

Apunts CE_5075 6.1

lloc: [Institut d'Ensenyaments a Distància de les Illes
Balears](#)
Curs: Big data aplicat
Llibre: Apunts CE_5075 6.1

Imprès per: Carlos Sanchez Recio
Data: dilluns, 24 de febrer 2025, 20:20

Taula de continguts

1. Ingestió de dades en temps real

2. Què és Apache Kafka

3. Instal·lació i primeres passes amb Kafka

4. Produir i consumir esdeveniments amb Python

4.1. Emprant esdeveniments JSON

4.2. Grups de consumidors

5. Kafka Connect

5.1. Workers i tasques

5.2. Cas pràctic 1: de fitxer a fitxer

5.3. Cas pràctic 2: de MySQL a HDFS (part del source)

5.4. Cas pràctic 2: de MySQL a HDFS (part del sink)

5.5. Cas pràctic 3: xarxes socials

5.6. Transformacions

6. Kafka Streams

7. Apache Flink

1. Ingestió de dades en temps real

Ja sabem que en els sistemes de *big data* és fonamental poder gestionar dades provinents de fonts molt diverses. En entorns tradicionals, les dades es recollien, es processaven en lots (processos ETL -*Extraction, Transformation, Load*-), es guardaven en un magatzem de dades i, finalment, se'n feia una anàlisi. No obstant això, avui en dia, moltes d'aquestes fonts generen dades de manera contínua, com per exemple sensors IoT, aplicacions mòbils, xarxes socials, bases de dades transaccionals, *logs* de servidors, entre d'altres. En aquest context, per poder aprofitar aquest immens volum de dades i obtenir-ne informació valuosa, s'ha fet imprescindible poder gestionar fluxos de dades (*streams*) en temps real.

Un *stream* o flux de dades és una seqüència contínua d'esdeveniments que es generen al llarg del temps. A diferència del processament per lots, on es treballa amb conjunts de dades fixes, el processament de fluxos permet analitzar i actuar sobre les dades a mesura que arriben. Aquest enfocament és útil en molts casos d'ús, com per exemple, la detecció de frauds, l'anàlisi del comportament d'usuaris en xarxes socials o el monitoratge de sensors, entre molts d'altres.

Dins de l'ecosistema Hadoop podem trobar **Sqoop** i **Flume**, dues eines que tradicionalment s'han emprat per a traslladar grans volums de dades cap a HDFS, de manera que després puguin ser utilitzades per altres eines de l'ecosistema com Hive o Impala. En concret, Sqoop permet importar i exportar dades estructurades entre bases de dades relacionals i HDFS. D'altra banda, Flume permet dur a terme una ingestió en HDFS de fluxos de dades en temps real. En les edicions anteriors d'aquest curs vàrem dedicar un lliurament complet a Sqoop i Flume. No obstant això, aquestes eines han quedat abandonades, davant la necessitat de processar fluxos de dades en temps real i la creixent popularitat de solucions més flexibles i escalables.

L'eina que ha guanyat més rellevància en aquest context és **Apache Kafka**. Kafka no només permet moure grans volums de dades de manera eficient (substituint les funcionalitats que oferia Sqoop), sinó que, més important, ofereix funcionalitats per gestionar aquestes dades en temps real (substituint Flume). A diferència de Flume i Sqoop, Kafka actua com una plataforma de transmissió d'esdeveniments distribuïda, proporcionant garanties de durabilitat, escalabilitat i baixa latència.

En aquest lliurament, veurem en detall com funciona Kafka i com podem treballar amb fluxos de dades en temps real provinents de fonts diverses.

2. Què és Apache Kafka

Començarem veient la definició que podem trobar a la web d'Apache Kafka.



Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

Apache Kafka és una plataforma d'emissió d'esdeveniments distribuïda de codi obert emprada per milers de companyies per a *pipelines* de dades d'alt rendiment, analítica de fluxos, integració de dades i aplicacions de missió crítica.

DEFINICIÓ

Font: <https://kafka.apache.org/>

Kafka és una plataforma distribuïda per gestionar fluxos de dades (*streaming*) que segueix un **model basat en esdeveniment** (*events* en anglès), on les dades es transmeten mitjançant missatges estructurats en un sistema de publicació i subscripció (*publish-subscribe*). És a dir, Kafka ens permet llegir, escriure, emmagatzemar i processar esdeveniments a través de moltes màquines.

Un **esdeveniment** en Kafka és una unitat bàsica de dades. Pot representar qualsevol acció, com per exemple un usuari que fa clic en una pàgina web, una transacció bancària, una lectura d'un sensor IoT, un post en una xarxa social o una actualització de geolocalització d'un telèfon mòbil. Cada esdeveniment té una clau, un valor i una marca de temps. Sovint parlem de missatges com a sinònim d'esdeveniment.

Aquests esdeveniments s'organitzen i emmagatzemen en temes, **topics** en anglès. Un *topic* és una seqüència ordenada d'esdeveniments (o missatges). Fent un paral·lisme, un tema és similar a una carpeta d'un sistema de fitxers, i els esdeveniments són els fitxers d'aquesta carpeta.

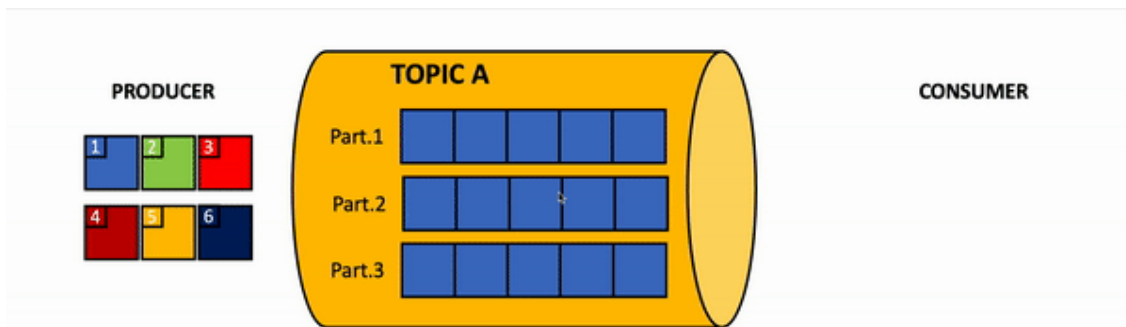
Els **producers** són les aplicacions que envien esdeveniments a Kafka. Poden publicar dades (esdeveniments) a un o diversos tòpics, i ho fan de manera asíncrona per garantir un alt rendiment.

Els **consumidors** són les aplicacions que llegeixen els esdeveniments d'un tòpic. Kafka permet diferents estratègies de consum, incloent-hi consumidors independents i grups de consumidors (*consumer groups*) per processar dades en paral·lel.

Finalment, un **broker** és un servidor de Kafka que rep i emmagatzema esdeveniments, gestionant les particions dels tòpics. Un clúster de Kafka està format per diversos *brokers* que treballen conjuntament per distribuir la càrrega i garantir la disponibilitat. Els *topics* es particionen i repliquen entre els diferents *brokers* del clúster. Així doncs, com que un *topic* pot estar separat en diverses **particions** sobre múltiples brokers, podem tenir múltiples consumidors llegint d'un *topic* en paral·lel. Això proporciona un gran rendiment.

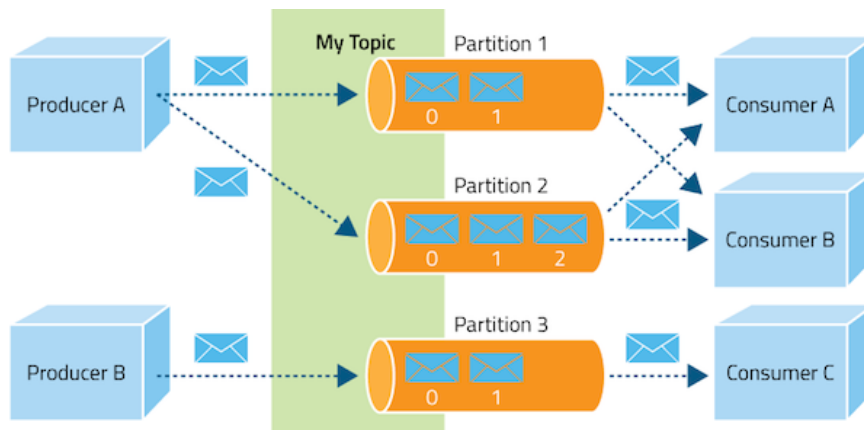
Per simplificar, en aquest curs només farem feina amb brokers independents, sense agrupar-se en clústers. Però hem de tenir clar que Kafka és una solució distribuïda, que garanteix durabilitat, eficiència i tolerància a fallades.

La següent figura mostra el funcionament de Kafka i tots aquests conceptes: el productor va generant esdeveniments (missatges) i els va col·locant en un topic anomenat A. En aquest cas, el topic està particionat en 3 particions. Finalment, el consumidor va llegint el *topic* A per extreure els missatges.



Imatge: Funcionament de Kafka. Autor: @Vladimir topelav

Una visió més completa és la següent, on podem veure que hi participen diversos productors i consumidors generant i llegint missatges en paral·lel:



Imatge: Funcionament de Kafka amb diversos productors i consumidors. Font: Cloudera

Perquè els consumidors puguin fer un seguiment dels missatges en les diferents particions d'un *topic* en particular, Kafka utilitza els *offsets*. L'**offset** és un número seqüencial que identifica de manera única cada missatge dins d'una partició d'un tòpic. D'aquesta manera, els consumidors poden saber quins missatges han llegit i per on continuar. Kafka no esborra immediatament els missatges després de ser consumits, així que un consumidor pot tornar a llegir un missatge a partir del seu *offset*. Cada partició té el seu propi conjunt d'*offsets*, que són immutables i ordenats cronològicament. Això permet implementar mecanismes de reprocessament o recuperació de fallades si un consumidor ha de tornar a llegir dades.

