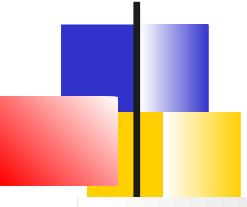


Kafka : Messagerie distribuée avec Apache Kafka

David THIBAU - 2025

david.thibau@gmail.com



Agenda

Introduction à Kafka

- Le projet Kafka
- Cas d'usage
- Concepts

Cluster

- Noeuds du cluster
- Distribution / Installation
- Utilitaires Kafka
- Outils graphiques

Kafka APIs

- Producer API
- Consumer API
- Schema Registry
- Connect API
- Admin API

Garanties Kafka

- Mécanismes de réPLICATION
- At Most Once, At Least Once
- Exactly Once
- Débit, latence, durabilité
- Stockage et rétention

KafkaStream et kSQLDB

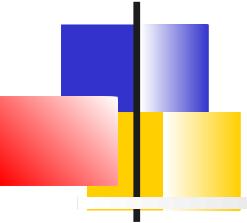
- Concepts
- Opérateurs stateless
- Opérateurs stateful

Sécurité

- Listeners
- TLS
- Authentification via SASL
- ACLs
- Quotas

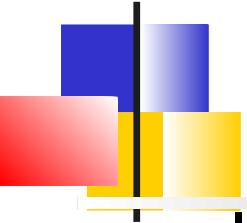
Frameworks Java

- Vert.x
- MicroProfile Reactive Messaging
- Spring Kafka
- Spring Cloud Stream



Introduction à Kafka

Le projet Kafka
Cas d'usage
Concepts



Origine

Initié par *LinkedIn*, mis en OpenSource en 2011

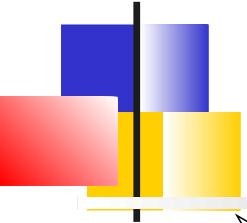
Écrit en *Scala* et *Java*

Au départ, un message broker gérant une file de messages

A évolué pour devenir une plate-forme de streaming d'événements temps-réel

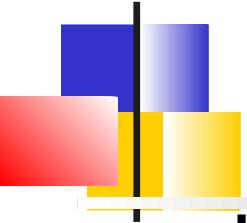
Organisé en cluster, taillé pour le BigData, il est basé sur l'abstraction « d'un journal de commit distribué »

Maintenu par *Confluent* depuis 2014



Objectifs

- Découpler producteurs et consommateurs de messages
- Persister les messages afin qu'ils puissent être consommés par de nombreux consommateurs, (éventuellement à posteriori)
- Atteindre de très haut débit et une latence faible
- Scaling horizontal flexible
- Offrir des garanties de fiabilité de la livraison de messages, malgré des défaillances !

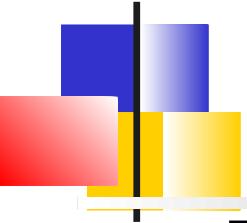


Fonctionnalités

Kafka a trois capacités clés:

- Publier et s'abonner à des flux de messages¹ avec certaines garanties de fiabilité.
- Stocker les flux de messages de manière durable et tolérante aux pannes.
- Traiter, transformer les flux de messages au fur et à mesure qu'ils se produisent.

1. Dans la suite des slides on utilise de façon non-différenciés les termes *message, événement, enregistrement*



Points forts

Très bonne scalabilité et flexibilité

- Gestion des abonnements multiples
- Facilité d'extension du cluster

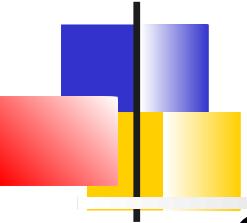
Très bonne performance

Disponibilité et tolérance aux fautes

Rétention sur disque

Traitements distribués d'événements

Intégration avec les autres systèmes



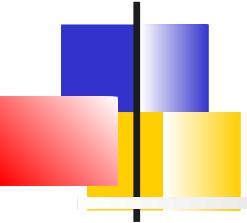
Confluent

Créé en 2014 par *Jay Kreps, Neha Narkhede, et Jun Rao*

Mainteneur principal d'Apache Kafka

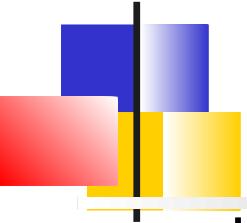
Plate-forme *Confluent* :

- Une distribution de Kafka
- Des librairies clientes dans de nombreux langages
- Fonctionnalités commerciales additionnelles



Introduction à Kafka

Le projet Kafka
Cas d'usage
Concepts



Kafka vs Message broker traditionnel

Kafka peut être utilisé comme **message broker** permettant de découpler un service producteur de services consommateurs

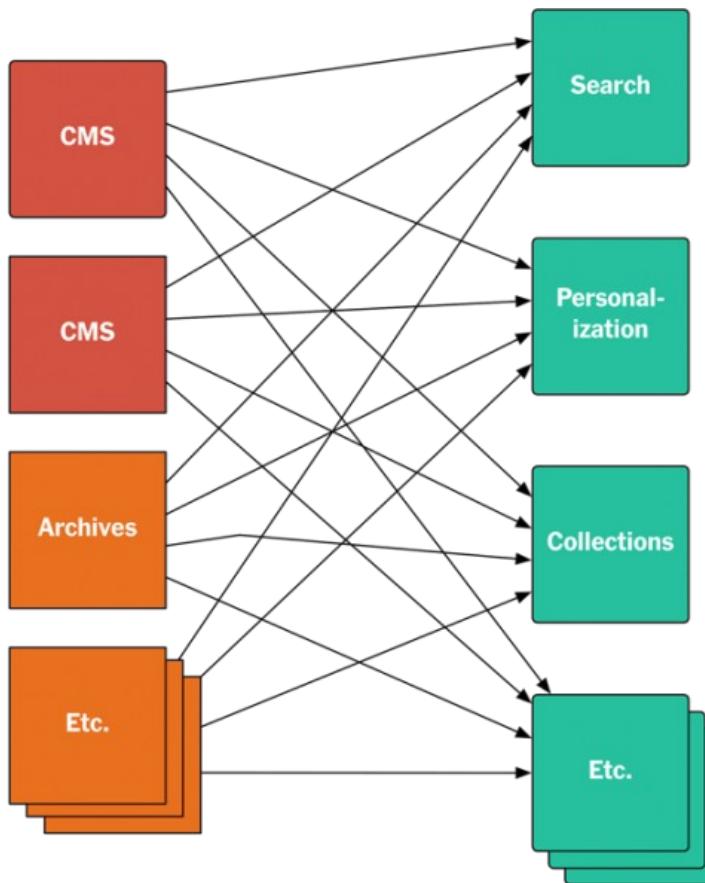
- Kafka n'offre que le modèle **PubSub**.
- Grâce au concept de **groupe de consommateur**, ce modèle est scalable
- Kafka offre une **garantie plus forte** sur l'ordre de livraison des messages
- Kafka ne supprime pas les messages après consommation. Ils peuvent être **consommés à posteriori**

=> Idéal comme plate-forme d'intégration entre services :
Architecture micro-services, ESB

Exemple ESB (New York Times)

Avant

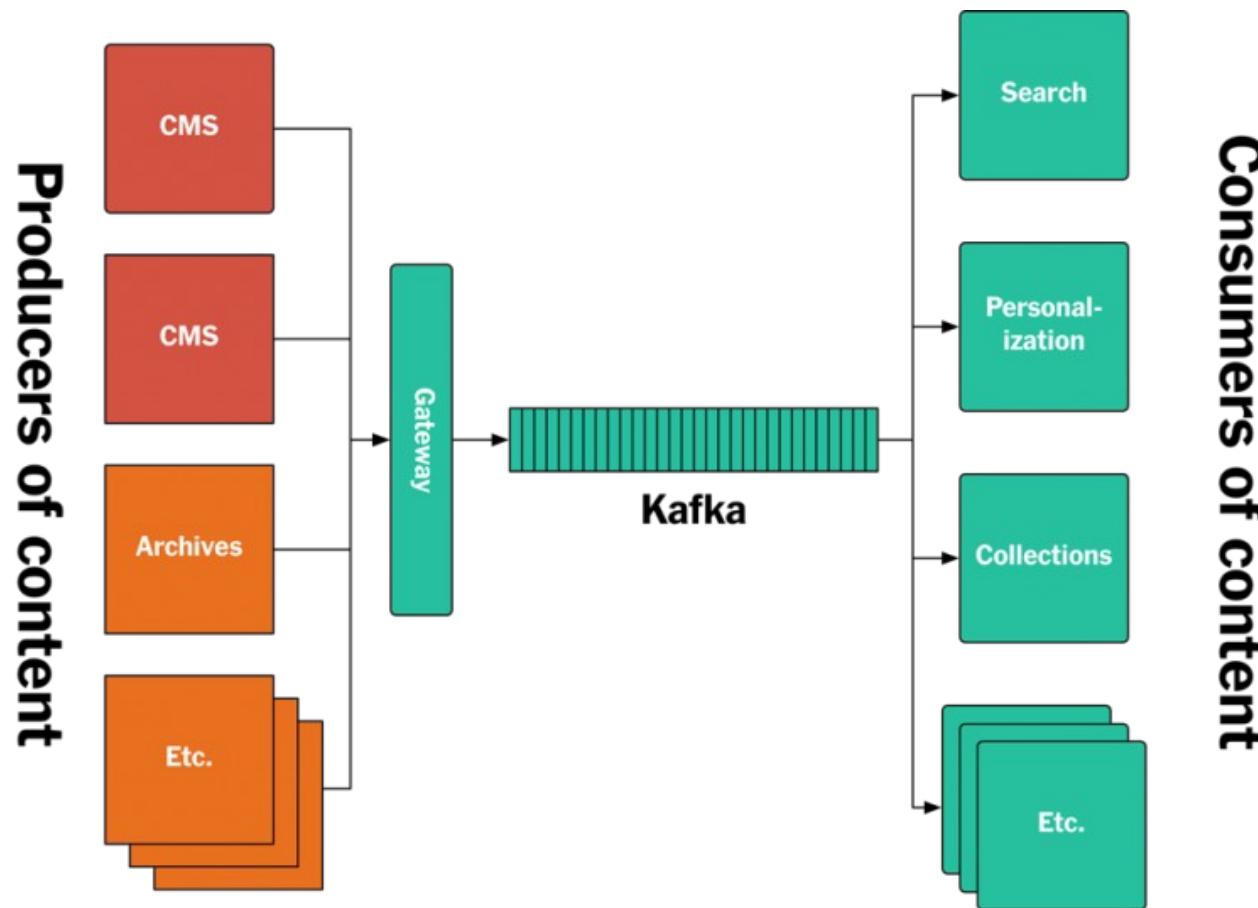
Producers of content

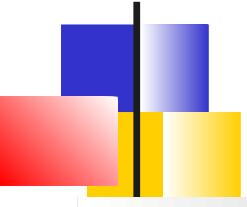


Consumers of content

Exemple ESB (New York Times)

