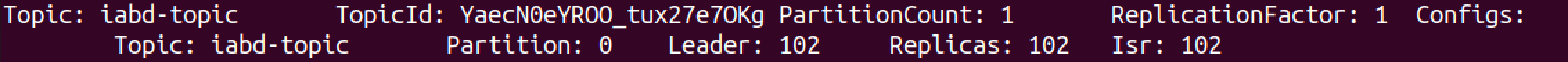
Si quisiéramos comprobar los topics que hemos creado, podemos obtener un listado mediante el parámetro --list:

*/opt/kafka/bin/kafka-topics.sh --list --bootstrap-server nodo1:9092*

Si queremos obtener la descripción del topic creado con la cantidad de particiones le pasamos el parámetro --describe:

*/opt/kafka/bin/kafka-topics.sh --describe --topic iabd-topic --bootstrap-server nodo1:9092*

Obteniendo la siguiente información:



### PRODUCIR MENSAJES

Para enviar un mensaje a un topic, ejecutaremos en un nuevo terminal un productor mediante el comando kafka-console-producer.sh. Por defecto, cada línea que introduzcamos será un evento separado que escribirá un mensaje en el topic (podemos pulsar CTRL+C en cualquier momento para cancelar):

*/opt/kafka/bin/kafka-console-producer.sh --topic iabd-topic --bootstrap-server nodo1:9092*

Así pues, escribimos los mensajes que queramos:

>Este es un mensaje

>Y este es otro

>Y el tercero

>

No pulsar CTRL+C de momento

### CONSUMIR MENSAJES

Y finalmente, en otro terminal, vamos a consumir los mensajes:

*/opt/kafka/bin/kafka-console-consumer.sh --topic iabd-topic --from-beginning --bootstrap-server nodo1:9092*

Al ejecutarlo veremos los mensajes que habíamos introducido antes (ya que hemos indicado la opción --from-beginning).

Si ahora volvemos a escribir en el productor, casi instantáneamente, aparecerá en el consumidor el mismo mensaje.

Ya podemos parar el productor y el consumidor con CTRL+C

## ELEMENTOS

Dentro de una arquitectura con Kafka, existen múltiples elementos que interactúan entre sí.

### TOPIC Y PARTICIONES

Un topic es un flujo particular de datos que funciona como una cola, almacenando de forma temporal o duradera los datos que se colocan en él.

Podemos crear tantos topics como queramos y cada uno de ellos tendrá un nombre unívoco.

Un topic se divide en particiones, las cuales se numeran, siendo la primera la 0. Al crear un topic podemos indicar la cantidad de particiones inicial, la cual podemos modificar a posteriori (en el archivo server.properties tenemos configurado que, por defecto, cada topic tenga una sola partición mediante la propiedad num.partitions=1, valor que evidentemente se puede cambiar).

Al crear un topic, si queremos indicar la cantidad de particiones, hemos de pasarle el parámetro --partitions y el número de particiones deseadas:

*/opt/kafka/bin/kafka-topics.sh --create topic iabd-topic-p3 --partitions 3 --bootstrap-server nodo1:9092*

Cada partición está ordenada, de manera que cada mensaje dentro de una partición tendrá un identificador incremental, llamado offset (desplazamiento).

Gráfico

Descripción generada automáticamente

Como podemos observar en la imagen, cada partición tiene sus propios offset (el offset 3 de la partición 0 no representa el mismo dato que el offset 3 de la partición 1).

Las particiones están ordenadas, pero el orden sólo se garantiza dentro de una partición (no entre particiones), es decir, el mensaje 7 de la partición 0 puede haber llegado antes, a la vez, o después que el mensaje 5 de la partición 1.

Los datos de una partición tienen un tiempo de vida limitado (retention period) que indica el tiempo que se mantendrán los mensajes antes de eliminarlos. Por defecto es de una semana. Además, una vez que los datos se escriben en una partición, no se pueden modificar (los mensajes son inmutables).

Para eliminar un topic, le pasaremos el parámetro --delete:

*kafka-topics.sh --delete --topic iabd-topic --bootstrap-server nodo1:9092*

Al borrar un topic, sus índices no se eliminan, de manera que volver a crear un topic con el mismo nombre es una mala idea ya que podemos obtener datos incongruentes.

### BROKERS

Un clúster de Kafka está compuesto de múltiples nodos conocidos como brokers, donde cada broker es un servidor de Kafka. Cada broker se identifica con un id, el cual debe ser un número entero.

Cada broker contiene un conjunto de particiones, de manera que un broker contiene parte de los datos, nunca los datos completos ya que Kafka es un sistema distribuido. Al conectarse a un broker del clúster (bootstrap broker), automáticamente nos conectaremos al clúster entero.

Por ejemplo, el siguiente gráfico muestra un clúster con tres brokers, de manera que el topic A está dividido en tres particiones, cada una de ellas residiendo en un broker diferente (no hay ninguna relación entre el número de la partición y el nombre del broker), y el topic B está a su vez dividido en dos particiones:

Interfaz de usuario gráfica

Descripción generada automáticamente

En el caso de haber introducido un nuevo topic con 4 particiones, uno de los brokers contendría dos particiones.

### FACTOR DE REPLICACIÓN

Para soportar la tolerancia a fallos, los topics deben tener un factor de replicación mayor que uno (normalmente se configura entre 2 y 3).