3. Instal·lació i primeres passes amb Kafka

En aquests apunts farem feina amb una màquina amb Ubuntu Desktop. Necessitam tenir-hi Java instal·lat, millor la darrera versió de l'OpenJDK, actualment la 21 (almenys hauríem de tenir una versió igual o posterior a la 17):

```
sudo apt update -y && sudo apt upgrade -y  
sudo apt install openjdk-21-jdk -y
```

Comprovem que ha quedat ben instal·lat amb:

```
java -version
```

Una vegada instal·lat Java, començam descarregant els binaris de Kafka (amb Scala 2.13) des de <https://kafka.apache.org/downloads>. Per un problema de compatibilitat amb un connector que emprarem més endavant, no treballarem amb la darrera versió (3.9.0 actualment) sinó just amb l'anterior (3.8.1). Aquest és l'enllaç directe de la descàrrega:

https://dlcdn.apache.org/kafka/3.8.1/kafka_2.13-3.8.1.tgz

Descomprimim el fitxer tgz i accedim al directori que es crea:

```
tar -xzf kafka_2.13-3.8.1.tgz  
cd kafka_2.13-3.8.1
```

Tenim dues maneres de treballar amb Kafka. La manera tradicional, és emprant Apache ZooKeeper, un servei de coordinació distribuïda que Kafka utilitza per gestionar informació crítica sobre l'estat del seu clúster. La segona és emprant el que s'anomena KRaft (Kafka Raft), un protocol que permet eliminar la dependència de ZooKeeper. En aquest cas, es fan servir uns controladors de quòrum per gestionar les metadades del clúster Kafka. Tot i que KRaft simplifica l'arquitectura de Kafka i fa que sigui més eficient, la configuració i gestió és més complexa. Així que nosaltres treballarem amb Kafka sobre ZooKeeper.

Per posar en marxa el nostre servidor de Kafka, primer hem d'arrancar ZooKeeper:

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

I ara, en un altre terminal, ja podem posar en marxa el *broker* de Kafka:

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

Recordem que Kafka funciona de manera que els productors aniran deixant esdeveniments en els *topics* corresponents i els consumidors van llegint els esdeveniments dels *topics* que els interessin.

Començarem, en un altre terminal, creant un *topic*, al qual li direm *tema-prova*. Hem d'especificar també sobre quin broker de Kafka s'ha de crear el *topic*, en el nostre cas a localhost:9092:

```
bin/kafka-topics.sh --create --topic tema-prova --bootstrap-server localhost:9092
```



En qualsevol moment podem veure el llistat de *topics* del *broker* amb:

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

Podem obtenir una ajuda de totes les opcions amb *topics* amb:

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

Per exemple, per esborrar un topic determinat:

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

Ara que ja està creat el *topic*, podem tenir productors i consumidors que es comuniquin amb el broker Kafka per escriure-hi o llegir-hi esdeveniments, respectivament. Per veure com funciona Kafka, anam a emprar dos clients que s'incorporen en la instal·lació de Kafka, ambdós en el directori bin. Un és un productor, *kafka-console-producer.sh*, i l'altre un consumidor, *kafka-console-consumer.sh*. Executarem cadascun en un terminal diferent.

En el terminal del productor, executam:

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

En el terminal del consumidor, executam:

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

Ara ja tenim els dos processos en marxa. Podem escriure en el terminal del productor. Cada línia que hi escrivim (cada vegada que pitgem <intro>), el productor genera un esdeveniment amb el text que hem escrit, que s'insereix en el *topic* tema-prova del nostre broker. Simultàniament, podem veure en el terminal del consumidor que, cada vegada que el broker rep un nou esdeveniment en el *topic* tema-prova, el consumidor el llegeix i el mostra per pantalla.



Imatge: Clients productor (esquerra) i consumidor (dreta)

Finalment, podem aturar els clients amb Ctrl-C en qualsevol moment. Deixarem, però, Zookeeper i el broker Kafka en marxa per als següents apartats.

4. Produir i consumir esdeveniments amb Python

En l'anterior apartat hem emprat els clients de productor i consumidor per a escriure i llegir esdeveniments en un topic. Ara farem el mateix, des de codi Python, emprant la llibreria [kafka-python](#).

Necessitam tenir instal·lat python i pip en la nostra màquina. Ubuntu Desktop ja porta instal·lat Python 3, podem comprovar la versió:

```
python3 --version
```

Per defecte, pip no està instal·lat. Per instal·lar-ho, podem emprar el paquet d'apt:

```
sudo apt install python3-pip
```

Podem ja instal·lar la llibreria kafka-python per treballar amb Kafka des de Python:

```
pip install kafka-python
```

En aquesta llibreria podem trobar les classes `KafkaProducer` i `KafkaConsumer`, respectivament per definir productors i consumidors de Kafka.

Comencem escrivint un consumidor (`consumer.py`) per al nostre *topic* `tema-prova`, emprant la classe `KafkaConsumer`:

```
from kafka import KafkaConsumer

consumer = KafkaConsumer('tema-prova',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest', # Comença a llegir des del primer missatge
    value_deserializer=lambda x: x.decode('utf-8')
)

for message in consumer:
    print("Missatge rebut: ", message.value)
```

Ho executam:

```
python3 consumer.py
```

I podem veure que, primer de tot, rep els esdeveniments que havíem escrit anteriorment en el *topic* `tema-prova`.

Anam ara a escriure un productor (`producer.py`), que generi un nou esdeveniment en el *topic* `tema-prova`, emprant la classe `KafkaProducer`:

```
from kafka import KafkaProducer

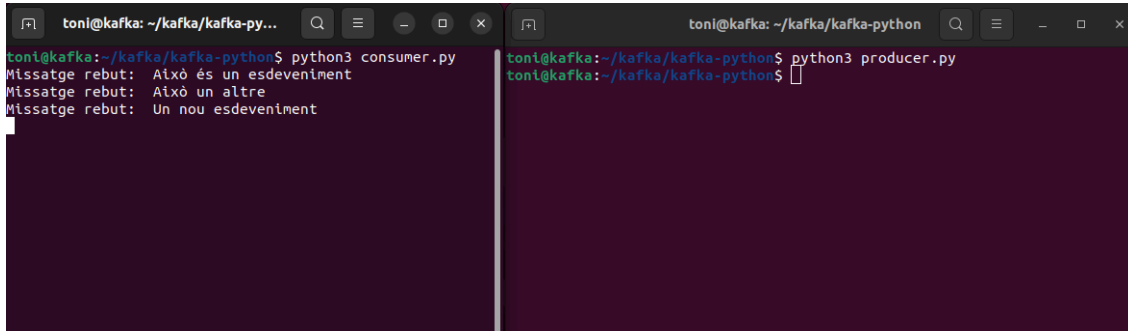
producer = KafkaProducer(
    bootstrap_servers=["localhost:9092"]
)

producer.send("tema-prova", value="Un nou esdeveniment".encode("utf-8")) # Envia
l'esdeveniment al buffer del broker
producer.flush() # Garanteix que el missatge s'envii immediatament
```


Ara executam el productor en un altre terminal:

```
python3 producer.py
```

Podem veure com el consumidor rep el nou missatge o esdeveniment que ha generat el productor, amb el text "Un nou esdeveniment".



```
toni@kafka: ~/kafka/kafka-py...  
toni@kafka:~/kafka/kafka-python$ python3 consumer.py  
Missatge rebut: Això és un esdeveniment  
Missatge rebut: Això un altre  
Missatge rebut: Un nou esdeveniment  
  
toni@kafka: ~/kafka/kafka-python  
toni@kafka:~/kafka/kafka-python$ python3 producer.py  
toni@kafka:~/kafka/kafka-python$
```

Imatge: Productor (dreta) i consumidor (esquerra) amb kafka-python

Si ens fixam en el codi del productor, veim que, una vegada hem creat un objecte `KafkaProducer`, hem invocat dos mètodes, `send()` i `flush()`. Vegem amb una mica més de detall com funcionen aquests mètodes.

El mètode `send()` envia un missatge a un *topic* de Kafka de manera asíncrona. Quan cridam `send()`, el missatge es posa en una cua interna del productor i Kafka l'envia quan considera òptim fer-ho (segons configuracions com la mida del buffer o el temps d'espera). Això fa que l'enviament sigui més eficient, però no immediat. D'altra banda, el mètode `flush()` força l'enviament immediat de tots els missatges en cua al broker de Kafka.