Après





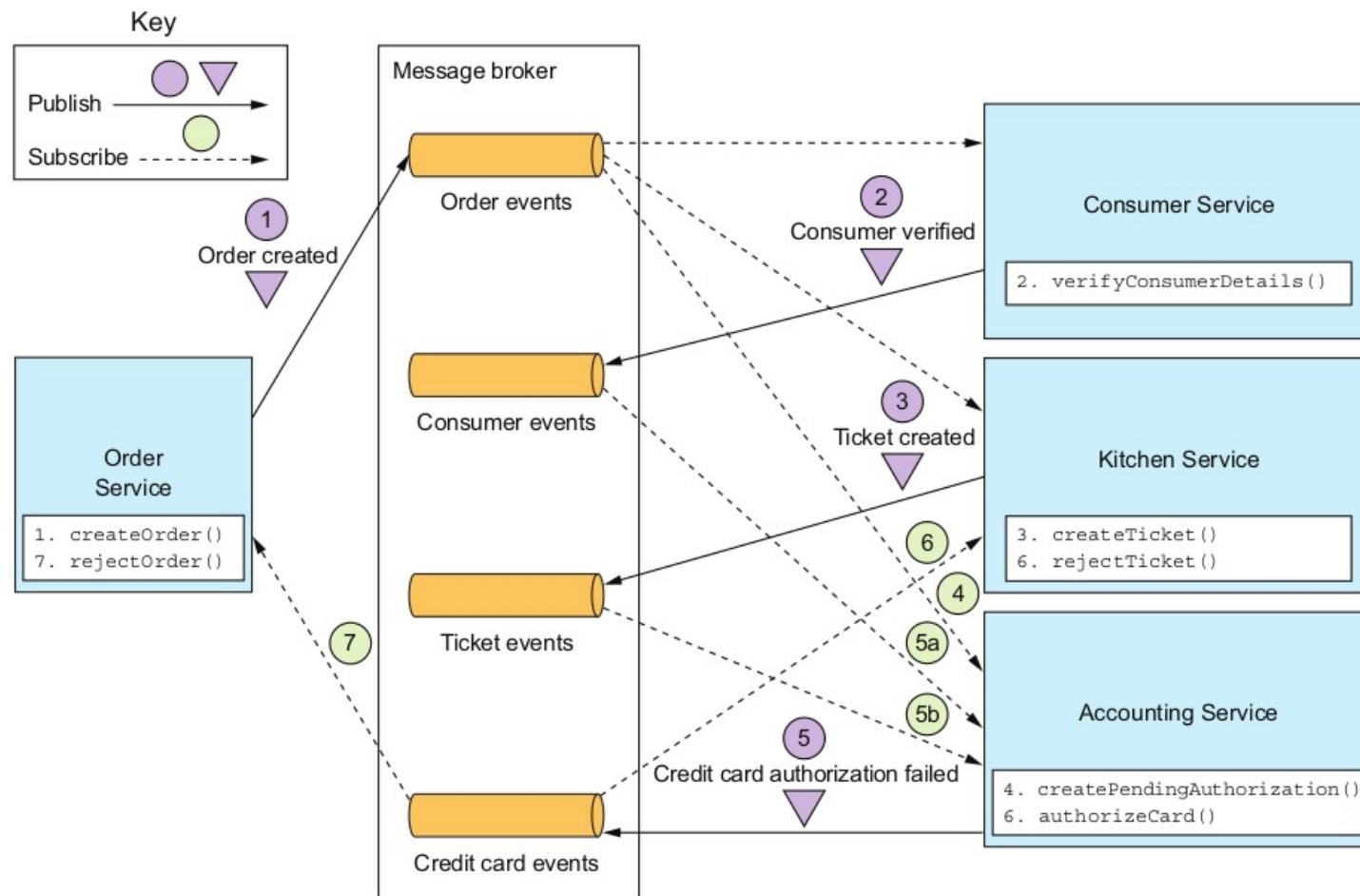
Exemple : micro-services

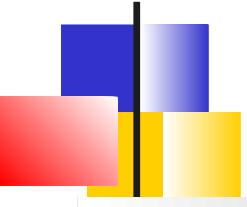
Kafka est souvent un « must-to-have » dans les architectures micro-services

- Permet tous les styles d'interaction :
One way notification,
Requête/Réponse synchrone asynchrone , ,
Publish and Subscribe avec ou sans réponse
- Cité dans de nombreux patterns micro-services, par exemple SAGA¹ (~ Transaction distribuée)

1. <https://microservices.io/patterns/data/saga.html>

Exemple Micro-services Message Broker et Design pattern SAGA





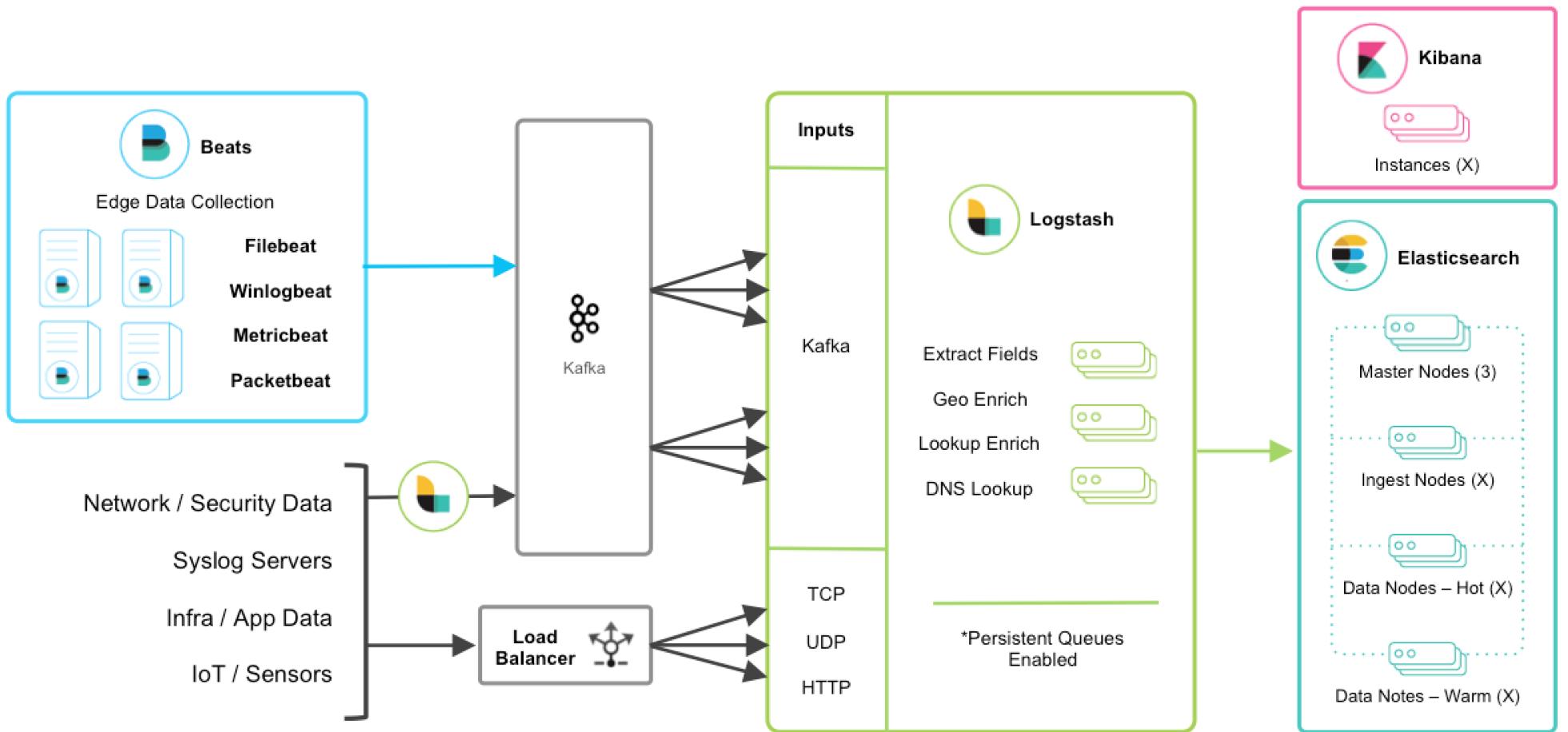
Kafka : Data Buffering

Les *topics* Kafka peuvent être utilisés pour bufferiser les événements provenant de multiples sources

- Ces événements alimentent des pipelines de traitement et transformation destinées à des solutions d'analyse temps-réel, d'alerting ou de machine learning
- Les topics peuvent alors absorber les pics de charge

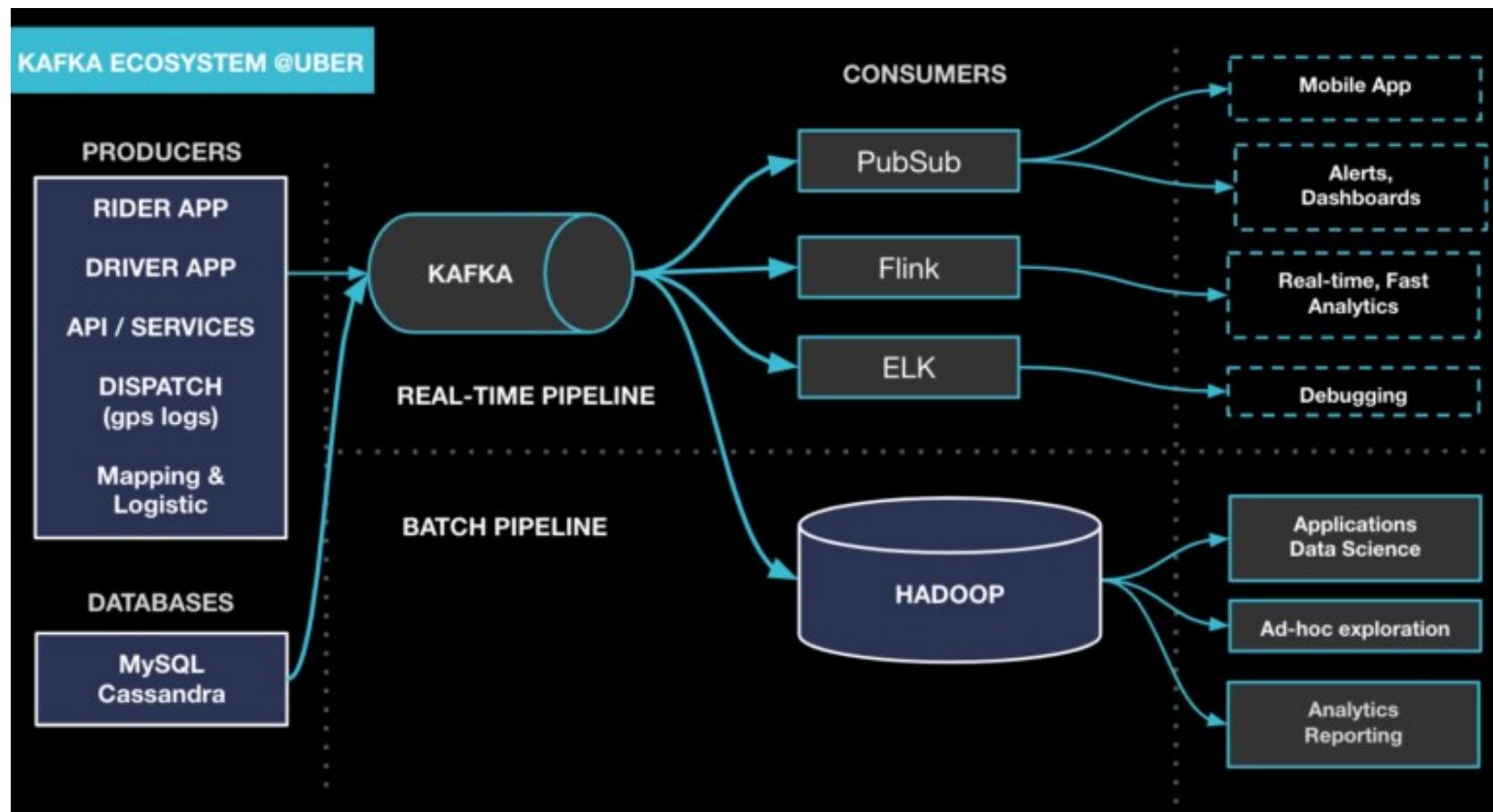
Exemple Architecture ELK

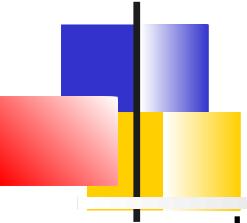
Bufferisation des événements



Ingestion massive de données

Exemple Uber





Architecture Event-driven

Les architectures *event-driven* sont une alternative intéressante dans un contexte micro-services.