En la siguiente imagen podemos ver como tenemos 3 brokers, y un topic A con dos particiones y un factor de replicación de 2, de manera que cada partición crea una réplica de sí misma:

Diagrama

Descripción generada automáticamente con confianza baja

Para crear la configuración del gráfico, debemos indicar el factor de replicación mediante --replication-factor:

*/opt/kafka/bin/kafka-topics.sh --create --topic TopicA --partitions 2 --replication-factor 2 --bootstrap-server nodo1:9092*

Si se cayera el broker 102, Kafka podría devolver los datos al estar disponibles en los nodos 101 y 103.

#### RÉPLICA LÍDER

En cualquier instante, una determinada partición tendrá una única réplica líder, y esta réplica líder será la única que pueda recibir y servir los datos de una partición. La réplica líder es importante porque todas las lecturas y escrituras siempre van a esta réplica. El resto de brokers sincronizarán sus datos. En resumen, cada partición tendrá un líder y múltiples ISR (in-sync replica).

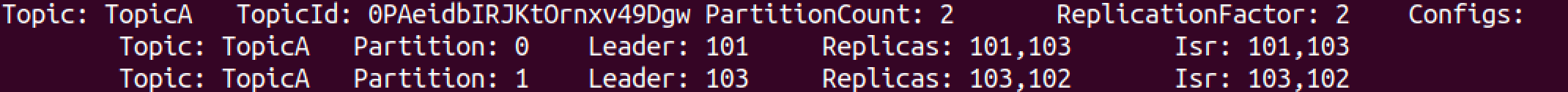
Diagrama

Descripción generada automáticamente con confianza media

Si se cayera el Broker 101 , entonces la partición 0 del Broker 102 se convertiría en la líder. Y cuando vuelva a funcionar el Broker 101, intentará volver a ser la partición líder.

Con el siguiente comando podemos ver las particiones, réplicas, quien es el líder…

*/opt/kafka/bin/kafka-topics.sh --describe --topic TopicA --bootstrap-server nodo1:9092*



### PRODUCTORES

Los productores escriben datos en los topics, sabiendo automáticamente el broker y la partición en la cual deben escribir. En el caso de un fallo de un broker, los productores automáticamente se recuperan y se comunican con el broker adecuado.

Si el productor envía los datos sin una clave determinada, Kafka utiliza un algoritmo de Round Robin, de manera que cada mensaje se va alternando entre los diferentes brokers.

Diagrama

Descripción generada automáticamente

Podemos configurar los productores para que reciban un ACK de las escrituras de los datos con los siguientes valores:

* ack=0: El productor no espera la confirmación (posible pérdida de datos), lo que se traduce en un envío asíncrono.
* ack=1: El productor espera la confirmación del líder (limitación de la pérdida de datos), de manera que los envíos son síncronos.
* ack=all: El productor espera la confirmación del líder y de todas las réplicas (sin pérdida de datos).

#### CLAVE DE MENSAJE

Los productores pueden enviar una clave con el mensaje (de tipo cadena, numérico, etc.…). Cuando la clave no se envía, ya hemos comentado que los datos se envían mediante Round Robin (primero Broker 101, luego el 102, el 103, etc... y vuelta al 101).

Si se envía la clave, todos los mensajes con la misma clave siempre irán a la misma partición. Por lo tanto, enviaremos una clave cuando necesitemos ordenar los mensajes por un campo específico (por ejemplo, el identificador de una operación).

Para enviar mensajes con la clave, desde el terminal, necesitamos indicar dos propiedades mediante --property:

* parse.key: si es true, obligatoriamente enviaremos la clave (por defecto es false)
* key.separator: carácter para separar la clave del valor, por ejemplo, :, ;, ....

Así pues, podemos enviar mensajes con clave mediante:

*/opt/kafka/bin/kafka-console-producer.sh --topic iabd-topic --property "parse.key=true" --property "key.separator=:" --bootstrap-server nodo1:9092*

Ahora ya podemos crear mensajes del tipo clave:valor

Por ejemplo:

customer\_id: {"customer\_id":"1", "customer\_fname": "Enaitz", "customer\_lname": "Ordoñana", "customer\_email": "ena.itz@gmail.com"}

### CONSUMIDORES

Los consumidores obtienen los datos de los topics y las particiones, y saben de qué broker deben leer los datos. En el caso de un fallo de un broker, los consumidores automáticamente se recuperan y se comunican con el broker adecuado.

Los datos se leen en orden dentro de cada partición, de manera que el consumidor no podrá leer, por ejemplo, los datos del offset 6 hasta que no haya leído los del offset 5. Además, un consumidor puede leer de varias particiones (se realiza en paralelo), pero el orden sólo se respeta dentro de cada partición, no entre particiones:

Imagen que contiene Tabla

Descripción generada automáticamente

#### GRUPO DE CONSUMIDORES

Un consumidor puede pertenecer a un grupo de consumidores, de manera que cada uno de los consumidores del grupo obtendrán una parte de los datos, es decir, una partición de un topic.

Por ejemplo, tenemos una aplicación compuesta de dos consumidores, formando un grupo de consumidores. El consumidor 1 lo hará de dos particiones, y el consumidor 2 lo hará de la tercera partición. También tenemos otra aplicación compuesta de tres consumidores, de manera que cada consumidor lo hará de cada una de las particiones. Finalmente, tenemos un tercer grupo de consumidores formado por un único consumidor que leerá las tres particiones. En conclusión, cada grupo de consumidores funciona como un único consumidor de manera que accede a todas las particiones de un topic.

En el caso de tener más consumidores que particiones, algunos consumidores no realizarán nada (no tiene mucho sentido)

Captura de pantalla de un celular con letras

Descripción generada automáticamente con confianza media

Los consumidores, por sí solos, no saben con qué partición se deben comunicar. Para ello, se utiliza un Group Coordinator y un Consumer Coordinator para asignar los consumidores a cada partición. Esta gestión la realiza Kafka.