4.1. Emprant esdeveniments JSON

En l'exemple següent, farem que el productor envii 10 esdeveniments. Haurem de fer 10 crides a *send()* i una a *flush()* al final. A més, en lloc de treballar amb missatges en text pla codificat en UTF-8, volem enviar-los en JSON. Per fer-ho, emprarem la propietat *value_serializer* de la classe *KafkaProducer*, que ens permet especificar com es transformen les dades abans d'enviar-les a Kafka. En el nostre cas, definim una funció lambda que converteix automàticament els missatges Python (com diccionaris) en JSON codificat en UTF-8 abans d'enviar-los. I ho farem sobre un topic nou, anomenat *tema-json*:

```
bin/kafka-topics.sh --create --topic tema-json --bootstrap-server localhost:9092
```

Aquest és el codi del productor:

```
import json
from kafka import KafkaProducer

producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8') # Serialitza els missatges
com JSON
)

for i in range(10):
    message = {"id": i, "text": f"Missatge número {i}"}
    producer.send('tema-json', value=message) # Envia el missatge al topic tema-json
    print(f"Enviat: {message}")

producer.flush() # Assegura que tots els missatges s'enviïn abans de tancar
producer.close() # Tanca el recurs i, si queda algun missatge per enviar, l'envia
```

També hem de modificar el consumidor, especificant un *value_deserializer* per deserialitzar l'objecte JSON que arriba:

```
import json
from kafka import KafkaConsumer

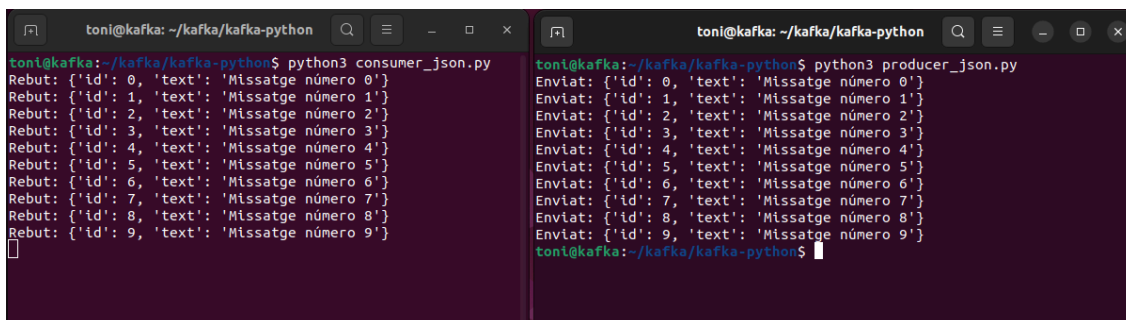
consumer = KafkaConsumer(
    'tema-json',
    bootstrap_servers='localhost:9092',
    auto_offset_reset='earliest', # Comença a llegir des del primer missatge
    value_deserializer=lambda v: json.loads(v.decode('utf-8')) # Deserialitza JSON
)

for message in consumer:
    print(f"Rebut: {message.value}")
```

Si feim un *print* d'algun dels objectes *message* que arriben al *consumer*, una vegada aplicada directament la funció especificada en *value_deserializer*, podem veure la seva estructura:

```
ConsumerRecord(topic='tema-json',
partition=0,
offset=0,
timestamp=1739898784487,
timestamp_type=0,
key=None,
value={'id': 0, 'text': 'Missatge número 0'},
headers=[],
checksum=None,
serialized_key_size=-1,
serialized_value_size=43,
serialized_header_size=-1
)
```

En concret, en el codi del nostre consumidor, per imprimir els missatges que arriben, hem emprat la propietat *value*, en aquest cas `{'id': 0, 'text': 'Missatge número 0'}`



The image shows two terminal windows side-by-side. The left window, titled 'toni@kafka: ~/kafka/kafka-python', shows the output of running 'python3 consumer_json.py'. It displays a series of messages received by the consumer, each as a JSON object: {'id': 0, 'text': 'Missatge número 0'}, {'id': 1, 'text': 'Missatge número 1'}, up to {'id': 9, 'text': 'Missatge número 9'}. The right window, also titled 'toni@kafka: ~/kafka/kafka-python', shows the output of running 'python3 producer_json.py'. It displays a series of messages sent by the producer, each as a JSON object: 'Enviat: {'id': 0, 'text': 'Missatge número 0'}', 'Enviat: {'id': 1, 'text': 'Missatge número 1'}', up to 'Enviat: {'id': 9, 'text': 'Missatge número 9'}'.

Imatge: Productor (dreta) i consumidor (esquerra) amb kafka-python amb missatges JSON

4.2. Grups de consumidors

No hem d'oblidar que Kafka és una solució distribuïda pensada per treballar amb grans volums de dades. Aquí estam fent feina només amb un únic broker Kafka i un productor i un consumidor. Però en un entorn real, l'escenari creix, amb diversos brokers en un clúster, amb diversos productors que van escrivint a diferents *topics* i diversos consumidors que van llegint-los.

A continuació volem afegir un altre consumidor al nostre escenari, de manera que Kafka vagi repartint els missatges (esdeveniments) que arriben al *topic* de manera equilibrada (fent un balanceig de càrrega) entre els dos consumidors. És a dir, cada missatge serà processat per només un dels dos consumidors.

Per aconseguir que cada missatge sigui processat per un únic consumidor, hem de definir un *group_id*, i els dos consumidors n'han de tenir el mateix. Vegem a continuació el codi del productor, que generarà 100 esdeveniments en el *topic* tema-grups (introduint un retard d'un segon entre cada missatge), i dels dos consumidors, que tenen el mateix *group_id* al qual hem anomenat 'grup-consumidors'.

Productor:

```
from kafka import KafkaProducer
import json
import time

producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

for i in range(100):
    message = {"id": i, "text": f"Missatge {i}"}
    producer.send('tema-grups', value=message)
    print(f"Enviat: {message}")
    time.sleep(1) # Simulam un retard entre missatges

producer.flush()
producer.close()
```

Consumidor 1:

```
from kafka import KafkaConsumer
import json

consumer = KafkaConsumer(
    'tema-grups',
    bootstrap_servers=['localhost:9092'],
    group_id='grup-consumidors', # Mateix grup per compartir els missatges
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    value_deserializer=lambda v: json.loads(v.decode('utf-8'))
)

print("Consumidor 1 esperant missatges...")
for message in consumer:
    print(f"Consumidor 1 ha rebut: {message.value}")
```

Consumidor 2:

```
from kafka import KafkaConsumer
import json

consumer = KafkaConsumer(
    'tema-grups',
    bootstrap_servers=['localhost:9092'],
    group_id='grup-consumidors', # Mateix grup per compartir els missatges
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    value_deserializer=lambda v: json.loads(v.decode('utf-8'))
)

print("Consumidor 2 esperant missatges...")
for message in consumer:
    print(f"Consumidor 2 ha rebut: {message.value}")
```

Com tots dos consumidors tenen el mateix *group_id*, Kafka reparteix els missatges. Cada missatge el processa un únic consumidor, però a priori no sabem quin. Kafka intenta equilibrar el volum entre els consumidors disponibles.

Què passa si només hi ha un consumidor?

Si només un consumidor està en marxa, ell rep tots els missatges. Quan l'altre s'afegeix, Kafka comença a repartir els nous missatges.

Què passa si tenen *group_id* diferents?

Si tenen *group_id* diferents (o si no l'hem especificat), tots dos reben tots els missatges, ja que cada grup té la seva pròpia cua de lectura.

enable_auto_commit

Val la pena també mencionar el funcionament de la propietat *enable_auto_commit* que en l'exemple hem establert a *True* (de fet, és el seu valor per defecte).

Aquesta propietat indica si Kafka ha de confirmar automàticament (*commit*) l'offset de cada un dels missatges que el consumidor ha llegit.

Si tenim el valor a *True*, quan un missatge és llegit, Kafka marca automàticament l'offset com a processat. Així, si el consumidor es reinicia, no tornarà a llegir missatges ja processats, perquè Kafka sap fins on ha llegit. Però podria passar que en el processament posterior amb les dades que s'han llegit es produís un error. En aquest cas, com que el missatge ja s'ha marcat com a processat, el consumidor no el podria recuperar. Per evitar això, en aplicacions crítiques on no es poden perdre dades en cas d'errors de processament, podem establir el valor d'*enable_auto_commit* a *False*. En aquest cas, el consumidor haurà de fer el *commit* manualment mitjançant *consumer.commit()*. Això permet garantir que un missatge només es marqui com a processat quan realment ha estat gestionat correctament.

Seria l'equivalent a una transacció d'una base de dades que involucra diverses sentències i feim el *commit* només quan s'han executat correctament totes. Aquí, en Kafka, inclouríem dins la "transacció", no només la lectura del missatge sinó també el seu processament posterior.

5. Kafka Connect

Fins ara, hem vist com Apache Kafka permet gestionar fluxos de dades en temps real a través de *topics*. I hem après a produir i consumir esdeveniments amb els clients de consola que incorpora Kafka (`kafka-console-producer.sh` i `kafka-console-consumer.sh`), així com a fer-ho amb Python emprant la llibreria `kafka-python`. També hem introduït com gestionar offsets i controlar el consum d'esdeveniments.

Això ens permet moure dades dins de Kafka, però el que realment volem és integrar Kafka amb altres sistemes de dades. Moltes aplicacions necessiten enviar o rebre dades de bases de dades, sistemes de fitxers, serveis en el núvol, data warehouses, serveis REST, ...