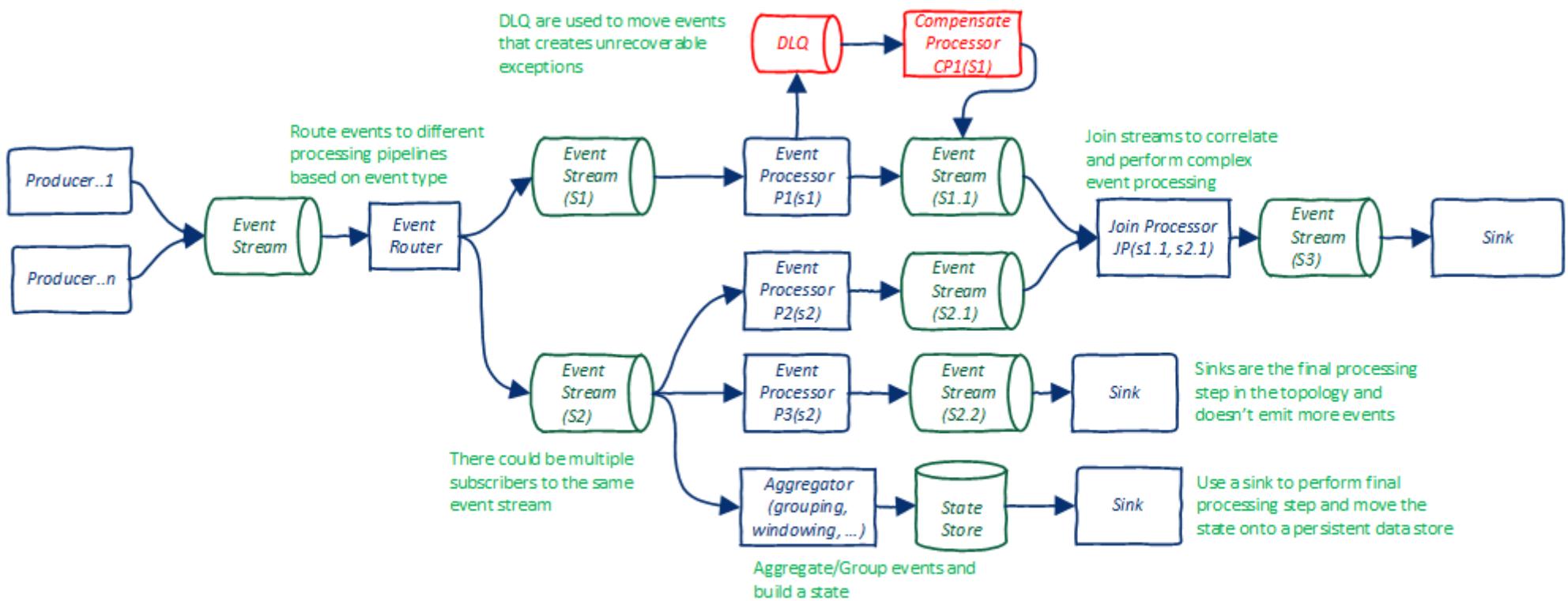
- Cela produit généralement des architectures plus souples et plus réactives.

Chaque micro-service consomme en continu des événements :

- Lit un ou plusieurs topics Kafka en entrée
- Effectue un traitement
- Écrit vers un ou plusieurs topics de sortie

Kafka Stream, Spring Cloud Stream, Spring Cloud Data Flow, Smallrye Reactive Messaging facilitent ce type d'architecture

Event-driven architecture

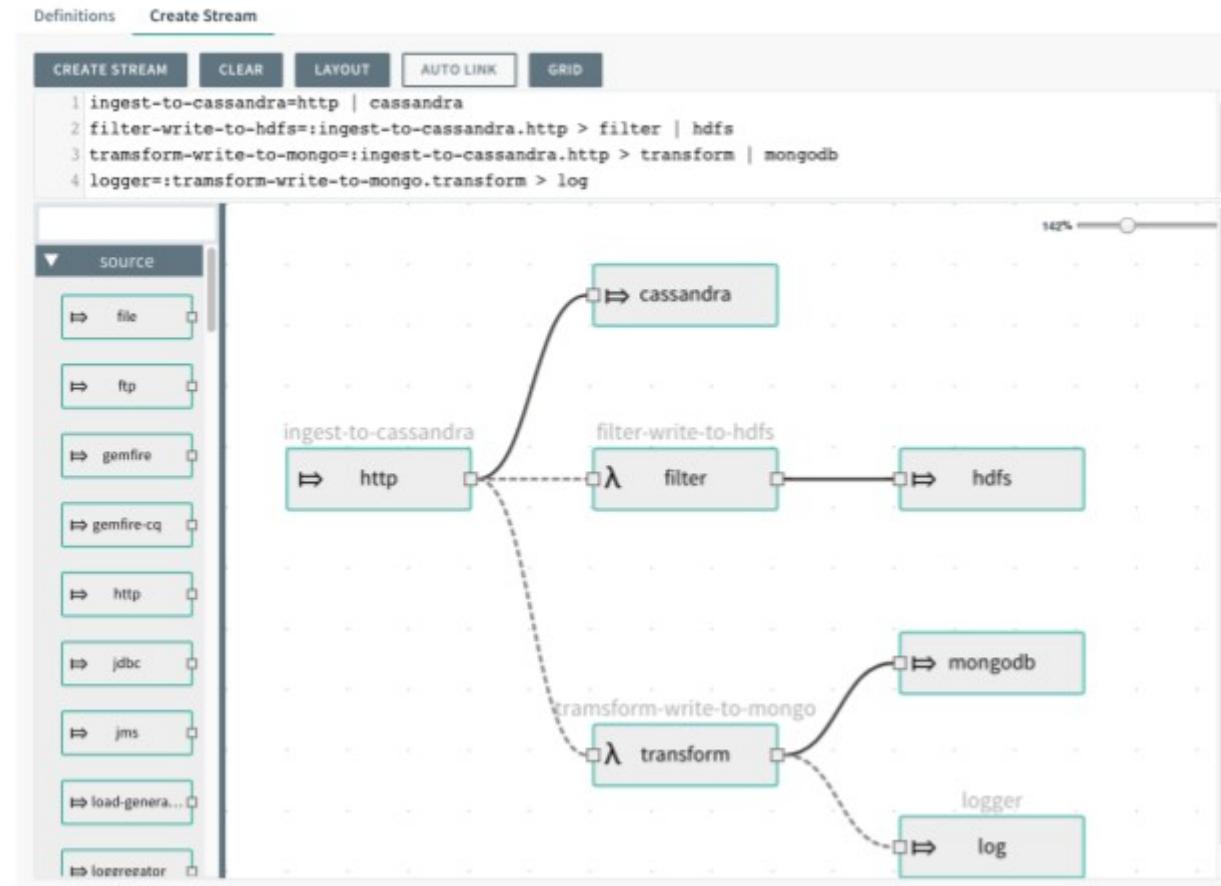


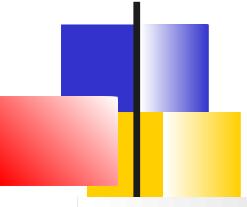
Event Processor – could be a microservice, serverless function, etc – which implements the logic of a particular processing step

Data Stream Exemple Spring Cloud Data Flow

Streams

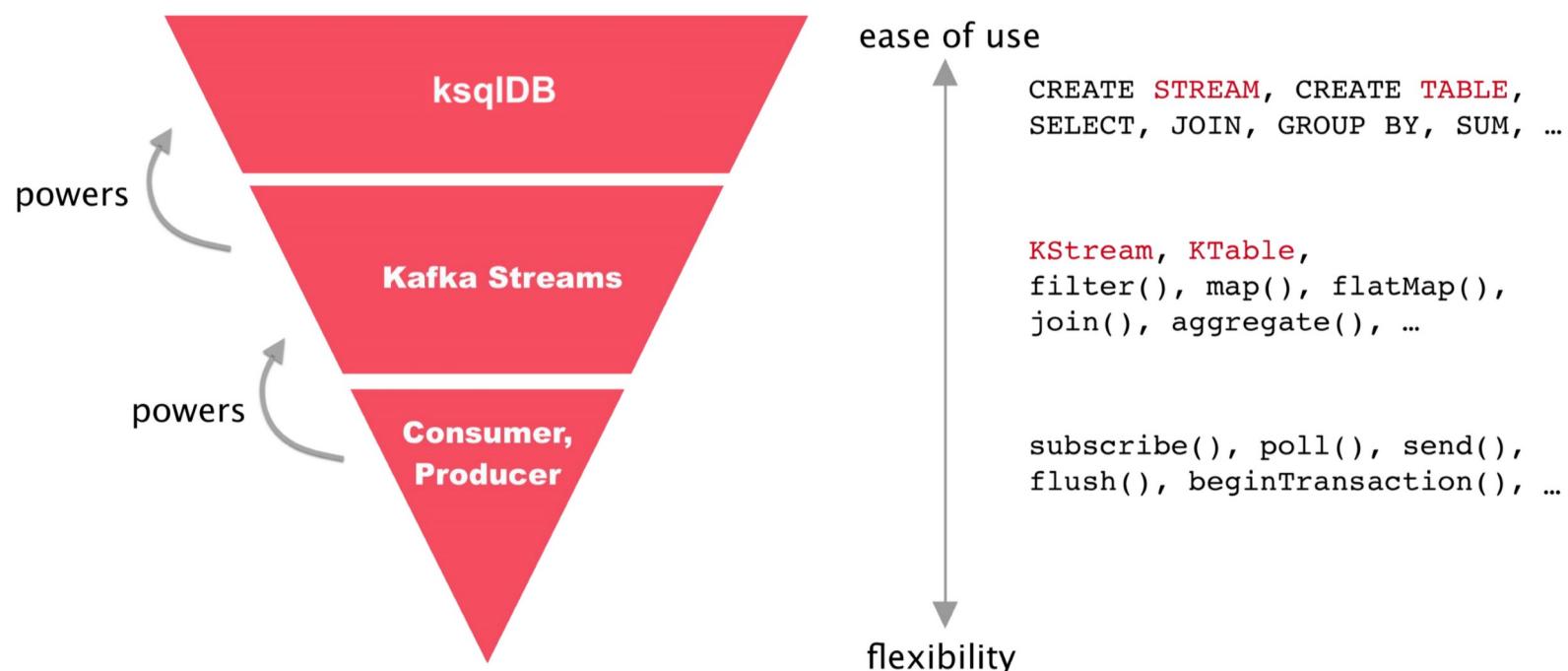
Create a stream using text based input or the visual editor.





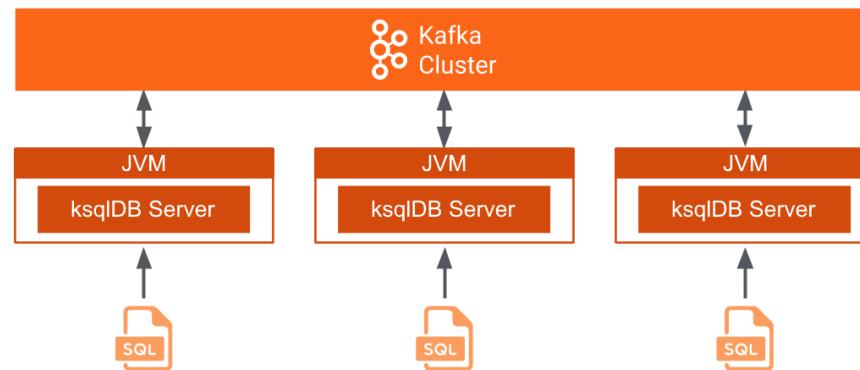
KsqlDB

KsqlDB est une abstraction au dessus de KafkaStream permettant de bâtir ces applications via des instructions SQL



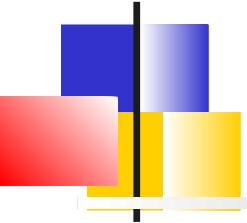
Architecture ksqlDB

ksqlDB Standalone Application (Headless Mode)



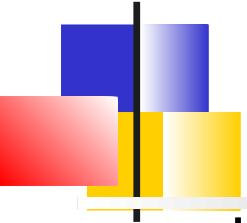
Intégration via

- API REST et ksqlCli
- Librairies : Clients Java, Python, NodeJS



Introduction à Kafka

Le projet *Kafka*
Cas d'usage
Concepts

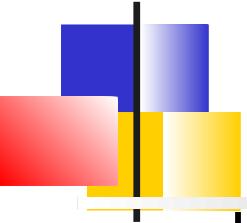


Concepts de base

Kafka s'exécute en **cluster** sur un ou plusieurs serveurs (**brokers**) pouvant être déployés dans différents data-center.