#### PROBANDO LOS GRUPOS DE CONSUMIDORES

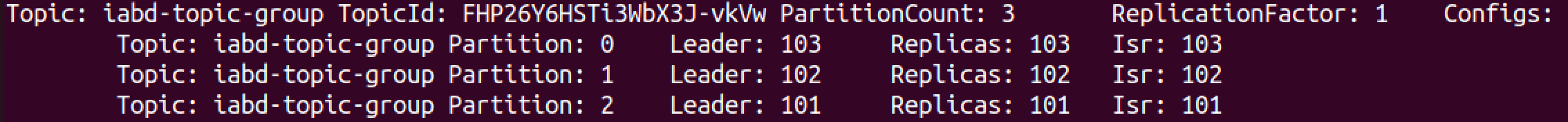
Vamos a simular el gráfico anterior mediante un ejemplo con el terminal. Primero crearemos un topic que contenga tres particiones:

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-topic-group --bootstrap-server nodo1:9092 --partitions 3*

Si comprobamos el estado del topic mediante:

*/opt/kafka/bin/kafka-topics.sh --describe --topic iabd-topic-group --bootstrap-server nodo1:9092*

Obtendremos la siguiente información:



A continuación, en dos nuevas terminales diferentes, vamos a crear dos consumidores que pertenezcan al mismo grupo de consumidores:

*/opt/kafka/bin/kafka-console-consumer.sh --topic iabd-topic-group --group iabd-app1 --bootstrap-server nodo1:9092*

Y finalmente, creamos un nuevo productor sobre el topic:

*/opt/kafka/bin/kafka-console-producer.sh --topic iabd-topic-group --bootstrap-server nodo1:9092*

Y si creamos varios mensajes en el productor, veremos cómo van llegando de manera alterna a los diferentes consumidores.

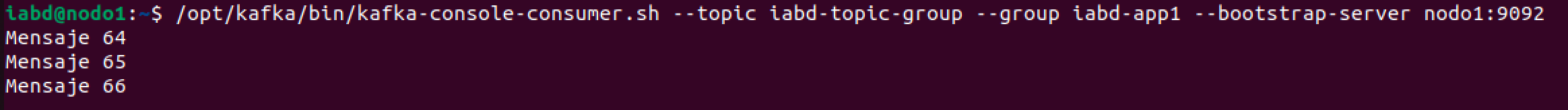
Cada partición irá a un consumidor (con mensajes cortos y con el productor de consola cuesta que cambie de partición al guardar el mensaje y hay que lanzar varios mensajes y varios productores (para con Ctrl + C y volver a lanzar para forzar el cambio de partición)).

Consumidor 1:

Texto

Descripción generada automáticamente

Consumidor 2:



Mediante el comando kafka-consumer-groups.sh podemos obtener información sobre los diferentes grupos de consumidores que tenemos creados, así como eliminarlos o resetear sus offsets.

Por ejemplo, si queremos listar los grupos de consumidores existentes ejecutaremos:

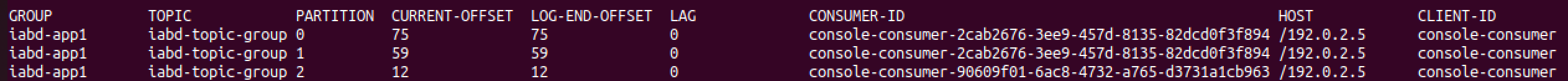
*/opt/kafka/bin/kafka-consumer-groups.sh --list --bootstrap-server nodo1:9092*

En cambio, si queremos obtener la información de un determinado grupo ejecutaremos:

*/opt/kafka/bin/kafka-consumer-groups.sh --describe --group iabd-app1 --bootstrap-server nodo1:9092*

Obteniendo información a destacar como:

* CURRENT-OFFSET: valor actual del offset
* LOG-END-OFFSET: offset del último mensaje de la partición
* LAG: cantidad de mensajes pendientes de leer



¿Qué sucederá si creamos un nuevo consumidor, pero con un grupo de consumidores diferente (por ejemplo, iabd-app2) y le pedimos que lea los mensajes desde el principio (mediante --from-beginning) ?

Que aparecerán todos los mensajes desde el principio.

Y si lo detenemos y volvemos a crear el mismo consumidor (también con el grupo de consumidores iabd-app2) ¿los vuelve a leer desde el principio también?

En esta ocasión, ya no recibirá ningún mensaje, ya que el primer consumidor hace commit de la lectura y el segundo al hacerlo desde el mismo grupo de consumidores ya tiene los mensajes previos marcados como leídos.

¿Y si detenemos todos los consumidores y seguimos creando mensajes en el productor?

Los mensajes se almacenan en el topic.

¿Y si arrancamos de nuevo un consumidor sobre el grupo de consumidores iabd-app2?

Que consumirá los mensajes que creados anteriormente y que todavía están sin leer.

#### OFFSETS DE CONSUMIDOR

Kafka almacena los offsets por el que va leyendo un grupo de consumidores, a modo de checkpoint, en un topic llamado \_\_consumer\_offsets.

Cuando un consumidor de un grupo ha procesado los datos que ha leído de Kafka, realizará un commit de sus offsets. Si el consumidor se cae, podrá volver a leer los mensajes desde el último offset sobre el que se realizó commit.