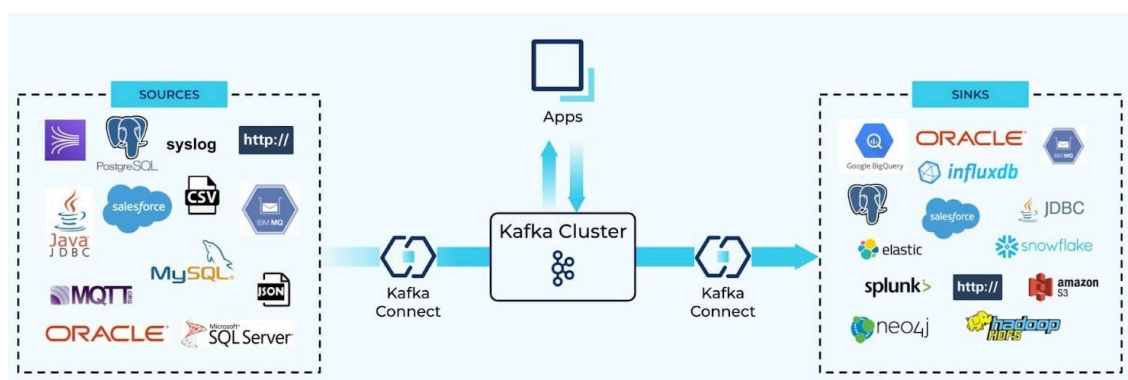
Suposem que volem processar els missatges sobre un determinat hashtag de Twitter per a carregar-los en temps real en Hive i poder fer una anàlisi temporal. Hauríem de seguir aquestes passes:

- Escriure un programa que, emprant l'API de Twitter, vagi obtenint els missatges que ens interessin.
- Escriure un productor Kafka que vagi generant un esdeveniment de Kafka per a cada un d'aquests missatges de Twitter, i l'insereixi en un *topic* determinat.
- Escriure un consumidor Kafka que llegeixi aquests esdeveniments del *topic* i recuperi les dades del missatge
- Escriure un altre programa, emprant alguna API que permeti interactuar amb HDFS, que escrigui les dades del missatge en un directori de HDFS, on podrà ser llegit per Hive

És aquí on entra **Kafka Connect**, un *framework* que facilita la integració de Kafka amb altres sistemes sense necessitat d'escriure codi personalitzat. Podríem dir que és com un "pont" entre Kafka i altres tecnologies, permetent connectar bases de dades, HDFS i altres sistemes de fitxers del núvol, serveis REST, eines d'anàlisi, etc. amb Kafka d'una manera senzilla.

Kafka Connect utilitza connectors ja desenvolupats per interactuar amb altres sistemes. Hi ha dos tipus principals:

- **Source Connectors** (connectors font): llegeixen dades d'una font externa (com una base de dades, API, fitxers...) i les envien a un *topic* de Kafka. Per exemple, llegir dades de MySQL i enviar-les a Kafka.
- **Sink Connectors** (connectors embornals): llegeixen dades d'un *topic* de Kafka i les escriuen en un sistema de destinació (HDFS, PostgreSQL, Elasticsearch...). Per exemple, escriure les dades de Kafka en HDFS per fer anàlisi posterior amb Hive.



Imatge: Visió general de Kafka Connect. Font: @TheDanicaFine / developer.confluent.io

Existeixen multitud de connectors disponibles, de manera que si empram Kafka Connect i algun d'aquests connectors, només hem de configurar-lo des d'un arxiu de propietats, no necessitam programar manualment els productors i/o consumidors. Tornant a l'exemple de Twitter i Hive, només hauríem de:

- Instal·lar el connector de Twitter i modificar algunes propietats del seu arxiu de configuració
- Instal·lar el connector de HDFS i modificar algunes propietats del seu arxiu de configuració

Molt més senzill!

Així doncs, per què és útil Kafka Connect?

- Evita haver de programar productors i consumidors personalitzats.
- Simplifica la gestió de dades en temps real amb connectors ja existents.
- Escala fàcilment per gestionar grans volums de dades.
- A més, permet fer transformacions d'esquema abans d'enviar les dades.








Desafortunadament, molts d'aquests connectors no són de codi obert i només es poden emprar en plataformes comercials de Kafka i Kafka Connect, com la de Confluent.

Confluent

Kafka és un projecte de codi obert desenvolupat originalment per LinkedIn i després donat a la Apache Software Foundation. Els creadors de Kafka a LinkedIn (Jay Kreps, Neha Narkhede i Jun Rao) varen fundar [Confluent](#) amb l'objectiu de proporcionar una plataforma empresarial sobre Kafka, amb eines addicionals i suport comercial. Així doncs, Kafka és el nucli de la tecnologia, però Confluent ofereix una versió millorada amb funcionalitats extra per facilitar la gestió i escalabilitat. En concret, Confluent ofereix una gran varietat de connectors "premium". Es pot trobar una informació més detallada sobre les diferències entre Kafka i Confluent a la [web de Confluent](#).

Confluent ofereix dues opcions: Confluent Platform, per a instal·lacions *on premise*, i Confluent Cloud, una solució nativa de núvol.

La taula següent mostra un resum amb les principals diferències entre Kafka i Confluent:

Característica	Apache Kafka (Open Source)	Confluent Platform (Comercial)
 Codi	Open Source (Apache 2.0)	Codi obert + propietari (algunes parts privades)
 Facilitat d'ús	Instal·lació manual	Instal·lació més senzilla amb eines addicionals
 Connectors de Kafka Connect	Només connectors Open Source	Inclou connectors propietaris (JDBC avançat, HTTP, Salesforce, etc.)
 Monitoratge i gestió	Administració manual amb scripts	Control avançat amb Confluent Control Center
 Eines extres	Kafka Streams, Kafka Connect	Confluent Schema Registry, Control Center, Confluent REST Proxy, ksqlDB
 Desplegament al núvol	Hauries de configurar-ho a mà	Confluent Cloud (Kafka gestionat)
 Suport	Comunitat d'usuaris	Suport empresarial de Confluent

En aquest curs no emprem Confluent, farem feina amb Kafka "pur" i només connectors de codi obert que puguin ser utilitzats en la versió oberta de Kafka. Això ens limita bastant, però ens donarà una idea clara de l'eina i també del que poden aportar les plataformes comercials.

5.1. Workers i tasques

En Kafka Connect, un **worker** és el procés encarregat d'executar els connectors i gestionar el flux de dades. Directament relacionat amb els workers, tenim dues maneres d'executar Kafka Connect:

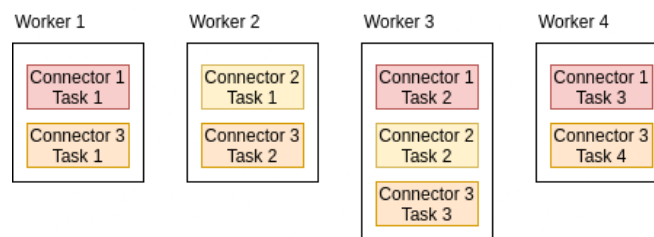
- En **mode standalone**, on tenim un únic *worker* que gestiona tots els connectors. El mode *standalone* només es fa servir en entorns de proves.
- En **mode distribuït**, on hi h podem tenir diversos *workers* connectats al mateix clúster de Kafka.

En un *worker* podem executar diversos connectors. Cada un d'ells es crea a partir d'un fitxer de configuració on s'especifiquen detalls com ara quina taula s'ha de llegir en un connector *source* per a MySQL o en quin *path* s'ha d'escriure en un connector *sink* per a HDFS.

A més, un connector pot dividir la càrrega de treball en diverses **tasques** (*tasks*), que treballen en paral·lel. Una de les propietats que s'especifiquen en el fitxer de configuració del connector és el nombre màxim de tasques amb què pot treballar. Aquestes tasques, quan fem feina en mode distribuït, es poden repartir entre diferents *workers*.

Imaginem que un connector *source* llegeix dades d'una base de dades i genera missatges a un *topic* Kafka. Si configuram el connector per utilitzar 4 tasques, aquestes poden executar-se en fins a 4 *workers* diferents. Això permet escalar la càrrega i fer el procés més eficient.

Resumint, mentre que en el mode *standalone*, totes les tasques (de tots els connectors) s'executen en el mateix *worker*, en el mode distribuït, podem distribuir les tasques en diferents *workers*. La imatge en mostra un exemple:



Imatge: Workers en Kafka Connect en mode distribuït.

Font: <https://daniel.arneam.com/blog/distributedarchitecture/2020-10-15-Kafka-Connect-Concepts/>

En l'apartat següent també veurem que la manera de configurar els connectors en mode *standalone* i en mode distribuït es fa de manera diferent.

5.2. Cas pràctic 1: de fitxer a fitxer

En aquest apartat anam a desenvolupar un primer cas pràctic amb Kafka Connect, on importarem en temps real dades des d'un fitxer a un *topic* Kafka i exportarem les dades des del *topic* a un altre fitxer. Això podria ser útil per a un sistema que vagi llegint dades des d'un fitxer de log: cada línia afegida en el fitxer de log generarà un nou esdeveniment en el *topic*.

Kafka Connect ja està inclòs en Kafka, no necessitam instal·lar res més. En Kafka Connect, cada connector és una classe Java, que es distribueix dins arxius *.jar* (que poden integrar diversos connectors). Per implementar aquest exemple emprarem dos connectors que venen amb la instal·lació de Kafka (en l'arxiu *lib/connect-file-3.8.1.jar*), un per al *source* (*FileStreamSource*) i un per al *sink* (*FileStreamSink*).