Le cluster Kafka stocke des flux d'enregistrements : les **records** dans des rubriques : les **topics** .

Chaque enregistrement se compose d'une clé éventuelle, d'une valeur, d'un horodatage et éventuellement des entêtes.

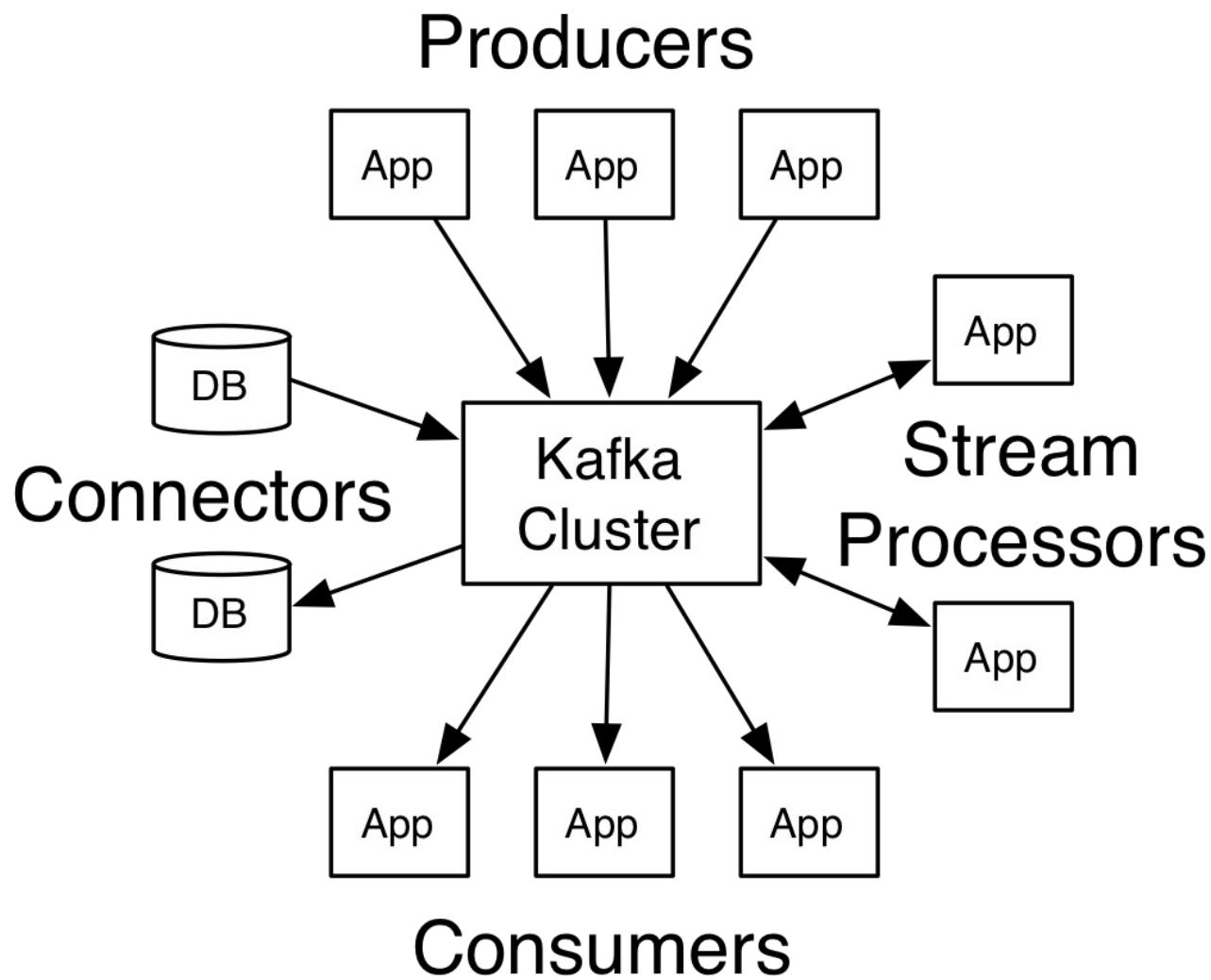


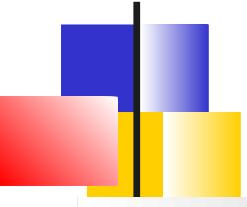
APIs

Kafka propose 5 principales APIs :

- L'API **Producer** permet à une application de publier un flux sur un ou plusieurs topics Kafka.
- L'API **Consumer** permet à une application de s'abonner à un ou plusieurs topics et de traiter le flux d'enregistrements associé.
- L'API **Streams** permet à une application d'agir comme un processeur de flux, consommant un ou plusieurs topic d'entrée et produisant un flux de sortie vers un ou plusieurs topics.
- L'API **Connector** permet de créer et d'exécuter des producteurs ou des consommateurs à partir de système tierces (BD, fichiers, STDOUT, ...)
- L'API **Admin** permet de gérer les topics et le cluster

APIs





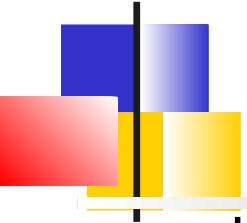
Protocole Client/Serveur

Dans Kafka, la communication entre les clients et les serveurs s'effectue via un protocole TCP simple, performant et indépendant du langage.

- Ce protocole est versionné et maintient une compatibilité ascendante avec les versions plus anciennes.

Apache fournit un client Java, mais les clients sont disponibles dans de nombreuses langages.¹

1. <https://cwiki.apache.org/confluence/display/KAFKA/Clients>



Topic

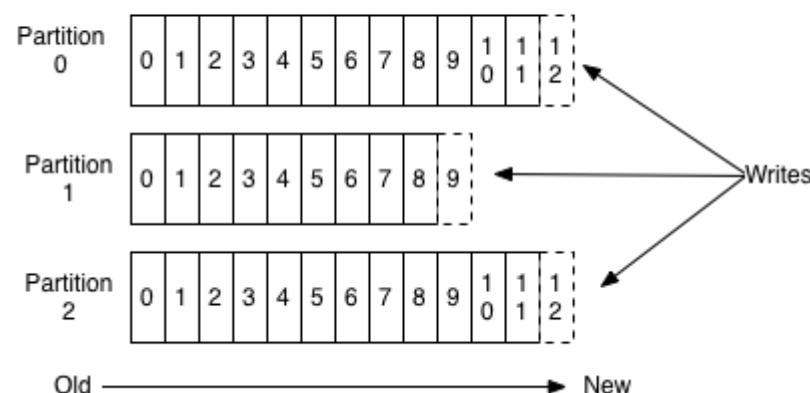
Les *records* sont publiés vers des **topics**.

Les *topics* de Kafka peuvent avoir Zéro, Un ou de multiples abonnés

Les topics peuvent être **partitionnés**.

Le cluster Kafka conserve donc un journal partitionné

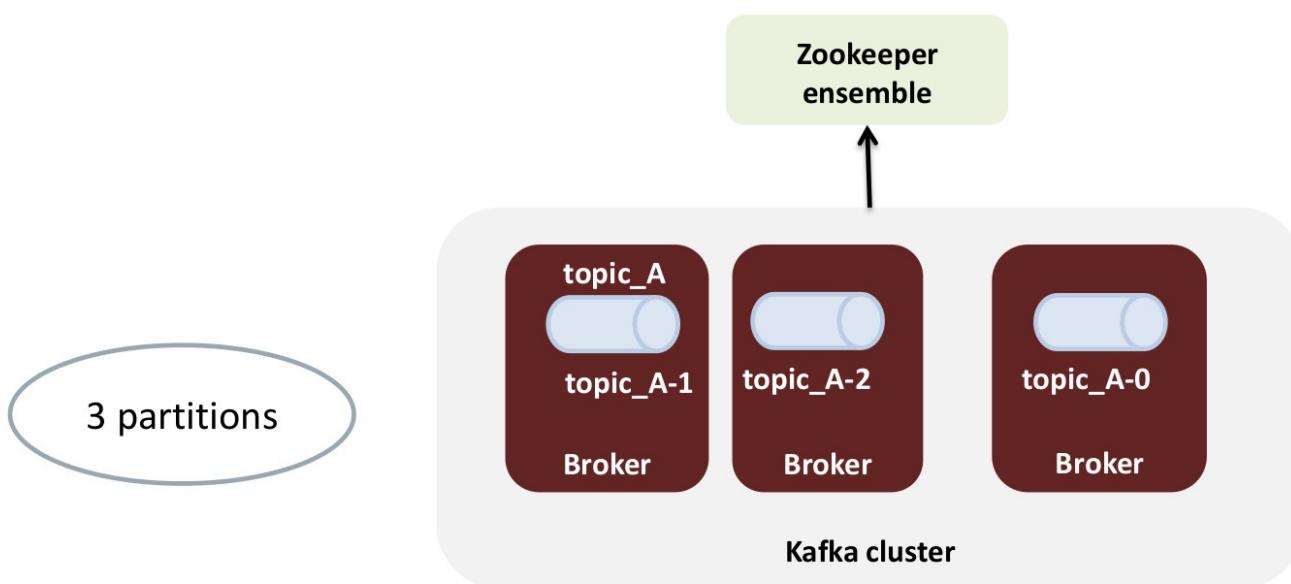
Anatomy of a Topic



Apport des partitions

Les partitions autorisent le parallélisme et augmentent la capacité de stockage en utilisant les capacités disque de chaque nœud.