Por ejemplo, supongamos que tenemos un consumidor el cual ha hecho un commit tras el offset 4262. Tras el commit seguimos leyendo los siguientes mensajes: 4263, 4264, 4265 y de repente el consumidor se cae sin haber hecho commit de esos mensajes. Cuando el consumidor vuelva a funcionar, volverá a leer los mensajes desde el 4263, asegurándose que no se ha quedado ningún mensaje sin procesar.

Gráfico

Descripción generada automáticamente con confianza media

Los consumidores eligen cuando realizar el commit de los offsets:

* As most once: se realiza el commit del mensaje tan pronto como se recibe el mensaje. Si falla su procesamiento, el mensaje se perderá (y no se volverá a leer).
* At least once (opción más equilibrada): el commit se realiza una vez procesado el mensaje. Este enfoque puede resultar en un procesado duplicado de los mensajes, por lo que hemos de asegurarnos que el volver a procesar un mensaje no tendrá un impacto en el sistema.
* Exactly once: este método es igual que el anterior pero verifica que un mensaje se procese una única vez, sólo se puede conseguir utilizando flujos de trabajo mediante el API de Kafka Streams. Si necesitamos la interacción de Kafka con un sistema externo, como una base de datos, se recomienda esta opción que nos asegura que no habrá duplicados en la base de datos.

### DESCUBRIMIENTO DE BROKERS

Cada broker de Kafka es un bootstrap server, lo que significa que dicho servidor contiene un listado con todos los nodos del clúster, de manera que al conectarnos a un broker, automáticamente nos conectaremos al clúster entero.

Mediante esta configuración, cada broker conoce todos los brokers, topics y particiones del clúster.

Así pues, cuando un cliente se conecta a un broker, también realiza una petición de los metadatos, y obtiene un listado con todos los brokers. Tras ello, ya puede conectarse a cualquiera de los brokers que necesite:

Diagrama

Descripción generada automáticamente

### ZOOKEEPER

ZooKeeper es un servicio para mantener la configuración, coordinación y aprovisionamiento de aplicaciones distribuidas dentro del ecosistema de Apache.

En el caso de Kafka, Zookeeper:

* gestiona los brokers (manteniendo una lista de ellos).
* ayuda en la elección de la réplica líder.
* envía notificaciones a Kafka cuando hay algún cambio (por ejemplo, se crea un topic, se cae un broker, se recupera un broker, al eliminar un topic, etc.…).

Por todo ello, Kafka no puede funcionar sin Zookeeper.

En un entorno real, se instalan un número impar de servidores Zookeeper (3, 5, 7). Para su gestión, Zookeeper define un líder (gestiona las escrituras) y el resto de los servidores funcionan como réplicas de lectura.

Diagrama

Descripción generada automáticamente

Pese a su dependencia, los productores y consumidores no interactúan nunca con Zookeeper, sólo lo hacen con Kafka.

### DECISIONES DE RENDIMIENTO

Cada partición puede manejar un rendimiento de unos pocos MB. Al añadir más particiones, obtendremos mejor paralelización y por tanto, mejor rendimiento. Además, podremos ejecutar más consumidores en un grupo. Pero el hecho de añadir más brokers al clúster para que las particiones los aprovechen, provocará que Zookeeper tenga que realizar más elecciones y que Kafka tenga más ficheros abiertos.

Una propuesta es:

* Si nuestro clúster es pequeño (menos de 6 brokers), crear el doble de particiones que brokers.
* Si tenemos un clúster grande (más de 12 brokers), crear la misma cantidad de particiones que brokers.
* Ajustar el número de consumidores que se ejecutan en paralelo en los picos de rendimiento.

De todas formas siempre, hay que realizar pruebas de rendimiento con diferentes configuraciones.

Respecto al factor de replicación, debería ser, al menos 2, siendo 3 la cantidad recomendada (es necesario tener al menos 3 brokers) y no sobrepasar de 4. Cuanto mayor sea el factor de replicación (RF):

* El sistema tendrá mejor tolerancia a fallos (se pueden caer RF-1 brokers).
* Pero tendremos mayor replicación (lo que implicará una mayor latencia si acks=all).
* Y también ocupará más espacio en disco.

Respecto al clúster, se recomienda que un broker no contenga más de 2000-4000 particiones (entre todos los topics de ese broker). Además, un clúster de Kafka debería tener un máximo de 20.000 particiones entre todos los brokers, ya que si se cayese algún nodo, Zookeeper necesitaría realizar muchas elecciones de líder.

## CASO 1: KAFKA Y PYTHON

Para poder producir y consumir mensajes desde Python necesitamos instalar la librería kafka-python:

*sudo apt install python3-pip*

*pip install kafka-python*

Vamos a crear un topic:

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-topic-caso1 --partitions 3 --bootstrap-server nodo1:9092*

### KAFKACONSUMER

Vamos a crear un consumidor, mediante un KafkaConsumer, que escuche de nuestro servidor de Kafka:

*nano consumer.py*

***consumer.py***

from kafka import KafkaConsumer

import json

consumer = KafkaConsumer(

'iabd-topic-caso1',

auto\_offset\_reset='earliest',

enable\_auto\_commit=True,

group\_id='iabd-grupo-1',

value\_deserializer=lambda m: json.loads(m.decode('utf-8')),

bootstrap\_servers=['nodo1:9092'])

for m in consumer:

print(m.value)

Si al ejecutar consumer.py da el error:

**ModuleNotFoundError: No module named 'kafka.vendor.six.moves'**

Es porque hay algún error con la versión 3.12 de python y para solucionarlo:

En la terminal ejecutamos el comando:

pip install kafka-python-ng

Ejecutamos el consumidor: python3 consumer.py

Si el error persiste al inicio del archivo: consumer.py añadimos el siguiente código:

import six

import sys

if sys.version\_info >= (3, 12, 0):

sys.modules['kafka.vendor.six.moves'] = six.moves

Al ejecutar el consumidor no debería dar error.