Necessitarem dos fitxers de propietats, un per a cada connector. En el directori *config* de la nostra instal·lació de Kafka ja tenim dos fitxers que podem emprar: *connect-file-source.properties* i *connect-file-sink.properties*. En el nostre cas, només hem de modificar les següents propietats:

- El *topic* que emprarem (en el *source* i en el *sink*), al qual li direm *connect-fitxers*.
- El nom del fitxer des d'on volem llegir (en el *source*), al qual li direm *entrada.txt*
- El nom del fitxer on volem escriure (en el *sink*), al qual li direm *sortida.txt*

Aquest serà el contingut dels dos fitxers de propietats:

config/connect-file-source.properties:

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=entrada.txt
topic=connect-fitxers
```

config/connect-file-sink.properties:

```
name=local-file-sink
connector.class=FileStreamSink
tasks.max=1
file=sortida.txt
topics=connect-fitxers
```

Execució en mode *standalone*

Anam a veure primer com executar el nostre exemple fent feina en el mode *standalone*, és a dir, amb un únic worker, que és la manera més senzilla.

Haurem d'editar un tercer fitxer de propietats, *config/connect-standalone.properties*, que ens permet configurar el *worker*. Només hem d'afegir la següent línia al final, indicant on pot trobar les classes dels nostres connectors, en el nostre cas l'arxiu *connect-file-3.8.1.jar*:

```
plugin.path=libs/connect-file-3.8.1.jar
```

Ara podem crear el fitxer *entrada.txt*, en el directori arrel de Kafka:

```
touch entrada.txt
```

I, seguint en el directori arrel de Kafka, suposant que ja tenim ZooKeeper i el broker de Kafka en marxa, ja podem arrancar el *worker* de Kafka Connect en mode *standalone*: amb el script *bin/connect-standalone.sh*:

```
bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-
source.properties config/connect-file-sink.properties
```

Per provar-ho, en un altre terminal, des del directori arrel de Kafka, podem anar escrivint en el fitxer entrada.txt. Per exemple:

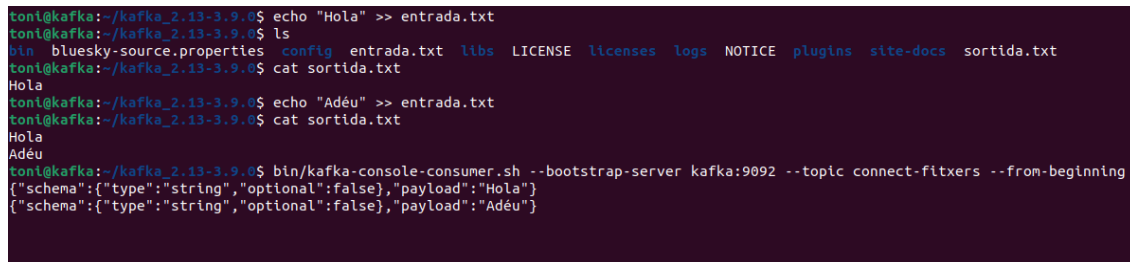
```
echo "Hola" >> entrada.txt
echo "Adéu" >> entrada.txt
```

Podem comprovar que s'ha creat el fitxer sortida.txt i que conté

```
Hola
Adeu
```

Podem seguir provant d'escriure en entrada.txt i veurem com es va copiant a sortida.txt. Si volem, també podem comprovar els esdeveniments que s'han inserit en el *topic* connect-fitxers:

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



```
toni@kafka:~/kafka_2.13-3.9.0$ echo "Hola" >> entrada.txt
toni@kafka:~/kafka_2.13-3.9.0$ ls
bin bluesky-source.properties config entrada.txt libs LICENSE licenses logs NOTICE plugins site-docs sortida.txt
toni@kafka:~/kafka_2.13-3.9.0$ cat sortida.txt
Hola
toni@kafka:~/kafka_2.13-3.9.0$ echo "Adéu" >> entrada.txt
toni@kafka:~/kafka_2.13-3.9.0$ cat sortida.txt
Hola
Adéu
toni@kafka:~/kafka_2.13-3.9.0$ bin/kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic connect-fitxers --from-beginning
{"schema":{"type":"string","optional":false},"payload":"Hola"}
{"schema":{"type":"string","optional":false},"payload":"Adéu"}
```

Imatge: Execució de l'exemple

Execució en mode distribuït

Quan treballam en mode distribuït, en primer lloc, hem de configurar el fitxer config/connect-distributed.properties, on especifiquem les propietats generals del *worker*. En concret, igual que amb el mode standalone, aquí hem de donar-li valor a la propietat plugins, amb el *path* on tenim els connectors:

```
plugin.path=libs/connect-file-3.8.1.jar
```

A continuació, arrancam el worker de Kafka Connect amb el script bin/connect-distributed.sh (prèviament hem hagut d'arrancar ZooKeeper i el broker de Kafka):

```
bin/connect-distributed.sh config/connect-distributed.properties
```

Això, a més, crea un endpoint al port 8083 (*http://localhost:8083* en el nostre cas), amb una API REST amb la qual podem interactuar per a gestionar els connectors. Ho veurem amb més detall una després.

D'altra banda, quan treballam en mode distribuït, els fitxers de configuració dels connectors s'han d'especificar en format JSON. En la propietat "file" hem d'especificar el path absolut on tenim els fitxers d'entrada (en el *source*) i sortida (en el *sink*). En els dos casos, treballam amb el topic "connect-fitxers". Aquests seran els fitxers de configuració:

```
config/connect-file-source.json
```

```
{
  "name": "file-source-distributed",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": "1",
    "file": "/home/toni/kafka_2.13-3.8.1/entrada.txt",
    "topic": "connect-fitxers"
  }
}
```

config/connect-file-sink.json

```
{
  "name": "file-sink-distributed",
  "config": {
    "connector.class": "FileStreamSink",
    "tasks.max": "1",
    "file": "/home/toni/kafka_2.13-3.8.1/sortida.txt",
    "topics": "connect-fitxers"
  }
}
```

Anam ara ja a configurar el nostre connector per al *source* en el *worker*, emprant l'API REST, executant la següent ordre:

```
curl -X POST -H "Content-Type: application/json" --data @config/connect-file-source.json
http://localhost:8083/connectors
```

I per a configurar el *sink*:

```
curl -X POST -H "Content-Type: application/json" --data @config/connect-file-sink.json
http://localhost:8083/connectors
```

Podem veure els connectors configurats amb:

```
curl http://kafka:8083/connectors
```

I el llistat de plugins disponibles amb:

```
curl http://kafka:8083/connector-plugins
```



Podeu trobar una descripció detallada de l'API REST de Kafka Connect a la documentació de Kafka: https://kafka.apache.org/documentation.html#connect_rest

Aquesta és la resposta, podem veure que els dos primers són els que acabam de configurar:

```
[{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector","type":"sink","version":"3.8.1"}  
{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector","type":"source","version":"3.8.1"}  
{"class":"org.apache.kafka.connect.mirror.MirrorCheckpointConnector","type":"source","version":"3.8.1"}  
{"class":"org.apache.kafka.connect.mirror.MirrorHeartbeatConnector","type":"source","version":"3.8.1"}  
{"class":"org.apache.kafka.connect.mirror.MirrorSourceConnector","type":"source","version":"3.8.1"}
```

Podem provar d'anar escrivint amb *echo* en el fitxer entrada.txt i comprovar que el que escrivim es va copiant a sortida.txt.

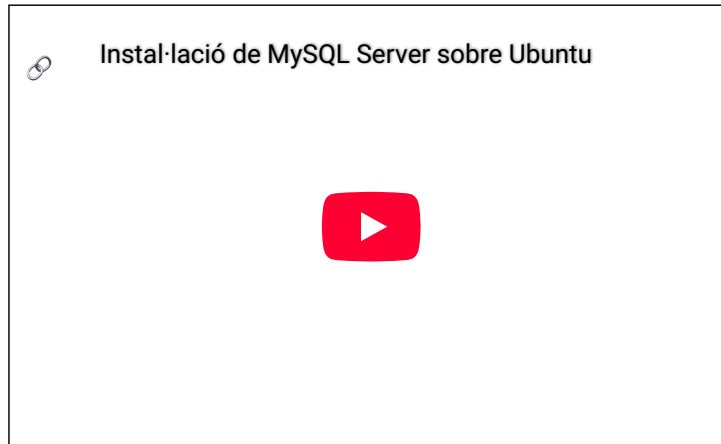
Finalment, si volem eliminar els connectors, ho feim també mitjançant l'API REST, emprant el nom dels connectors:

```
curl -X DELETE http://localhost:8083/connectors/file-source-distributed  
curl -X DELETE http://localhost:8083/connectors/file-sink-distributed
```

5.3. Cas pràctic 2: de MySQL a HDFS (part del source)

En aquest segon cas pràctic amb Kafka Connect, anam a suposar un escenari on tenim una base de dades MySQL amb una taula on un sensor de temperatures va inserint lectures en temps real. Volem carregar aquestes dades en HDFS perquè després puguem utilitzar alguna de les eines que hem vist en el mòdul (Pig, Hive o Impala) per poder analitzar les dades.

Haurem de començar instal·lant MySQL Server en la nostra màquina Ubuntu. Podeu trobar una explicació detallada en aquest vídeo:



Vídeo: Instal·lació de MySQL Server sobre Ubuntu

En tot cas, no cal instal·lar Workbench, que també s'explica en el vídeo.

Una vegada instal·lat MySQL, podem fer la resta de l'exercici amb l'usuari *root*, tot i que estaria bé crear un usuari *kafka*, ja que per seguretat, sempre és convenient no emprar l'usuari *root* per connectar-nos des d'aplicacions.

Crearem una base de dades anomenada *sensors*, i en ella, una taula *temperatures*, amb la següent definició:

```
CREATE TABLE temperatures (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  datahora DATETIME NOT NULL,  
  valor FLOAT NOT NULL  
);
```

Més tard anirem fent inserts en la taula simulant lectures de la temperatura del sensor. De moment, però anam a configurar el nostre connector per llegir les dades de la taula i inserir-les en un *topic* de Kafka.

Començarem creant un directori *plugins* dins *kafka_2.13-3.8.1*, on anirem posant tots els connectors que necessitem. En aquest exemple, per al *source* emprem el connector JDBC de Confluent, que podem trobar a <https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc>. Aquest connector és obert i pot emprar-se amb Kafka Server, no només des de la Plataforma Confluent. El descarregam (l'opció Self-Hosted) i una vegada descomprimit, copiam la carpeta sencera a *plugins*.

Ara hem de modificar el fitxer *config/connect-distributed.properties* per afegir el directori *plugins* a la propietat *plugin.path* (si volem, ja podem eliminar la referència al connector de fitxers):