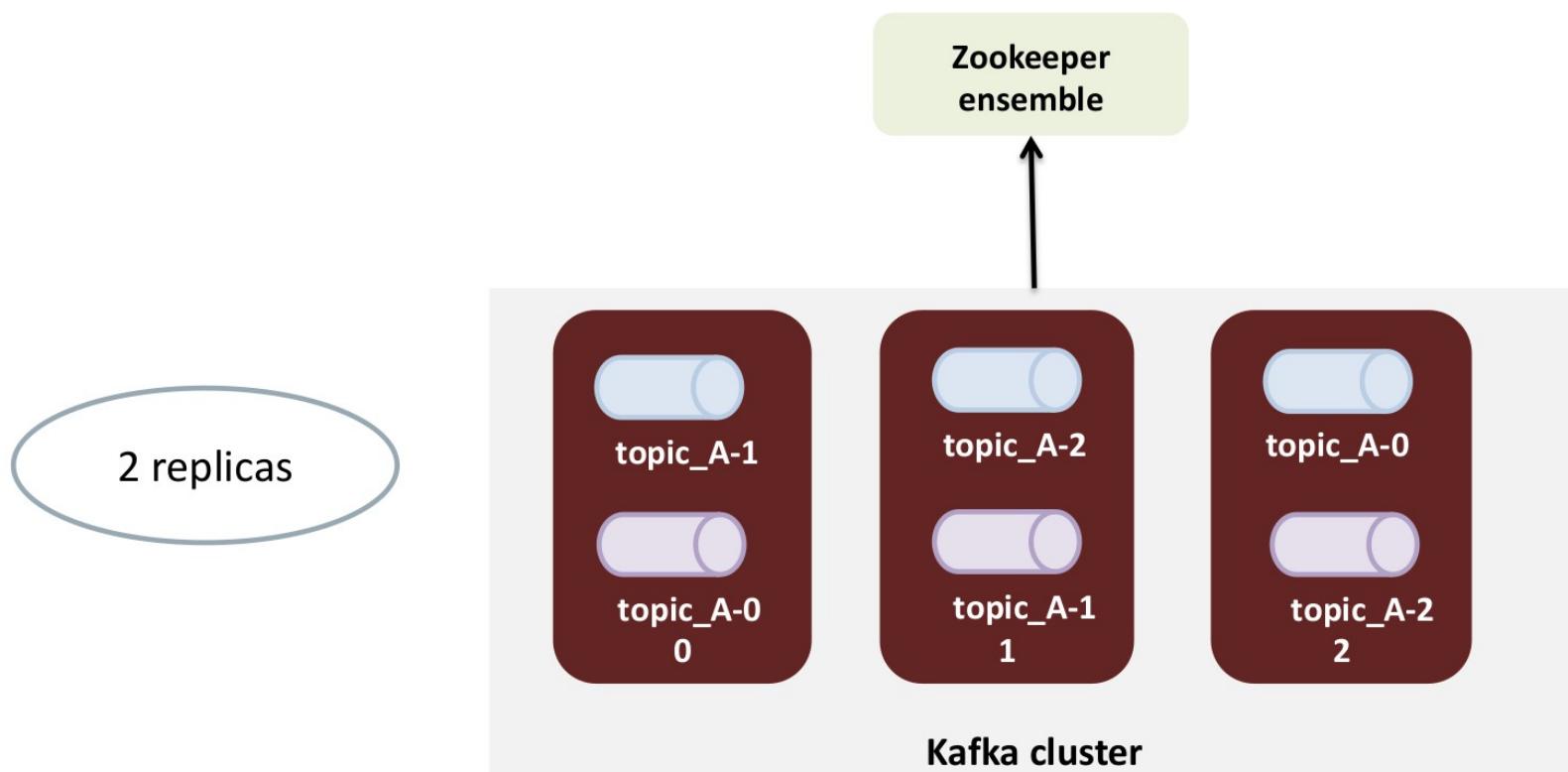
L'ordre des messages n'est garanti qu'à l'intérieur d'une partition
Le nombre de partition est fixé à la création du *topic*. Il peut être augmenté mais pas réduit.

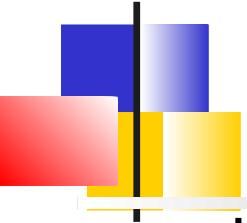


RéPLICATION

Les partitions peuvent être **répliquées**

- La réPLICATION permet la tolérance aux pannes et la durabilité des données





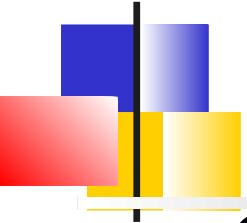
Distribution des partitions

Les partitions sont réparties sur les instances du cluster.

Les répliques sont distribuées sur des instances différentes

Pour chaque partition répliquée, une des instances agit comme **maître (leader)**. Les autres comme **suiveurs (follower)**

- Le maître coordonne les lectures et les écritures sur la partition
- Les suiveurs répliquent passivement le maître
- Si le maître défaillie, un processus d'élection choisit un autre maître parmi les répliques



Partition et offset

Chaque partition est une séquence
ordonnée et immuable
d'enregistrements.

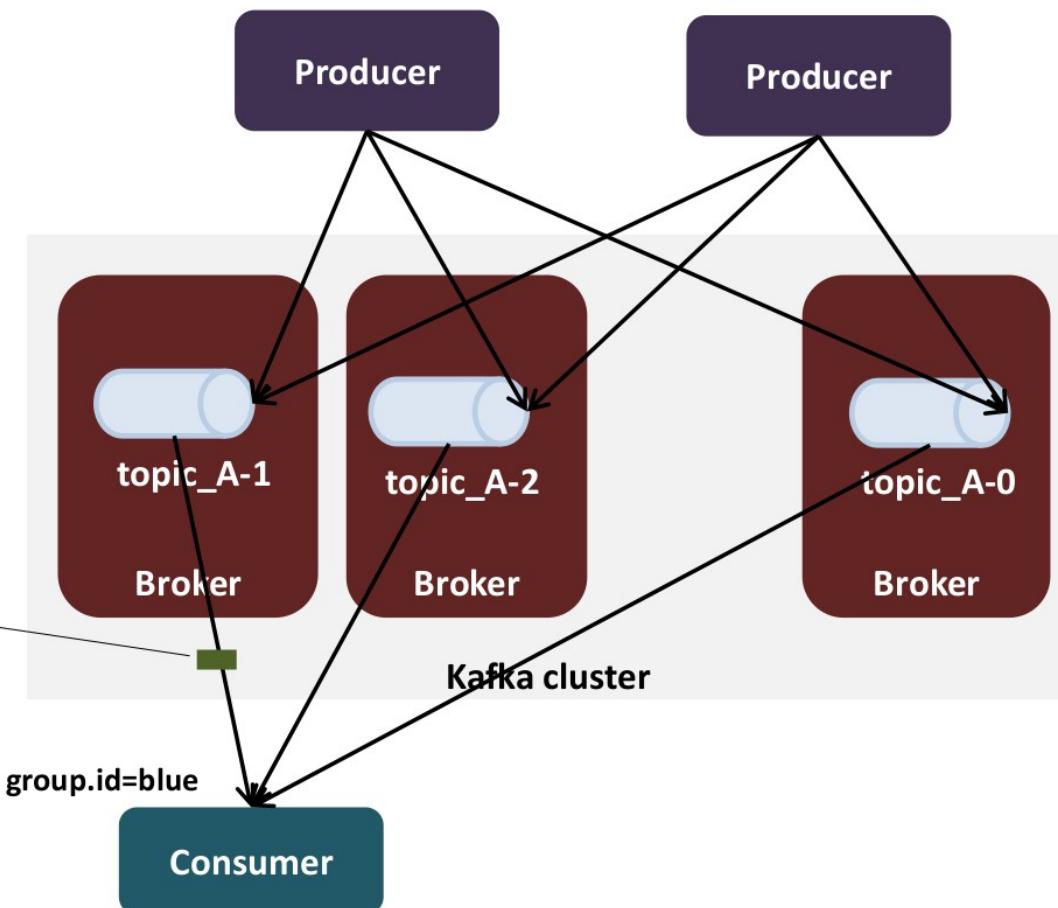
Un numéro d'identification séquentiel
nommé **offset** est attribué à chaque
enregistrement.

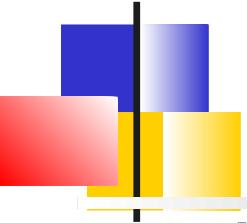
Le cluster Kafka conserve durablement tous
les enregistrements publiés, qu'ils aient
ou non été consommés, en utilisant une
période de rétention configurable.

Clients du cluster

Les producteurs et consommateurs sont connectés à tous les brokers détenant le *topic*

message (record, event)
• key-value pair
• string, binary, json, avro
• serialized by the producer
• stored in broker as byte arrays
• deserialized by the consumer



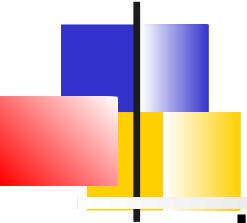


Routing des messages

Les producteurs sont responsables du choix de la partition en fonction de l'enregistrement

Cela peut être fait

- via une stratégie Round-Robin assurant un équilibrage de charge
- En fonction des données de l'enregistrement. Typiquement, la clé

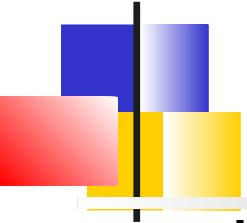


Groupe de consommateurs

Les consommateurs sont taggés avec un nom de **groupe**

- Chaque enregistrement d'un topic est remis à une instance de consommateur au sein de chaque groupe.
- Les instances de consommateurs peuvent se trouver dans des threads, processus ou machines distincts.

=> Scalabilité et Tolérance aux fautes

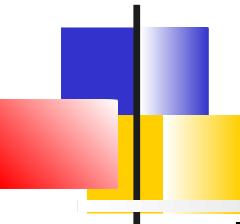


Offset consommateur

La seule métadonnée conservée pour un groupe de consommateurs est son **offset** du journal.

Cet offset est contrôlé par le consommateur:

- normalement, le consommateur avance son offset au fur et à mesure de sa lecture des enregistrements,
- mais, il peut consommer dans l'ordre qu'il souhaite.
Par exemple, retraitier les données les plus anciennes ou repartir d'un offset particulier.



Consommateur vs Partition

Rééquilibrage dynamique

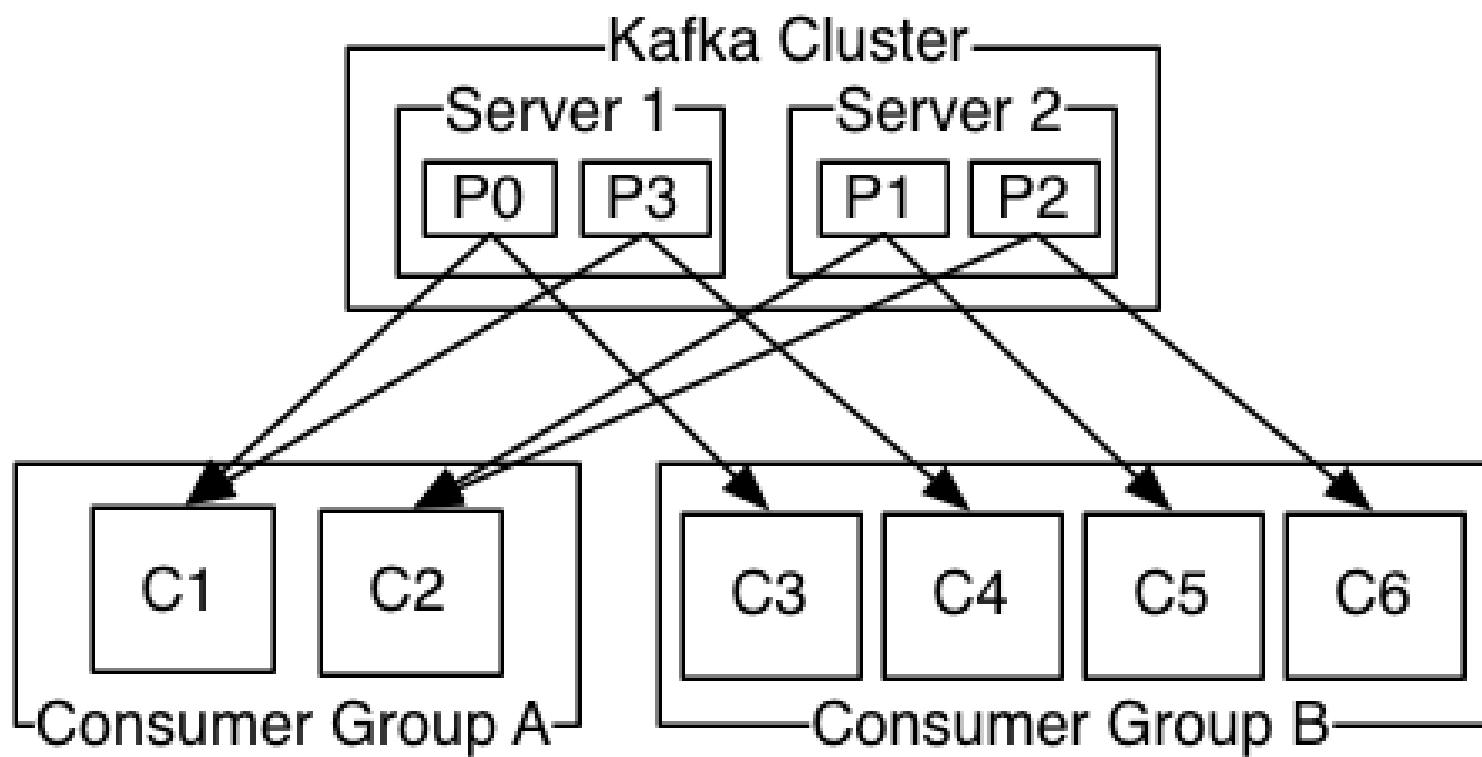
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

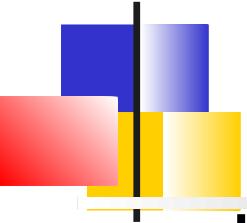
- A tout moment, une partition est associée exclusivement à un consommateur

Ceci est géré dynamiquement par le protocole Kafka.