Al crear el consumidor, configuramos los siguientes parámetros:

* en el primer parámetro indicamos el topic desde el que vamos a consumir los mensajes
* bootstrap\_servers: listado de brokers de Kafka. En este caso nos conectamos al servidor nodo1, pero podemos poner la IP de varios brokers por si acaso alguno está caído.
* auto\_offset\_reset: le indica al consumidor desde donde empezar a leer los mensajes si se cae: earliest se moverá hasta el mensaje más antiguo y latest al más reciente.
* enable\_auto\_commit: si True, el offset del consumidor realizará periódicamente commit en segundo plano.
* value\_deserializer: método utilizado para deserializar los datos. En este caso, transforma los datos recibidos en JSON.

### KAFKAPRODUCER

Y para el productor, mediante un [KafkaProducer](https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html), vamos a enviar 10 mensajes en formato JSON mediante el método [send](https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html#kafka.KafkaProducer.send), pasándole como primer parámetro el topic y luego el mensaje en sí:

*nano producer.py*

***producer.py***

from kafka import KafkaProducer

from json import dumps

import time

producer = KafkaProducer(

value\_serializer=lambda m: dumps(m).encode('utf-8'),

bootstrap\_servers=['nodo1:9092'])

for i in range(10):

producer.send("iabd-topic-caso1", value={"nombre": "producer " + str(i)})

producer.flush()

producer.flush() sirve para garantizar que cualquier mensaje que haya sido aceptado por el productor hasta ese momento se envíe y procese antes de que la aplicación continúe).

Tras ejecutar ambos programas en pestañas diferentes, en la salida del consumidor recibiremos:

{'nombre': 'producer 0'}

{'nombre': 'producer 1'}

{'nombre': 'producer 2'}

{'nombre': 'producer 3'}

{'nombre': 'producer 4'}

{'nombre': 'producer 5'}

{'nombre': 'producer 6'}

{'nombre': 'producer 7'}

{'nombre': 'producer 8'}

{'nombre': 'producer 9'}

## CASO 2: KAFKA Y PYTHON

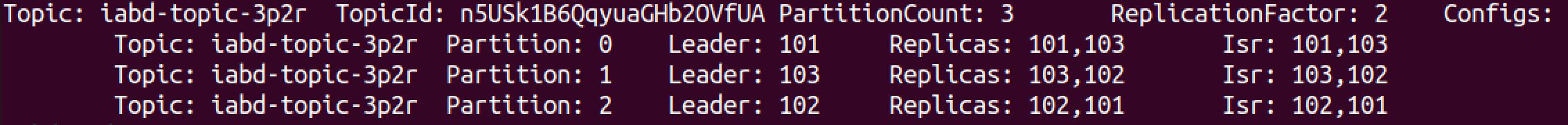
Cuando estemos trabajando en un clúster es conveniente al crear un topic hay que indicar el número de particiones y el factor de replicación:

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-topic-3p2r --bootstrap-server nodo1:9092 --partitions 3 --replication-factor 2*

Si ahora queremos obtener la información del topic:

*/opt/kafka/bin/kafka-topics.sh --describe --topic iabd-topic-3p2r --bootstrap-server nodo1:9092*

Podemos observar cómo cada partición tiene la partición líder en un broker distinto y en qué brokers se encuentran las réplicas:



Respecto al código Python, va a ser el mismo que hemos visto antes pero modificando:

* el nombre del topic
* la lista de boostrap\_servers (aunque podríamos haber dejado únicamente el nodo principal, ya que Kafka le comunica al cliente el resto de los nodos del clúster, es una buena práctica por si el nodo al que nos conectamos de manera explícita está caído).

*nano producer-cluster.py*

***producer-cluster.py***

from kafka import KafkaProducer

from json import dumps

import time

producer = KafkaProducer(

value\_serializer=lambda m: dumps(m).encode('utf-8'),

bootstrap\_servers=['nodo1:9092','nodo2:9092','nodo3:9092'])

for i in range(10):

clave = ("iabd" + str(i % 3)).encode('utf-8')

producer.send("iabd-topic-3p2r", value={"nombre": "producer " + str(i)}, key=clave)

producer.flush()

En el consumidor, además hemos modificado la forma de mostrar los mensajes para visualizar más información:

*nano consumer-cluster.py*

***consumer-cluster.py***

from kafka import KafkaConsumer

from json import loads

consumer = KafkaConsumer(

'iabd-topic-3p2r',

auto\_offset\_reset='earliest',

enable\_auto\_commit=True,

group\_id='iabd-grupo-1',

value\_deserializer=lambda m: loads(m.decode('utf-8')),

bootstrap\_servers=['nodo1:9092','nodo2:9092','nodo3:9092'])

for m in consumer:

print(f"P:{m.partition} O:{m.offset} K:{m.key} V:{m.value}")

Hemos enviado los mensajes con una clave de partición, para asegurarnos a que partición van, con el atributo key del método send:

P:0 O:0 K:b'iabd0' V:{'nombre': 'producer 0'}

P:0 O:1 K:b'iabd0' V:{'nombre': 'producer 3'}

P:0 O:2 K:b'iabd0' V:{'nombre': 'producer 6'}

P:0 O:3 K:b'iabd0' V:{'nombre': 'producer 9'}

P:1 O:0 K:b'iabd1' V:{'nombre': 'producer 1'}

P:1 O:1 K:b'iabd1' V:{'nombre': 'producer 4'}

P:1 O:2 K:b'iabd1' V:{'nombre': 'producer 7'}

P:1 O:3 K:b'iabd1' V:{'nombre': 'producer 8'}

P:2 O:0 K:b'iabd2' V:{'nombre': 'producer 2'}

P:2 O:1 K:b'iabd2' V:{'nombre': 'producer 5'}

P:2 O:2 K:b'iabd2' V:{'nombre': 'producer 8'}

## CASO 3: DE AEMET A S3 Y MONGODB

Vamos a desacoplar los productores de los consumidores usando Kafka.