```
plugin.path=libs/connect-file-3.8.1.jar,/home/toni/kafka_2.13-3.8.1/plugins
```

També hem de copiar el driver JDBC de MySQL perquè el connector pugui emprar-lo. Descarregam la darrera versió des de <https://dev.mysql.com/downloads/connector/j/> com a "Platform independent" i de l'arxiu comprimit extreim *mysql-connector-j-9.2.0.jar* i el copiam al directori *lib* del connector.

Si no estaven ja en marxa, podem arrancar de nou ZooKeeper i el broker de Kafka:

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

I posarem en marxa el *worker* de Kafka Connect en mode distribuït:

```
bin/connect-distributed.sh config/connect-distributed.properties
```

Ara ja podem configurar el connector *source*. Primer, crearem el fitxer JSON de configuració `config/mysql-source.json`:

```
{
  "name": "mysql-temperatures",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "tasks.max": 1,
    "connection.url": "jdbc:mysql://localhost/sensors",
    "connection.user": "kafka",
    "connection.password": "XXXXXXXX",
    "table": "temperatures",
    "topic.prefix": "mysql_",
    "mode": "incrementing",
    "incrementing.column.name": "id"
  }
}
```

Hem anomenat `mysql-temperatures` al nostre connector, que fa feina amb una única tasca, es connecta mitjançant JDBC a la base de dades `sensors` amb l'usuari `kafka` i contrasenya `XXXXXXXX` per llegir la taula `temperatures`. Cada nova inserció generarà un nou esdeveniment al topic `mysql_temperatures` (prefix + nom de la taula). Per tant, abans de carregar el connector, hem de crear el *topic*:

```
bin/kafka-topics.sh --create --topic mysql_temperatures --bootstrap-server localhost:9092
```

Utilitzam l'API REST de Kafka Connect per configurar el connector JDBC com a *source*:

```
curl -X POST -H "Content-Type: application/json" --data @config/mysql-source.json
http://localhost:8083/connectors
```

Ja tenim el nostre connector en marxa. Podem provar que tot funciona correctament fent una inserció en la taula `temperatures`:

```
INSERT INTO temperatures(datahora, valor) VALUES('2025-02-21 10:00:00', 14.5);
```

Podem comprovar que s'ha generat l'esdeveniment en el *topic* `mysql_temperatures`:

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

I podrem veure com tenim un esdeveniment:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      { "type": "int32", "optional": false, "field": "id" },
      { "type": "int64", "optional": false, "name": "org.apache.kafka.connect.data.Timestamp", "version": 1, "field": "datahora", "logicalName": "timestamp", "isPrimitive": true },
      { "type": "float", "optional": false, "field": "valor" }
    ],
    "optional": false,
    "name": "temperatures"
  },
  "payload": { "id": 1, "datahora": 1740132000000, "valor": 14.5 }
}
```

On primer trobam l'esquema de la taula (objecte *schema*) i després, dins l'objecte *payload*, les dades de la fila que hem inserit: {"id":1,"datahora":1740132000000,"valor":14.5}. La columna *datahora*, de tipus *datetime* es mostra com un enter, en format Unix *timestamp*.

Podem seguir provant fent altres insercions.

5.4. Cas pràctic 2: de MySQL a HDFS (part del sink)

Ara que ja tenim configurat el *source* i les dades es van inserint en el *topic* `mysql_temperatures`, podem configurar el sink per copiar les dades en un directori de HDFS del nostre clúster Hadoop del lliurament 1.

El [Confluent Hub](#) és un repositori amb una gran quantitat de connectors, la majoria elaborats per Confluent (com el que ja hem emprat de JDBC), però també d'altres companyies com Microsoft, MongoDB, Snowflake, etc.

Aquí podem trobar 4 connectors relacionats amb HDFS, tots desenvolupats per Confluent. N'hi ha dos (un de *source* i un de *sink*) per a HDFS 2 i altres dos per a HDFS 3. Mentre que els connectors de HDFS 2 estan disponibles sota llicència comunitària i poden utilitzar-se amb la versió oberta de Kafka, els de HDFS 3 tenen una llicència comercial i només es poden emprar en Confluent Platform i Confluent Cloud, així que no els podem emprar amb el nostre broker Kafka. D'altra banda, com que el nostre clúster del lliurament 1 utilitza Hadoop 3, tampoc no podem emprar els connectors de HDFS 2.

No perdrem el temps instal·lant una versió d'avaluació de Confluent Platform, ni tampoc sol·licitarem accés de prova a Confluent Cloud, per a la qual cosa hem de donar una targeta de crèdit. Tampoc val la pena instal·lar un clúster de Hadoop 2 per fer les proves. Així que, en lloc d'utilitzar un connector de Kafka Connect per escriure en HDFS, anam a desenvolupar un **consumidor emprant kafka-python**.

Abans de fer això, hem de configurar la nostra màquina amb Ubuntu perquè pugui accedir al clúster Hadoop. Això està explicat a l'[apartat 6.7 dels apunts del primer lliurament](#). Depenent de si la màquina Ubuntu és física o virtual, pot ser més senzill redireccionar ports (si és física) o bé configurar un nou adaptador de xarxa amb "Adaptador sólo anfitrión" amb una IP 10.10.10.X i configurar `etc/hosts` per poder veure els nodes del clúster (almenys `hadoopmaster`).

A continuació, és necessari activar WebHDFS al NameNode. WebHDFS és una API REST que permet interactuar amb HDFS mitjançant peticions HTTP, facilitant la lectura, escriptura i gestió de fitxers sense necessitat d'un client Hadoop. Entrem en `hadoopmaster` i editam el fitxer `$HADOOP_HOME/etc/hadoop/hdfs-site.xml` on hem d'afegir:

```
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
<property>
  <name>dfs.namenode.http-address</name>
  <value>0.0.0.0:9870</value>
</property>
```

Podem arrancar ja els serveis de Hadoop amb:

```
start-all.sh
```

I podem comprovar que WebHDFS està ben configurat, executant:

```
hdfs getconf -confKey dfs.webhdfs.enabled
```

Que ens hauria de respondre

```
true
```

També podem accedir des del navegador: <http://hadoopmaster:9870/webhdfs/v1/?op=LISTSTATUS>

Ara cream un directori i un fitxer de text en HDFS i li donarem permisos:


```
hdfs dfs -mkdir /user/hadoop/kafka
hdfs dfs -touchz /user/hadoop/kafka/temperatures.txt
hdfs dfs -chmod -R 777 /user/hadoop/kafka
```

Tornam a la nostra màquina amb Ubuntu i hem d'instal·lar la llibreria hdfs de python, que ens servirà per interactuar amb el nostre clúster Hadoop emprant WebHDFS:

```
pip install hdfs
```

I ja podem escriure el nostre consumidor (consumer_hdfs_temperatures.py):

```
from kafka import KafkaConsumer
from hdfs import InsecureClient

hdfs_client = InsecureClient("http://hadoopmaster:9870", user="hadoop")
hdfs_file = "/user/hadoop/kafka/temperatures.txt"

if not hdfs_client.status(hdfs_file, strict=False):
    with hdfs_client.write(hdfs_file, overwrite=True) as writer:
        writer.write(b"") # Crear un fitxer buit

consumer = KafkaConsumer('mysql_temperatures',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='my_group_id',
    value_deserializer=lambda x: x.decode('utf-8')
)

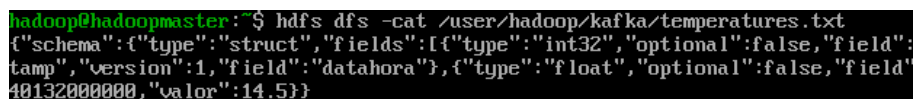
for message in consumer:
    msg = message.value
    print("Missatge rebut: ", msg)

    # Afegir el missatge a HDFS
    with hdfs_client.write(hdfs_file, append=True) as writer:
        writer.write((msg + "\n").encode("utf-8"))
```

Quan ho executem amb:

```
python3 consumer_hdfs_temperatures.py
```

Podem comprovar que s'ha copiat el missatge complet en el fitxer /user/hadoop/kafka/temperatures.txt:



```
hadoop@hadoopmaster:~$ hdfs dfs -cat /user/hadoop/kafka/temperatures.txt
{"schema":{"type":"struct","fields":[{"type":"int32","optional":false,"field":"temp","version":1,"field":"datahora"}, {"type":"float","optional":false,"field":"valor"}]}, "version":1, "field": "datahora"}, {"type": "float", "optional": false, "field": "valor": 14.5}}
```

Imatge: Missatge copiat al fitxer en HDFS

Podem millorar una mica la sortida. Emprarem un deserialitzador per a JSON i extraurem els valors de *datahora* (que aprofitarem per donar-li format de data i hora) i *valor*, de manera que el fitxer ens quedarà amb format CSV:

```

import json
from datetime import datetime
from kafka import KafkaConsumer
from hdfs import InsecureClient

hdfs_client = InsecureClient("http://hadoopmaster:9870", user="hadoop")
hdfs_file = "/user/hadoop/kafka/temperatures.txt"

if not hdfs_client.status(hdfs_file, strict=False):
    with hdfs_client.write(hdfs_file, overwrite=True) as writer:
        writer.write(b'') # Crear un fitxer buit

consumer = KafkaConsumer('mysql_temperatures',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='my_group_id',
    value_deserializer=lambda x: json.loads(x.decode('utf-8')) # Convertir JSON a
diccionari
)

for message in consumer:
    msg = message.value
    print("Missatge rebut: ", msg)

    payload = msg.get("payload", {})

    datahora_ms = payload.get("datahora", 0)
    if datahora_ms:
        datahora = datetime.utcfromtimestamp(datahora_ms / 1000).strftime('%d/%m/%Y
%H:%M:%S')
    else:
        datahora = "UNKNOWN"

    valor = payload.get("valor", 0.0)

    # Afegir només els valors de datahora i valor, separats per comes, a HDFS
    with hdfs_client.write(hdfs_file, append=True) as writer:
        writer.write(f"{datahora},{valor}\n".encode("utf-8"))

```

Aquest és el resultat després d'haver fet algunes insercions:

```

21/02/2025 10:00:00,14.5
21/02/2025 11:00:00,16.7
21/02/2025 12:00:00,17.1

```

Imatge: Files escrites al fitxer en HDFS

5.5. Cas pràctic 3: xarxes socials

Les xarxes socials generen una quantitat massiva de dades en temps real. Publicacions, comentaris, *likes*, comparticions i missatges directes són només alguns exemples d'esdeveniments que es produeixen constantment. Per gestionar aquest volum de dades de manera eficient, moltes plataformes utilitzen Kafka.