- Si de nouvelles instances rejoignent le groupe, elles reprendront certaines partitions des autres membres du groupe;
- si une instance meurt, ses partitions seront distribuées aux instances restantes.

Exemple

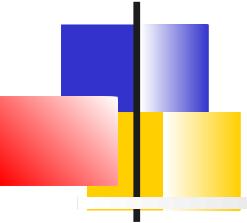




Ordre des enregistrements

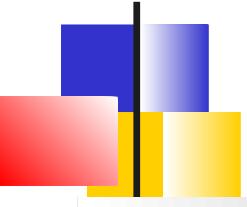
Kafka garantit un ordre total sur les enregistrements d'une partition, mais pas sur les différentes partitions d'un topic.

- L'ordre sur les partitions, combiné à la possibilité de partitionner les données par une clé est suffisant pour la plupart des applications.
- Si une application nécessite un ordre strict sur tous les enregistrements. Il faut que le topic n'est qu'une seule partition



Cluster

Nœuds du cluster
Distributions / Installation
Utilitaires *Kafka*
Outils graphiques



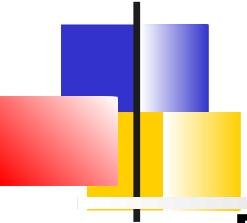
Cluster

Kafka est exécuté comme un cluster d'un ou plusieurs **serveurs** pouvant s'étendre sur plusieurs centres de données.

- Certains de ces serveurs appelés les **brokers** forment la couche de stockage.
- Un serveur est désigné **contrôleur**¹.
Son rôle est de prendre des décisions concernant le cluster comme l'affectation de partitions

Avant la version 3.x, un cluster Kafka nécessitait également un ensemble Zookeeper² permettant de stocker les métadonnées nécessaires au contrôleur

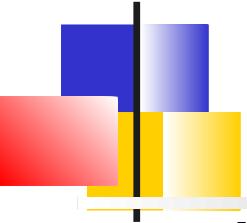
1. *Lors de la présence d'un contrôleur, le cluster s'exécute en mode Kraft*
2. *Voir annexe Zookeeper*



Nombre de brokers

Pour déterminer le nombre de brokers :

- Premier facteur :
Le niveau de **tolérance aux pannes** requis
- Second facteur :
La **capacité de disque** requise pour conserver les messages et la quantité de stockage disponible sur chaque *broker*.
- 3ème facteur :
La capacité du cluster à traiter **le débit** de requêtes en profitant du parallélisme.



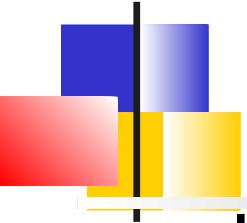
Configuration et démarrage d'un nœud

La distribution fournit un script de démarrage :

kafka-server-start.sh

Chaque nœud est démarré via ce script qui lit sa configuration :

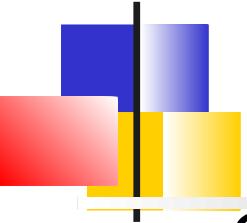
- Dans un fichier ***server.properties***
- Des propriétés peuvent être surchargées par la commande en ligne via l'option **--override**



Principales configuration

Les configurations principales sont :

- ***cluster.id*** : Identique pour chaque serveur.
- ***node.id*** : Différent pour chaque broker
- ***process.roles*** : Les rôles possibles du nœud : *broker*, *controller* ou les 2 à la fois
- ***log.dirs*** : Ensemble de répertoires de stockage des enregistrements
- ***listeners*** : Ports ouverts pour communication avec les clients et inter-broker



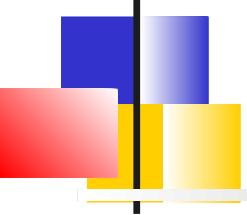
Contrôleurs

Certains processus Kafka sont donc des contrôleurs, ils participent aux quorums sur les méta-données.

- Une majorité des contrôleurs doivent être vivants pour maintenir la disponibilité du cluster
- Ils sont donc typiquement en nombre impair. (3 pour tolérer 1 défaillance, 5 pour 2)

Tous les serveurs découvrent les votants via la propriété **controller.quorum.voters** qui les liste en utilisant l'*id*, le *host* et le *port*

controller.quorum.voters=id1@host1:port1, id2@host2:port2, id3@host3:port3



Configuration par défaut des topics

num.partitions : Par défaut 1

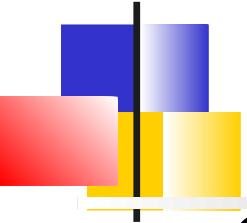
default.replication.factor : Par défaut 1

min.insync.replicas : Joue sur les garanties de message. Par défaut 1

log.retention.ms : Durée de rétention de messages.
Par défaut 7J

log.segment.bytes : Taille d'un segment. 1Go par défaut

message.max.bytes : Taille max d'un message. 1Go par défaut

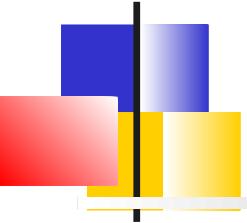


Cluster ID

Chaque cluster a un **ID** qui doit être présent dans les données de configurations des répertoires de stockage¹

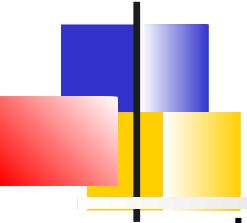
L'outil *kafka-storage.sh* permet de générer un ID aléatoire, puis de formatter les répertoires de stockage à partir de la config d'un borker.

1. Dans le mode zookeeper *cluster.id* est présent dans *server.properties*



Cluster Kafka

Cluster
Distributions / Installation
Utilitaires *Kafka*
Outils graphiques



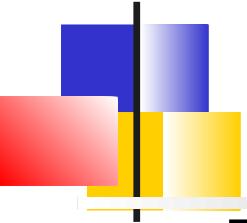
Introduction

Kafka peut être déployé

- sur des serveurs physiques, des machines virtuelles ou des conteneurs
- sur site ou dans le cloud

Différentes distributions peuvent être récupérées :

- Binaire chez Apache.
OS recommandé Linux + pré-installation de Java
(Support de Java 17 pour 3.1.0)
- Images docker (Apache, Bitnami par exemple)
- Packages Helm (Bitnami par exemple)
- Confluent Platform (Téléchargement ou cloud)

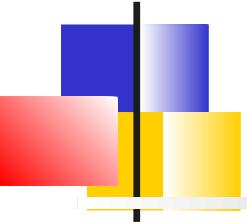


Installation à partir de l'archive

Téléchargement, puis

```
# tar -zxf kafka_<version>.tgz
# mv kafka_<version> /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk17
# cd /usr/local/kafka/bin
# ./kafka-server-start.sh -daemon ../config/server.properties
#
```

Les propriétés de configuration définies dans ***server.properties*** peuvent être surchargées en ligne de commande par l'option **--override**



Images bitnami

Images basées sur minideb (minimaliste Debian)

```
docker run -d --name kafka-server \
--network app-tier \
-e ALLOW_PLAINTEXT_LISTENER=yes \
bitnami/kafka:latest
```

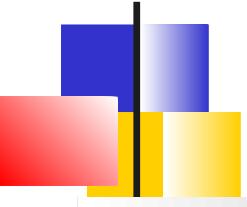
Configuration via variables d'environnement :

Variables spécifiques bitnami.

Ex : **BITNAMI_DEBUG**, **ALLOW_PLAINTEXT_LISTENER**, ...

Propriétés Kafka préfixée par *KAFKA_CFG_*

Ex : **KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE**



Kubernetes

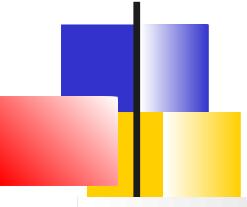
Bitnami propose des packages Helm pour déployer vers Kubernetes ou des clouds

```
helm install my-release  
oci://registry-1.docker.io/bitnamicharts/kafka
```

De nombreux paramètres sont disponibles :

- *replicaCount* : Nombre de répliques
- *config* : Fichier de configuration
- *existingConfigmap* : Pour utiliser les ConfigMap
- ...

Strimzi est également une solution simple pour déployer Kafka dans Openshift/Kubernetes.



Vérifications de l'installation

Création de topic :

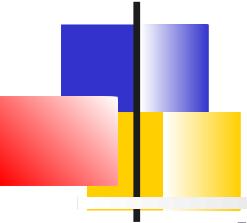
```
bin/kafka-topics.sh --create --bootstrap-server  
localhost:9092 --replication-factor 1 --partitions 1 --  
topic test
```

Envois de messages :

```
bin/kafka-console-producer.sh --bootstrap-server  
localhost:9092 --topic test  
This is a message  
This is another message  
^D
```

Consommation de messages :

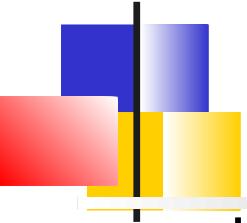
```
bin/kafka-console-consumer.sh --bootstrap-server  
localhost:9092 --topic test --from-beginning  
This is a message  
This is another message
```



Script d'arrêt

La distribution propose également un script d'arrêt permettant d'arrêter les process Kafka en cours d'exécution.

bin/kafka-server-stop.sh

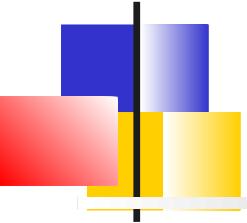


Fichiers de trace

La configuration des fichiers de trace est définie dans
conf/log4j.properties

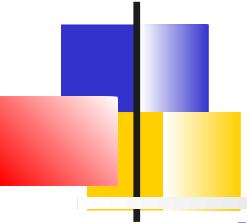
Par défaut sont générés :

- ***server.log*** : Traces générales
- ***state-change.log*** : Traces des changements d'état
(topics, partition, brokers, répliques)
- ***kafka-request.log*** : Traces des requêtes clients et
inter-broker
- ***log-cleaner.log*** : Suppression des messages expirés
- ***controller.log*** : Logs du contrôleur
- ***kafka-authorizer.log*** : Traces sur les ACLs



Cluster Kafka

Cluster
Distributions / Installation
Utilitaires Kafka
Outils graphiques

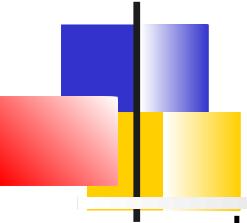


Introduction

Le répertoire ***\$KAFKA_HOME/bin*** contient de nombreux scripts utilitaires dédiés à l'exploitation du cluster.