Para este caso de uso, vamos a recoger datos de un servicio REST, en concreto, datos climatológicos de la Agencia Estatal de METeorología (AEMET) y los vamos a persistir tanto en MongoDB como AWS en S3.

Vamos a realizar llamadas a la URL <https://www.el-tiempo.net/api/json/v2/provincias/01/municipios/01059>, en la cual vamos a recuperar los datos del municipio de Vitoria-Gasteiz (01159) que pertenece a la provincia de Araba (01), obteniendo datos similares a:

{

"origin": {

"productor": "Agencia Estatal de Meteorología - AEMET. Gobierno de España",

"web": "https://www.aemet.es",

...

},

"title": "elTiempo.net | El tiempo en Vitoria-Gasteiz (Araba/Álava)",

...

"municipio": {

"CODIGOINE": "01059000000",

"ID\_REL": "1010590",

"COD\_GEO": "01001",

"CODPROV": "01",

"NOMBRE\_PROVINCIA": "Araba\/\u00c1lava",

"NOMBRE": "Vitoria-Gasteiz",

"POBLACION\_MUNI": 242082,

"SUPERFICIE": 27681.2008,

"PERIMETRO": 111412,

...

},

"fecha": "2024-02-08",

"stateSky": {

"description": "Cubierto",

"id": "16n"

},

"temperatura\_actual": "10",

"temperaturas": {

"max": "14",

"min": "8"

},

"humedad": "68",

"viento": "13",

"precipitacion": "0",

"lluvia": "95",

...

}

De todos los datos recuperados, nos vamos a centrar en recuperar la ciudad, la temperatura y la humedad.

### PRODUCTOR BRONZE

Tenemos que:

* enviar el resultado de nuestra consulta a un topic que llamaremos iabd-aemet-bronze.
* generar un nuevo mensaje JSON más pequeño con la información que nos interesa, y enviar este nuevo mensaje a un topic que llamaremos iabd-aemet-silver.

Así pues, primero creamos los topics:

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-aemet-bronze --bootstrap-server nodo1:9092*

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-aemet-silver --bootstrap-server nodo1:9092*

Y a continuación desarrollamos el productor, que hará uso de la librería request:

*nano productorBronze.py*

***productorBronze.py***

from kafka import KafkaProducer

from json import dumps

from datetime import datetime

import time

import requests

# Creamos el productor de Kafka

producer = KafkaProducer(

bootstrap\_servers=['nodo1:9092'])

url\_aemet = "https://www.el-tiempo.net/api/json/v2/provincias/01/municipios/01059"

while True:

r = requests.get(url\_aemet)

resp\_json = r.json()

producer.send("iabd-aemet-bronze", value=dumps(resp\_json).encode('utf-8'))

# Creamos el nuevo mensaje JSON con los datos

temp = resp\_json["temperatura\_actual"]

humedad = resp\_json["humedad"]

ciudad = resp\_json["municipio"]["NOMBRE"]

fecha = datetime.now()

fecha\_str = dumps(fecha, default=str)

datos\_json = {

"fecha": fecha,

"ciudad": ciudad,

"temp": temp,

"humedad": humedad

}

producer.send("iabd-aemet-silver", value=dumps(datos\_json, default=str).encode('utf-8'))

time.sleep(60)

### CONSUMIDOR BRONZE

En el consumidor del topic iabd-aemet-bronze vamos a recuperar las peticiones REST almacenadas en JSON y las persistiremos en S3.

Para poder conectarnos a S3 primero hay que configurar aws cli en Ubuntu:

*sudo apt install awscli*

Si no podemos instalar aws cli porque no encuentra el paquete, el error que nos da es:

***E: El paquete «awscli» no tiene un candidato para la instalación***

Instalamos aws cli manualmente siguiendo los pasos:

Paso1:Descarga AWS CLI:

*curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"*

Paso2:Descomprimimos el archivo:

*unzip awscliv2.zip*

Paso3:Ejecutamos el instalador:

*sudo ./aws/install*

Paso4: Verificamos la instalación

*aws --version*

*nano ~/.aws/credentials*

***~/.aws/credentials***

[default]

aws\_access\_key\_id = TU\_ACCESS\_KEY

aws\_secret\_access\_key = TU\_SECRET\_KEY

aws\_session\_token = TU\_SESSION\_TOKEN

Tienes que tener en cuenta que cada vez que reinicies el laboratorio de AWS tus claves cambian.

En este caso, como el mensaje ya está en formato JSON y no queremos tratarlo, no hace falta deserializarlo ni volverlo a serializar.