Kafka actua com una espina dorsal per a la transmissió d'esdeveniments en aquestes plataformes, garantint un flux de dades fiable i escalable. Cada interacció es pot modelar com un esdeveniment que es publica en un *topic* de Kafka i es distribueix a diferents serveis que el consumeixen. Això permet implementar funcionalitats com la recomanació de contingut, la detecció de tendències i l'anàlisi de l'activitat dels usuaris en temps real.

En Confluent Hub podem trobar un [connector source per a Twitter/X](#). Tot i que aquest connector és de codi obert amb llicència Apache, i es pot desplegar sense problemes en Kafka Server, des de fa un temps l'API de Twitter és de pagament. Fins i tot l'accés bàsic té un cost de 175\$/mes. Per aquesta raó, no podem emprar aquesta eina.

En els darrers mesos, [Bluesky](#) i [Mastodon](#) han experimentat un augment significatiu en popularitat com a alternatives a Twitter. Aquest increment es deu, en part, als canvis implementats per Elon Musk després de l'adquisició de Twitter, que han generat descontentament entre molts usuaris. Bluesky, creada pel cofundador de Twitter Jack Dorsey, ofereix una plataforma descentralitzada que permet als usuaris tenir més control sobre el seu contingut i experiència. Per la seva banda, Mastodon és una xarxa social de codi obert que permet als usuaris crear i gestionar les seves pròpies comunitats, oferint una experiència més personalitzada i lliure de les polítiques de les grans corporacions. Ambdues plataformes han atret usuaris que busquen alternatives més transparents i controlables en comparació amb Twitter.

A més, tant Bluesky com Mastodon ofereixen APIs obertes i gratuïtes. En aquest context, Dale Lane ha desenvolupat dos [connectors de Kafka per a Bluesky i Mastodon](#), de codi obert, disponibles a GitHub. Nosaltres treballarem aquí amb el de Bluesky, que podem descarregar des de <https://github.com/dalelane/kafka-connect-bluesky-source/releases>. Simplement hem de descarregar l'arxiu [kafka-connect-bluesky-source-0.0.1-jar-with-dependencies.jar](#) i copiar-lo al directori de *plugins*.

Abans de configurar-ho, si no en tenim, haurem de tenir un usuari de Bluesky. En el meu cas, el meu usuari és toninavarrete. Una vegada ja la tinguem el compte creat, hem d'anar als "Ajustes" i en "Privacidad y seguridad", entrar en "Contraseñas de app". També hi podem arribar des d'aquesta URL: <https://bsky.app/settings/app-passwords>

Aquí hem de pitjar "Añadir contraseñas de app" per generar una contrasenya que emprem per connectar-nos des del connector Kafka. Ens demana introduir un nom de la contrasenya, podem dir-li "Kafka" i no cal marcar "Permitir el acceso a tus mensajes directos". Ens apareixerà una contrasenya que hem d'anotar i que emprem més tard.

Ara ja podem escriure el nostre fitxer de configuració, per fer feina amb el mode distribuït.

El que volem en aquest exemple és que ens vagi creant esdeveniments cada vegada que algú publiqui algun missatge amb una o unes paraules. En aquest cas, jo he triat "LaLiga". En el fitxer de propietats, hem d'especificar les següents propietats:

- name: el nom del connector, jo li diré bluesky-laliga
- connector.class: "uk.co.dalelane.kafkaconnect.bluesky.source.BlueskySourceConnector" (la classe dins de l'arxiu jar que conté el connector)
- key.converter: "org.apache.kafka.connect.storage.StringConverter"
- value.converter: "org.apache.kafka.connect.json.JsonConverter" (produeix esdeveniments en JSON)
- bluesky.identity: el nostre usuari, seguit de .bsky.social. En el meu cas "toninavarrete.bsky.social"
- bluesky.password: la contrasenya per a apps que hem generat anteriorment
- bluesky.searchterm: recuperarà missatges que continguin les paraules. Jo he emprat LaLiga
- bluesky.topic: com es dirà el *topic* on s'escriuran els esdeveniments. Jo li he posat "bluesky_laliga"
- tasks.max: màxim de tasques, podem deixar en 1

Així doncs, el meu fitxer de configuració (bluesky-source.json) serà aquest:

```
{
  "name": "bluesky-laliga",
  "config": {
    "connector.class":
    "uk.co.dalelane.kafkaconnect.bluesky.source.BlueskySourceConnector",

    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",

    "bluesky.identity": "toninavarrete.bsky.social",
    "bluesky.password": "XXXX-XXXX-XXXX-XXXX",

    "bluesky.searchterm": "LaLiga",
    "bluesky.topic": "bluesky_laliga",

    "tasks.max": 1
  }
}
```

Si no els teníem ja en marxa, arrancam ZooKeeper, el broker de Kafka i el worker de Kafka Connect en mode distribuït:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
bin/connect-distributed.sh config/connect-distributed.properties
```

I configuram el connector a partir del fitxer bluesky-source.json:

```
curl -X POST -H "Content-Type: application/json" --data @config/bluesky-source.json
http://localhost:8083/connectors
```

Podem obrir el client consumidor per veure com es van generant esdeveniments amb:

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

Aquest és un d'aquests esdeveniments:

```
{
  "schema": {
    "type": "struct",
    "fields": {
      "type": "struct",
      "fields": {
        "type": "string",
        "optional": false,
        "field": "uri"
      },
      "type": "string",
      "optional": false,
      "field": "cid"
    },
    "optional": false,
    "field": "id"
  },
  "type": "string",
  "optional": false,
  "field": "text"
},
{
  "type": "array",
  "items": {
    "type": "string",
    "optional": false
  },
  "optional": false,
  "field": "langs"
},
{
  "type": "int64",
  "optional": true,
  "name": "org.apache.kafka.connect.data.Timestamp",
  "version": 1,
  "field": "createdAt"
},
{
  "type": "struct",
  "fields": {
    "type": "string",
    "optional": false,
    "field": "handle"
  },
  "type": "string",
  "optional": true,
  "field": "displayName"
},
{
  "type": "string",
  "optional": true,
  "field": "avatar"
}],
"optional": false,
"field": "author"
}],
"optional": false,
"name": "st
payload": {
  "id": {
    "uri": "at://did:plc:irdwytev7x5ahhjqlnx2dxc/app.bsky.feed.post/3lip7xnohfs2j",
    "cid": "bafyreicjakwawhbixnn7qqwv73bsadi52xwv4ml2ph3bwpowqt6pcfi2y",
    "text": "Convocatoria del Villarreal CF per al partit de LaLiga davant el Rayo
Vallecano\nnoticiesdigitals.com/convocatoria...\n
#convocatoria #villarrealcf #futbol #esports #noticies",
    "langs": ["es"],
    "createdAt": 1740155312026,
    "author": {
      "handle": "noticiesdigitals.bsky.social",
      "displayName": "Notícies Digitals",
      "avatar": "https://cdn.bsky.app/img/avatar/plain/did:plc:irdwytev7x5ahhjqlnx2dxc/bafkreigc2kpr236c"
```

Podem veure que el post parla sobre la convocatòria del Villareal contra el Rayo Vallecano i que ha estat publicat per l'usuari noticiesdigitals.bsky.social. De la uri també podem extreure el id del post: L'id del post: 3lip7xnohfs2j. Així que, a partir d'això, podem obtenir la URL del post: <https://bsky.app/profile/noticiesdigitals.bsky.social/post/3lip7xnohfs2j>

@

🏠

🔍

🔔

💬

#


⋮

👤

⚙️

← Publicar

🔗



Noticies Digitals


@noticiesdigitals.bsky.social

+ Seguir

Convocatoria del Villarreal CF per al partit de LaLiga davant el Rayo Vallecano

noticiesdigitals.com/convocatoria...

#convocatoria #villarrealcf #futbol #esports #noticies



Imatge: Post recuperat de Bluesky

5.6. Transformacions

Quan integrem diverses fonts de dades amb Kafka Connect, sovint ens trobem amb la necessitat de modificar, enriquir o filtrar els missatges abans d'emmagatzemar-los o distribuir-los. Aquí és on entren en joc les **Single Message Transformations (SMTs)**. Aquestes transformacions permeten alterar els missatges en temps real, sense necessitat de modificar el codi dels connectors o dels consumidors.

Les SMTs es poden utilitzar per a tasques com:

- Filtratge de dades: descartar missatges que no compleixen certs criteris.
- Canvi de format: convertir timestamps a un format llegible, eliminar camps innecessaris, o normalitzar valors.
- Enriquiment de dades: afegir informació contextual o valors calculats basats en el contingut del missatge.
- Repartició de dades: modificar el topic de destí en funció del contingut.

Vegem un exemple, suposam que tenim una taula temperatures2 a la qual, hem afegit un id de sensor i la seva localització:

id	sensor_id	localitzacio	temperatura	datahora
1	101	Palma	22.5	2025-02-10 12:00:00
2	102	Eivissa	21.8	2025-02-10 12:05:00

Aquestes són les sentències SQL (en la base de dades sensors):