Les scripts utilisent l'API Java de Kafka

Une aide est disponible pour chaque script décrivant les paramètres requis ou optionnels



Gestion des topics

Le script ***bin/kafka-topics.sh*** permet de créer, supprimer, modifier ou visualiser un topic

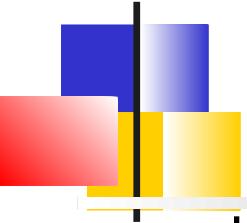
Les valeurs par défaut des topics sont définis dans *server.properties* mais peuvent être surchargées pour chaque topic

Exemple création de topic :

```
bin/kafka-topics.sh --create \  
  --bootstrap-server localhost:9092 \  
  --replication-factor 1 --partitions 13 \  
  --topic my-topic
```

Exemple listing des topics :

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

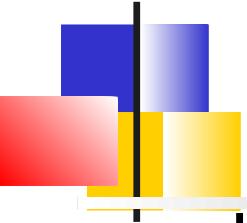


Choix du nombre de partitions

Une fois un *topic* créé, on ne peut pas diminuer son nombre de partitions. Une augmentation est possible mais peut générer des problèmes.

Lors du choix, il faut tenir compte de :

- Du débit des écritures
- Le débit maximum de consommation
- L'espace disque et la bande passante réseau sur chaque broker
- En général en augmentant le nombre de partitions, on augmente le débit global de l'application en s'autorisant plus de consommateurs
- Mais évitez de surestimer, car chaque partition utilise de la mémoire et le nombre de partitions augmente le temps du processus d'élection du maître



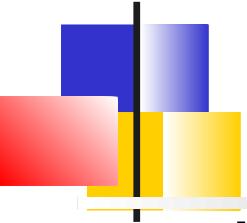
Envoi de message

Le script ***bin/kafka-console-producer.sh*** permet de tester l'envoi des messages sur un topic

Exemple :

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

...Puis saisir les messages sur l'entrée standard

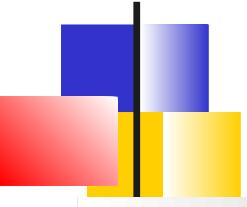


Lecture de message

Le script ***bin/kafka-console-consumer.sh*** permet de consommer les messages d'un topic

Exemple lecture depuis l'origine :

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



Autre utilitaires

kafka-consumer-groups.sh permet de gérer les groupes de consommateurs : les lister et manipuler leurs offsets

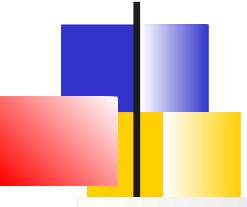
kafka-reassign-partitions.sh permet de gérer les partitions, déplacement sur de nouveau brokers, de nouveaux répertoire, extension de cluster, ...

kafka-replica-verification.sh : Vérifie

kafka-log-dirs.sh : Obtient les informations de configuration des log.dirs du cluster

kafka-dump-log.sh permet de parser un fichier de log et d'afficher les informations utiles pour debugger

kafka-delete-records.sh Permet de supprimer des messages jusqu'à un offset particulier



Autre utilitaires (2)

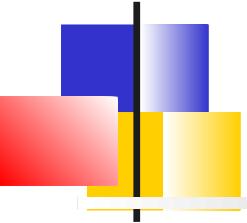
kafka-configs.sh permet des mises à jour de configuration dynamique des brokers

kafka-cluster.sh obtenir l'ID du cluster et sortir un broker du cluster

kafka-metadata-quorum.sh permet d'afficher les informations sur les contrôleurs

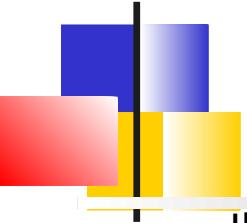
kafka-acls.sh permet de gérer les permissions sur les topics

kafka-verifiable-consumer.sh, **kafka-verifiable-producer.sh**: Utilitaires permettant de produire ou consommer des messages et de les afficher sur la console au format JSON



Cluster Kafka

Cluster
Distributions / Installation
Utilitaires Kafka
Outils graphiques



Outils graphiques

Il peut être intéressant de s'équiper d'outils graphiques aidant à l'exploitation du cluster.

Comme produit gratuit, semi-commerciaux citons :

- **Akhq** (<https://akhq.io/>)
- **Kafka Magic** (<https://www.kafkamagic.com/>)
- **Redpanda Console** (<https://docs.redpanda.com/docs/manage/console/>)

Ils proposent une interface graphique permettant :

- De rechercher et afficher des messages,
- Transformer et déplacer des messages entre des topics
- Gérer les topics
- Automatiser des tâches.

La distribution commerciale de Confluent a naturellement un outil graphique d'exploitation

Kafka Magic

The screenshot shows the Kafka Magic interface for Apache Kafka. The left sidebar has a purple header with the title "Kafka Magic for Apache Kafka®". It includes a "Cluster Explorer" button and three icons: a refresh, settings, and a cluster icon. The main menu is expanded to show "Kafka Clusters" and "Local". Under "Local", there are sections for "Topics" (with a "find..." search bar) and "Partitions" (with a "Partition 0" entry). On the right, the main content area has a purple header with "Register New" and "Pro Features" / "Feedback" buttons. The main content displays "Registered connections to Kafka clusters" with a table:

Name	Bootstrap Servers	Schema Registry Url
Local	localhost:9092	Not configured

At the bottom, there are pagination controls: "Items per page: 10", "1 – 1 of 1", and navigation arrows.

Page footer: © 2019 - 2023 Digitsy, Inc. Community edition v.4.0.1.138

akhq



0.24.0

my-cluster

Nodes

Topics

Live Tail

Consumer Groups

ACLS

Schema Registry

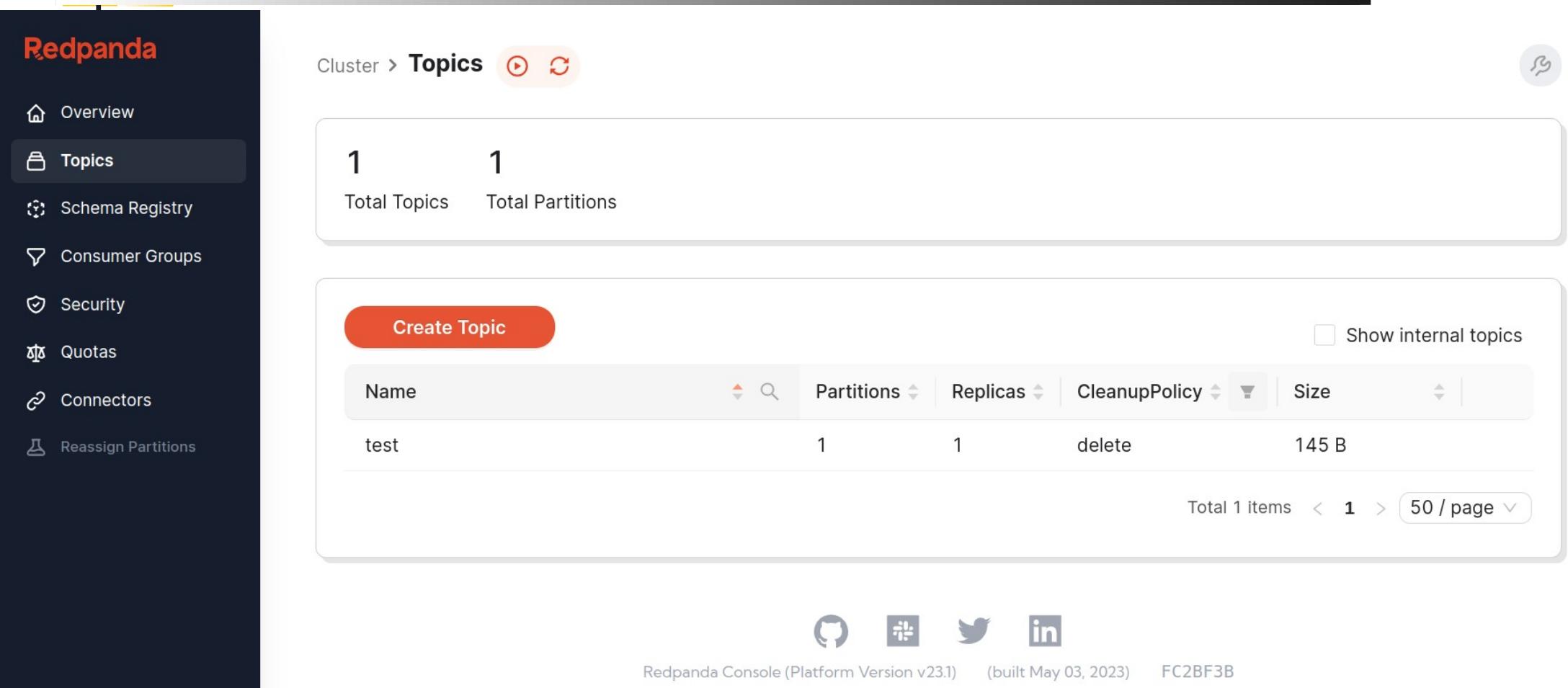
Settings

Nodes

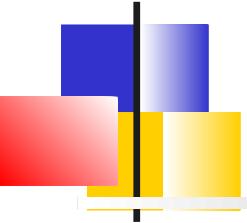
default

ID	Host	Controller	Partitions (% of total)	Rack
1	localhost:9092	False	1 (100.00%)	
2	localhost:9192	True	4 (100.00%)	
3	localhost:9292	False	4 (100.00%)	

Redpanda Console

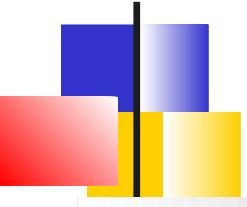


The screenshot shows the Redpanda Console interface. The left sidebar has a dark theme with the Redpanda logo at the top. The 'Topics' item is selected, highlighted with a dark background and white text. Below the sidebar, the main content area has a light gray background. At the top of the content area, it says 'Cluster > Topics' with two small circular icons (refresh and search). In the center, there's a summary box with '1 Total Topics' and '1 Total Partitions'. Below this, a 'Create Topic' button is visible. A table lists one topic: 'test'. The table columns are 'Name', 'Partitions', 'Replicas', 'CleanupPolicy', and 'Size'. The 'Name' column shows 'test', 'Partitions' shows '1', 'Replicas' shows '1', 'CleanupPolicy' shows 'delete', and 'Size' shows '145 B'. To the right of the table, there's a checkbox labeled 'Show internal topics'. At the bottom of the table, it says 'Total 1 items' and '50 / page'. At the very bottom of the page, there are social media sharing icons for GitHub, LinkedIn, Twitter, and a red square icon. The footer contains the text 'Redpanda Console (Platform Version v23.1) (built May 03, 2023) FC2BF3B'.



APIs

Producer API
Consumer API
Schema Registry
Connect API
Admin et Stream API

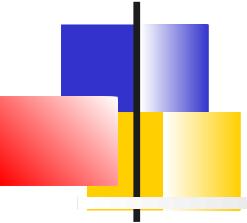


Introduction

L'API est simple mais les contraintes applicatives influencent la façon de l'utiliser ainsi que la configuration des *topics*

Les questions devant être posées :

- Chaque message est-il critique, ou peut-on tolérer des pertes de messages?
- La duplication accidentelle de messages est-elle autorisée ?
- Y-a-t-il des exigences strictes de latence ou de débit ?