*nano consumidorBronze.py*

***consumidorBronze.py***

from kafka import KafkaConsumer

from datetime import datetime

import boto3

# Consumidor Kafka

consumer = KafkaConsumer(

'iabd-aemet-bronze',

enable\_auto\_commit=True,

group\_id='iabd-caso3',

bootstrap\_servers=['nodo1:9092'])

# Conexión con S3

s3r = boto3.resource('s3', region\_name='us-east-1')

bucket = s3r.Bucket('kafka-caso3')

for m in consumer:

resp\_json = m.value

# Creamos el nombre del fichero con la fecha y lo metemos en la carpeta bronze

nom\_fichero = "bronze/" + datetime.now().isoformat() + ".json"

bucket.put\_object(Key=nom\_fichero, Body=resp\_json)

### CONSUMIDOR SILVER Y PRODUCTOR GOLD

El siguiente paso es más complejo:

1. crearemos un consumidor para los mensajes de la cola silver
2. y almacenamos los mensajes en S3
3. además, el mismo mensaje lo insertaremos en MongoDB en la colección caso3
4. y cuando tengamos 10 mensajes crearemos uno nuevo con el cálculo de la temperaturas y humedades medias
5. para finalmente producir dicho mensaje al topic iabd-aemet-gold.

Así pues, primero creamos el topic gold:

*/opt/kafka/bin/kafka-topics.sh --create --topic iabd-aemet-gold --bootstrap-server nodo1:9092*

Si no tienes MongoDB instalado puedes instalarlo con docker (estoy usando la versión latest pero si os da problemas con VirtualBox podeis usar una versión de mongo anterior):

*sudo apt-get install docker-compose*

*nano docker-compose.yml*

***docker-compose.yml***

services:

mongodb:

ports:

- "27017:27017"

image: mongo:latest

Una vez creado el archivo, construimos el contenedor mediante:

*sudo docker-compose -p nifimongodb up -d*

Vamos a instalar también mongosh por si queremos hacer algunas comprobaciones después:

*wget -qO - https://www.mongodb.org/static/pgp/server-5.0.asc | gpg --dearmor | sudo tee /usr/share/keyrings/mongodb-archive-keyring.gpg*

*echo "deb [signed-by=/usr/share/keyrings/mongodb-archive-keyring.gpg] http://repo.mongodb.org/apt/ubuntu $(lsb\_release -cs)/mongodb-org/5.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list*

*sudo apt-get update*

*sudo apt-get install -y mongodb-mongosh*

Ahora que ya tenemos todos instalado y configurado, vamos a ir creando el fichero consumidorSilverProductorGold.py por partes. Para empezar, colocamos las dependencias y la función para transformar un diccionario JSON con todo el contenido en String a diferentes tipos (float, ISODate, ...):

*nano consumidorSilverProductorGold.py*

***consumidorSilverProductorGold.py***

from kafka import KafkaConsumer

from kafka import KafkaConsumer, KafkaProducer

from datetime import datetime

from json import loads, dumps

from pymongo import MongoClient

import boto3

import pandas as pd

import numpy as np

# Convierte un diccionario JSON con todo el contenido en String a diferentes tipos (float, ISODate, ...)

def esquema\_json(dct):

result = {}

if ('fecha' in dct):

result["fecha"] = datetime.fromisoformat(dct["fecha"])

if 'temp' in dct:

result["temp"] = float(dct["temp"])

if 'humedad' in dct:

result["humedad"] = float(dct["humedad"])

if 'ciudad' in dct:

result["ciudad"] = dct["ciudad"]

return result

A continuación definimos las conexiones necesarias para los pasos 1 (consumir Kafka en el topic silver), 2 (enviar a S3), 3 (persistir en MongoDB) y 5 (producir Kafka en el topic gold):

# Paso 1 - Consumir Silver

consumer = KafkaConsumer(

'iabd-aemet-silver',

enable\_auto\_commit=True,

group\_id='iabd-caso3',

bootstrap\_servers=['nodo1:9092'])

# Paso 2 - Conexión con S3

s3r = boto3.resource('s3', region\_name='us-east-1')

bucket = s3r.Bucket('kafka-caso3')

# Paso 3 - Conexión con MongoDB

clienteMongo = MongoClient('mongodb://localhost:27017')

# Paso 5 - Producir Gold

producer = KafkaProducer(

bootstrap\_servers=['nodo1:9092'])

Y para cada mensaje que consumimos, realizamos los pasos 2 (enviar a S3), 3 (persistir en MongoDB), 4 (calcular los datos agregados) y 5 (producir en Kafka en gold):

cantidad = 1

mensajes = []

for m in consumer:

resp\_json = m.value

doc\_json = loads(resp\_json, object\_hook=esquema\_json)

mensajes.append(doc\_json)

# Paso 2 - Metemos en S3 cada mensaje

nom\_fichero = "silver/" + datetime.now().isoformat() + ".json"

bucket.put\_object(Key=nom\_fichero, Body=resp\_json)

# Paso 3 - Lo insertamos en MongoDB

colcaso3 = clienteMongo.iabd.caso3

resultado = colcaso3.insert\_one(doc\_json)

# Guardamos 10 mensajes

cantidad = cantidad + 1

if cantidad == 11:

# Paso 4 - realizamos el cálculo

# Usamos pandas

pd\_mensajes = pd.DataFrame(mensajes)

pd\_mensajes.temp = pd\_mensajes['temp'].astype('float')

pd\_mensajes.humedad = pd\_mensajes['humedad'].astype('float')

pd\_agg = pd\_mensajes.groupby("ciudad").agg({"fecha":"max", "temp":"mean", "humedad":"mean"})

pd\_agg.reset\_index(inplace=True)

json\_gold = pd\_agg.to\_json(orient="records")

print(json\_gold)

# Paso 5 - producimos el mensaje a Kafka

producer.send("iabd-aemet-gold", value=dumps(json\_gold).encode('utf-8'))

# vaciamos la lista e inicializamos el contador

mensajes = []

cantidad = 1

Si en un terminal creamos un consumidor del topic iabd-aemet-gold:

*/opt/kafka/bin/kafka-console-consumer.sh --topic iabd-aemet-gold --from-beginning --bootstrap-server nodo1:9092*

Después de lanzar el resto de los consumidores y productores:

*python3 consumidorBronze.py*

*python3 productorBronze.py*

*python3 consumidorSilverProductorGold.py*

Veremos que los mensajes que llegan son similares a:

"[{\"ciudad\":\"Vitoria-Gasteiz\",\"fecha\":1707858046506,\"temp\":11.0,\"humedad\":83.6}]"

También hay que probar en S3 y en MongoDB que se están guardando los mensajes.