```
CREATE TABLE temperatures2 (  
  id INT PRIMARY KEY,  
  sensor_id INT NOT NULL,  
  localitzacio VARCHAR(100) NOT NULL,  
  temperatura FLOAT NOT NULL,  
  datahora DATETIME NOT NULL  
);  
INSERT INTO temperatures2 (id, sensor_id, localitzacio, temperatura, datahora) VALUES  
(1, 101, 'Palma', 22.5, '2025-02-10 12:00:00'),  
(2, 102, 'Eivissa', 21.8, '2025-02-10 12:05:00');
```

Volem que Kafka generi els missatges aplicant aquestes transformacions sobre les dades:

- Eliminar la columna "localitzacio" (perquè no la necessitam)
- Canviar "datahora" a "ingest_time"
- Assegurar que "temperatura" sigui un float
- Afegir un camp "origen" amb el valor "MySQL" per saber d'on venen les dades.

En el fitxer de configuració podem definir aquestes tres transformacions, a les que hem anomenat dropLocation, renameTimestamp i castTemperature, respectivament. Vegem-ho:


```
{
  "name": "mysql-temperatures2",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "jdbc:mysql://localhost/sensors",
    "connection.user": "kafka",
    "connection.password": "XXXXXXX",
    "table": "temperatures2",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "topic.prefix": "mysql_",

    "transforms": "dropLocation, renameTimestamp, castTemperature, addOrigin",

    "transforms.dropLocation.type":
    "org.apache.kafka.connect.transforms.ReplaceField$Value",
    "transforms.dropLocation.exclude": "localitzacio",

    "transforms.renameTimestamp.type":
    "org.apache.kafka.connect.transforms.ReplaceField$Value",
    "transforms.renameTimestamp.renames": "datahora: ingest_time",

    "transforms.castTemperature.type": "org.apache.kafka.connect.transforms.Cast$Value",
    "transforms.castTemperature.spec": "temperatura: float64",

    "transforms.addOrigin.type": "org.apache.kafka.connect.transforms.InsertField$Value",
    "transforms.addOrigin.static.field": "origen",
    "transforms.addOrigin.static.value": "MySQL"
  }
}
```

Vegem que hem aplicat aquestes 4 transformacions:

- "dropLocation": elimina el camp "localització".
- "renameTimestamp": canvia "timestamp" per "ingest_time".
- "castTemperature": converteix "temperatura" a float64.
- "addOrigin": afegeix el camp "origen" amb el valor "MySQL".

Aquest és el valor del "payload" dels dos missatges que es generen, un per a cada fila:

```
{
  "id": 1,
  "sensor_id": 101,
  "temperatura": 22.5,
  "ingest_time": 1739188800000,
  "origen": "MySQL"
}
{
  "id": 2,
  "sensor_id": 102,
  "temperatura": 21.799999237060547,
  "ingest_time": 1739189100000,
  "origen": "MySQL"
}
```

Podem trobar més informació sobre les transformacions a la documentació de Kafka: https://kafka.apache.org/documentation/#connect_transforms

6. Kafka Streams

Kafka Streams és una **llibreria de Java** per processar dades en temps real directament dins de Kafka. És diferent de Kafka Connect, que es fa servir per moure dades d'un lloc a un altre.

Kafka Streams serveix per **transformar, agregar i analitzar** les dades mentre flueixen per Kafka, sense necessitat d'un sistema extern com Flink o Spark.

La taula següent mostra les principals diferències entre Kafka Connect i Kafka Streams:

Característica	Kafka Connect	Kafka Streams
Objectiu	Moure dades entre sistemes	Processar dades en temps real
Codi necessari?	No (configuració JSON)	Sí (codi en Java o Scala)
Cas d'ús típic	Llegir d'una BD i escriure a HDFS	Filtrar, agrupar, calcular mitjanes
Escalabilitat	Amb més instàncies de Connectors	Automàtica, amb particions de Kafka

Alguns possibles exemples que mostren la utilitat de Kafka Streams són:

- Filtrar missatges, per exemple, que només es generin esdeveniments en el cas en què la columna "temperatura" tenguin un valor superior a 20°C.
- Fer operacions matemàtiques, com per exemple, calcular la mitjana de temperatura per cada sensor en el darrer dia.
- Fer conversions, com per exemple, convertir JSON en Avro o Parquet abans de guardar-ho en un altre *topic*.

Tal com ja hem mencionat, Kafka Streams és una llibreria de Java, que ens permetrà escriure classes Java per dur a terme aquestes operacions. No obstant això, en Python existeixen diverses llibreries que permeten treballar amb fluxos de dades de manera similar. Aquestes llibreries no implementen Kafka Streams directament, però ofereixen funcionalitats equivalents per consumir, processar i produir dades en Kafka. La més coneguda d'elles és [Faust](#).

Faust, però, es va deixar de mantenir en 2020. [Faust-streaming](#) és un *fork* de Faust, que avui en dia és l'alternativa més utilitzada per treballar amb funcionalitat semblants a Kafka Streams.

La podem instal·lar amb:

```
pip install faust-streaming
```

Per veure com funciona, volem llegir els missatges del tòpic "mysql_temperatures2" i filtrar només aquells que tenen una temperatura superior a 22°. El resultat el posarem en un nou *topic* "faust_temperatures_altes":

```
import faust

app = faust.App('temperature_app', broker='kafka://localhost:9092')

class TemperatureRecord(faust.Record):
    id: int
    sensor_id: int
    temperatura: float
    ingest_time: int
    origen: str

# Topic d'entrada i de sortida
input_topic = app.topic('mysql_temperatures2', value_type=TemperatureRecord)
output_topic = app.topic('faust_temperatures_altes', value_type=TemperatureRecord)

@app.agent(input_topic)
async def filter_high_temperatures(stream):
    async for record in stream:
        if record.temperatura > 22:
            await output_topic.send(value=record)

if __name__ == '__main__':
    app.main()
```

Per executar-ho, ho feim amb la següent instrucció:

```
faust -A faust_temperatures worker --loglevel=info
```

Finalment, per comprovar que tot ha anat bé, amb el client consumidor recuperem els esdeveniments del topic faust_temperatures_altes:

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

7. Apache Flink

Apache Flink és un motor de processament de dades en streaming dissenyat per gestionar fluxos d'esdeveniments amb alta disponibilitat, precisió i eficiència. Apareix com una resposta a les limitacions dels primers sistemes de processament en temps real, com Storm o Spark Streaming.

Podria semblar que Kafka i Flink se solapen, ja que tots dos són eines per a la gestió de *streams*. La realitat, però, és que són complementaris i molt sovint es fan servir conjuntament.

Kafka per si sol "només" proporciona un sistema distribuït de missatgeria i emmagatzematge de fluxos de dades, sense eines sofisticades per processar-los. Tal com he vist, les transformacions SMTs i, sobretot, Kafka Streams (i Faust) volen cobrir aquesta limitació i permeten fer algunes transformacions en streaming dins de Kafka, però són molt limitades. Flink millora aquestes capacitats oferint una arquitectura més flexible, potent i escalable per a processaments complexos, com finestres temporals avançades, gestió d'estat amb precisió i integració nativa amb fonts de dades diverses.

En qualsevol cas, Flink té una relació amb Kafka és molt estreta: Flink pot consumir dades directament des de Kafka, transformar-les en temps real i tornar-les a escriure en altres *topics*, complementant així Kafka Connect per crear fluxos de dades més intel·ligents i processats abans d'arribar a la seva destinació final.

D'aquesta manera, l'ús combinat de Kafka i Flink permet construir pipelines de dades avançats. Kafka actua com el sistema central de missatgeria, assegurant la disponibilitat i persistència dels missatges, mentre que Flink pot consumir aquestes dades per fer transformacions, agregacions o fins i tot anàlisis en temps real. Per exemple, un cas d'ús habitual seria llegir dades de sensors mitjançant Kafka Connect des d'una base de dades com MySQL, enviar-les a Kafka i, a continuació, utilitzar Flink per filtrar valors rellevants, calcular mitjanes en finestres temporals o detectar anomalies. Els resultats podrien ser reenviats a un altre *topic* de Kafka per a un posterior emmagatzematge o visualització.

No entrarem en més detalls sobre Apache Flink en aquest curs, però en un tema centrat en fluxos de dades, val la pena almenys conèixer la seva existència i utilitat.



Podeu trobar més informació a la web de Flink: <https://flink.apache.org/>

També pot es molt recomanable aquesta sèrie de dos articles sobre Flink i la seva integració amb Kafka:

- Part 1: [Navigating Apache Flink: A Software Engineer's Onboarding Journey](#)
- Part 2: [First-Time Kafka-Flink Integration: Stream Processing Insights](#)

AMPLIACIÓ