Ya podemos parar todos los consumidores y productores.

## CASO 4: KAFKA CONNECT

Kafka Connect permite importar/exportar datos desde/hacia Kafka, facilitando la integración en sistemas existentes mediante alguno del centenar de conectores disponibles.

Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente

Los elementos que forman Kafka Connect son:

* Conectores fuente (source), para obtener datos desde las fuentes de datos (E en ETL)
* Conectores destino (sink) para publicar los datos en los almacenes de datos (L en ETL).

Estos conectores facilitan que desarrolladores no expertos puedan trabajar con sus datos en Kafka de forma rápida y fiable, de manera que podamos introducir Kafka dentro de nuestros procesos ETL.

Vamos a realizar un ejemplo muy sencillo leyendo datos de una base de datos para meterlos en Kafka (en un topic, en formato JSON).

Para ello, utilizaremos la base de datos de retail\_db que tenemos en mysql.

Si en nuestra máquina no tenemos acceso a dicha base de datos podemos instalar mariadb y crear la base de datos rápidamente:

*sudo apt install mariadb-server*

*sudo mysql\_secure\_installation*

Para entrar en la base de datos:

*sudo mariadb*

Tenemos que crear otro usuario que no sea root y darle todos los permisos:

*>CREATE USER 'admin'@'localhost' IDENTIFIED BY 'Password0';*

*>GRANT ALL PRIVILEGES ON \* . \* TO 'admin'@'localhost';*

>exit

Ejecutamos el script de creación de la base de datos:

*sudo mariadb < crear\_db.sql*

Ahora que ya tenemos todo vamos a cargar en Kafka los datos de la tabla categories, aunque primero vamos a entrar en mariadb para ver su estructura.

*sudo mariadb*

*>use retail\_db*

*>describe categories;*

Imagen que contiene circuito, mujer, computadora, azul

Descripción generada automáticamente

### CONFIGURACIÓN

Cuando ejecutamos Kafka Connect, le debemos pasar un archivo de configuración.

Dependiendo de si estamos trabajando con Kafka Connect en modo standalone o distribuido tenemos los ficheros connect-standalone.properties o connect-distributed.properties (que por defecto tienen los conectores para JSON)

Los conectores se incluyen en Kafka como plugins. Para ello, primero hemos de indicarle a Kafka donde se encuentran, indicando en el archivo de configuración la ruta donde los almacenaremos.

Por lo tanto tenemos que añadir al fichero de configuración la siguiente propiedad:

*nano /opt/kafka/config/connect-standalone.properties*

***connect-standalone.properties***

plugin.path=/opt/kafka/plugins

### EXTRAYENDO DATOS MEDIANTE KAFKA CONNECT

Necesitamos un fichero de configuración para acceder a mariadb utilizando un conector fuente de JDBC:

*nano /opt/kafka/config/mysql-source-connector.properties*

***mysql-source-connector.properties***

name=mysql-source

connector.class=io.confluent.connect.jdbc.JdbcSourceConnector

tasks.max=1

connection.url=jdbc:mysql://localhost/retail\_db

connection.user=iabd

connection.password=iabd

table.whitelist=categories

mode=incrementing

incrementing.column.name=category\_id

topic.prefix=iabd-retail\_db-

Antes de ponerlo en marcha, debemos descargar el conector (<https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc>) y colocar la carpeta descomprimida dentro de nuestra carpeta de plugins /opt/kafka/plugins y colocar el driver de MySQL (mysql-connector-j-8.0.31.jar) en la carpeta /opt/kafka/lib.

A continuación ya podemos ejecutar Kafka Connect mediante el comando connect-standalone.sh:

*/opt/kafka/bin/connect-standalone.sh /opt/kafka/config/connect-standalone.properties /opt/kafka/config/mysql-source-connector.properties*

Si ahora arrancamos un consumidor sobre el topic iabd-retail\_db-categories:

*/opt/kafka/bin/kafka-console-consumer.sh --topic iabd-retail\_db-categories --from-beginning --bootstrap-server localhost:9092*

Veremos que aparecen todos los datos que teníamos en la tabla en formato JSON (como no hemos indicado otro formato de salida lo saca en JSON):

{"schema":{

"type":"struct","fields":[

{"type":"int32","optional":false,"field":"category\_id"},

{"type":"int32","optional":false,"field":"category\_department\_id"},

{"type":"string","optional":false,"field":"category\_name"}],

"optional":false,"name":"categories"},

"payload":{"category\_id":1,"category\_department\_id":2,"category\_name":"Football"}}

{"schema":{

"type":"struct","fields":[

{"type":"int32","optional":false,"field":"category\_id"},

{"type":"int32","optional":false,"field":"category\_department\_id"},

{"type":"string","optional":false,"field":"category\_name"}],

"optional":false,"name":"categories"},

"payload":{"category\_id":2,"category\_department\_id":2,"category\_name":"Soccer"}}

...