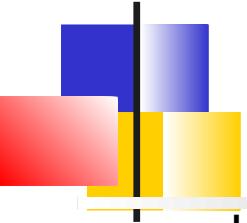
Dépendances

Java

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>${kafka-version}</version>
</dependency>
```

Python

```
pip install confluent-kafka
```

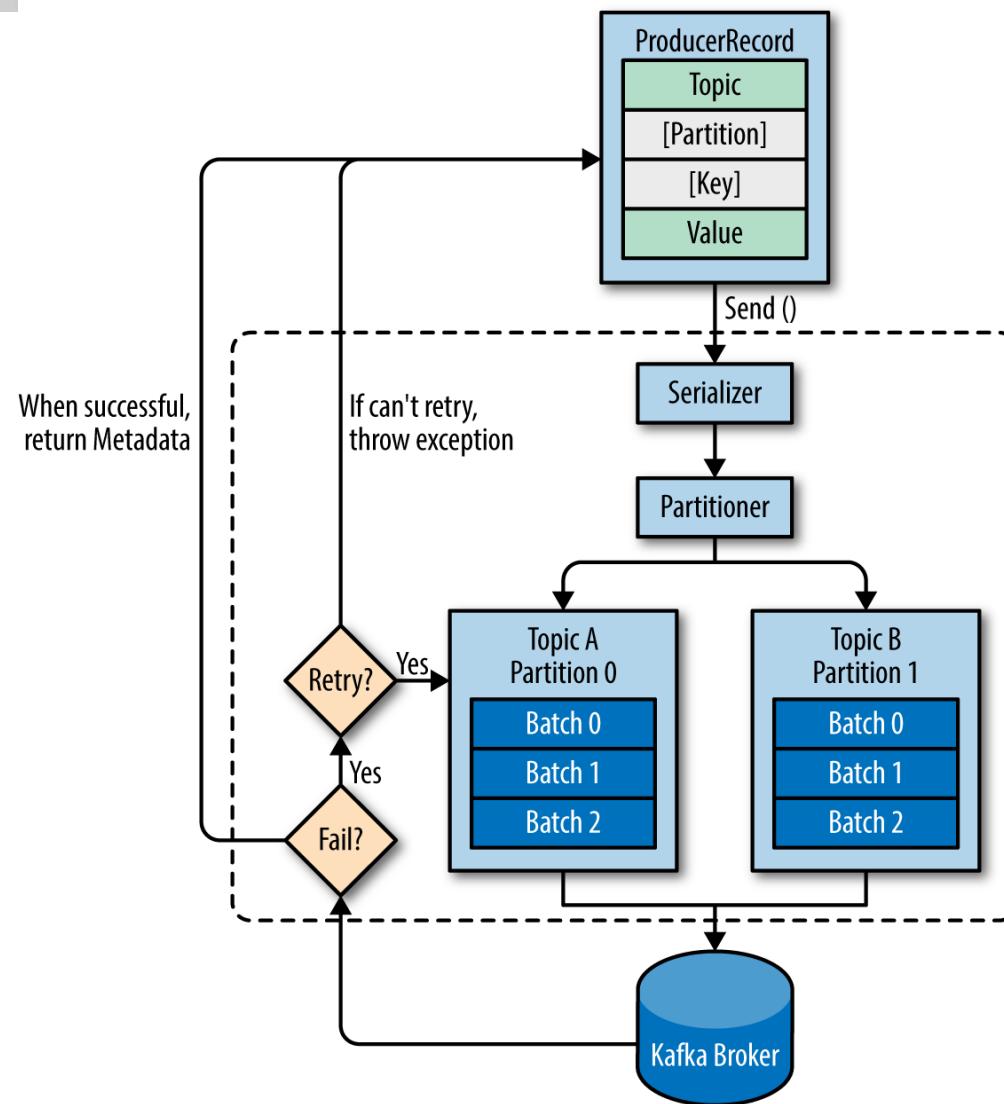


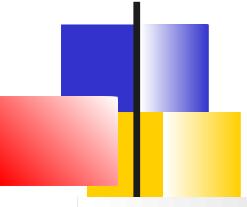
Étapes lors de l'envoi d'un message

L'envoi de message est constitué de plusieurs étapes :

- Création d'un objet ***ProductRecord*** encapsulant le topic et optionnellement une clé et une partition
- L'objet est **sérialisé** pour préparer sa transmission sur le réseau
- Les données sont ensuite fournies à un **partitionneur** qui détermine la partition de destination, (à partir de la partition indiquée, de la clé du message ou en Round-robin)
- Une fois la partition sélectionnée, le message est ajouté à un **lot de messages** destiné à la même partition. Une thread séparée envoie le lot de messages.
- Lorsque le broker reçoit le message, il renvoie une réponse sous la forme d'un objet ***RecordMetadata*** encapsulant le *topic*, la partition, la clé et l'offset
- Si le broker n'arrive pas à écrire le message dans le journal, il renvoie une erreur et le producteur peut réessayer un certain nombre de fois

Envoi de message





Construire un Producteur

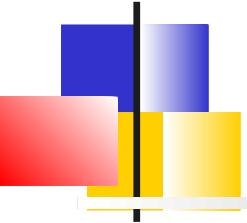
La première étape pour l'envoi consiste à instancier un **KafkaProducer**.

3 propriétés de configurations sont obligatoires :

- **bootstrap.servers** : Liste de brokers que le producteur contacte au départ pour découvrir le cluster
- **key.serializer** : La classe utilisée pour la sérialisation de la clé
- **value.serializer** : La classe utilisée pour la sérialisation du message ...

1 optionnelle est généralement positionnée :

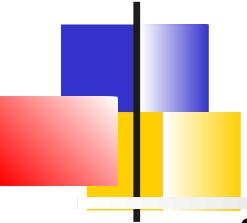
- **client.id** : Permet le suivi des messages



Exemple Java

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
    "broker1:9092,broker2:9092");
kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

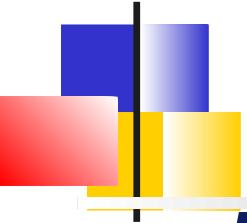
producer = new KafkaProducer<String, String>(kafkaProps);
```



Exemple python

```
from confluent_kafka import Producer  
  
import socket  
  
conf = {'bootstrap.servers':  
        "host1:9092,host2:9092"}  
  
producer = Producer(conf)
```

Rq : Serialiseurs sont fournies « out-of the box »



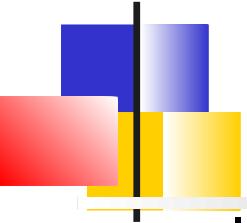
ProducerRecord

ProducerRecord représente l'enregistrement à envoyer à Kafka.

Il contient le nom du *topic*, une valeur et éventuellement une clé, une partition, un timestamp

Constructeurs :

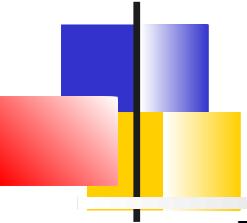
```
# Sans clé
ProducerRecord(String topic, V value)
# Avec clé
ProducerRecord(String topic, K key, V value)
# Avec clé et partition
ProducerRecord(String topic, Integer partition, K key, V value)
# Avec clé, partition et timestamp
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)
```



Méthodes d'envoi des messages

Il y 3 façons d'envoyer des messages :

- **Fire-and-forget** : Pas d'acquittement, on s'autorise à perdre quelques messages (même si c'est très rare)
- **Envoi synchrone** : La méthode renvoie un objet *Future* sur lequel on appelle la méthode *get()* pour attendre la réponse.
On traite éventuellement les cas d'erreurs
- **Envoi asynchrone** : Lors de l'envoi, on passe en argument une fonction de call-back.
La méthode est appelée lorsque la réponse est retournée



Fire And Forget

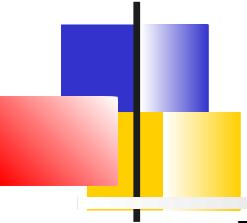
Java

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Precision", "France");
```

```
try {  
    producer.send(record);  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Python

```
producer.produce(topic, key="key", value="value")
```



Envoi synchrone

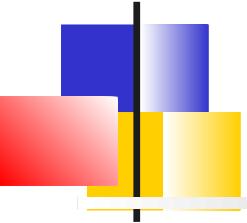
Java

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Precision", "France");
```

```
try {  
    producer.send(record).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Python

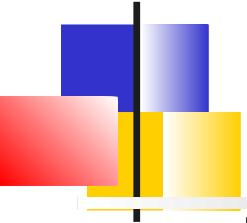
```
producer.produce(topic, key="key", value="value")  
producer.flush()
```



Envoi asynchrone avec call-back (Java)

```
private class DemoProducerCallback implements Callback {  
    @Override  
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {  
        if (e != null) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");  
producer.send(record, new DemoProducerCallback());
```

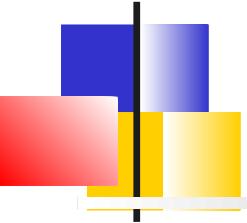


Envoi asynchrone avec call-back (Python)

```
def acked(err, msg):
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err)))
    else:
        print("Message produced: %s" % (str(msg)))

producer.produce(topic, key="key", value="value", callback=acked)

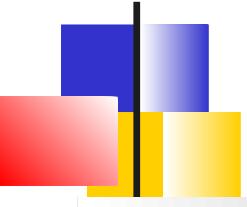
# Wait up to 1 second for events. Callbacks will be invoked during
# this method call if the message is acknowledged.
producer.poll(1)
```



Méthode flush

Sur le producer la méthode **flush()** permet de vider le batch des messages en attente d'émission.

Elle peut être utilisée pour s'assurer que tous les messages envoyés par send() sont vraiment parvenus au broker

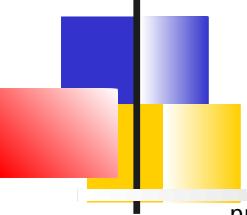


Sérialiseurs

La librairie *Kafka* inclut des (dé)sérialiseurs pour les types primitifs *string*, *integer* etc.

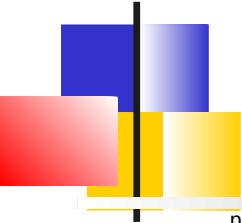
Pour des objets du domaine, il faut implémenter ses propres sérialiseurs/désérialiseurs en s'appuyant sur des librairies comme *Avro*, *Thrift*, *Protobuf* ou *Jackson*

Certains sérialiseurs permettent une gestion fine des évolutions des formats de messages via les ***Schema Registry***



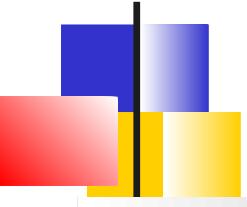
Exemple sérialiseur s'appuyant sur Jackson

```
public class JsonPOJO.Serializer<T> implements Serializer<T> {  
    private final ObjectMapper objectMapper = new ObjectMapper();  
    /**  
     * Default constructor requis par Kafka  
     */  
    public JsonPOJO.Serializer() { }  
  
    @Override  
    public void configure(Map<String, ?> props, boolean isKey) { }  
  
    @Override  
    public byte[] serialize(String topic, T data) {  
        if (data == null)  
            return null;  
  
        try {  
            return objectMapper.writeValueAsBytes(data);  
        } catch (Exception e) {  
            throw new SerializationException("Error serializing JSON message", e);  
        }  
    }  
  
    @Override  
    public void close() { }  
}
```



Exemple déserialiseur basé sur Jackson

```
public class JsonPOJODeserializer<T> implements Deserializer<T> {  
    private ObjectMapper objectMapper = new ObjectMapper();  
    private Class<T> tClass;  
    // Default constructor requis par Kafka  
    public JsonPOJODeserializer() {}  
  
    @Override  
    public void configure(Map<String, ?> props, boolean isKey) {  
        tClass = (Class<T>) props.get("JsonPOJOClass");  
    }  
  
    @Override  
    public T deserialize(String topic, byte[] bytes) {  
        if (bytes == null)  
            return null;  
        T data;  
        try {  
            data = objectMapper.readValue(bytes, tClass);  
        } catch (Exception e) {  
            throw new SerializationException(e);  
        }  
        return data;  
    }  
    @Override  
    public void close() {}  
}
```



Envoi de message

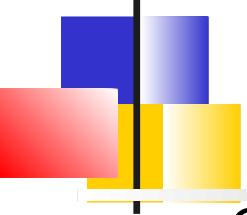
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("value.serializer", "org.myappli.JsonPOJO.Serializer");

String topic = "customerContacts";

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

Customer customer = CustomerGenerator.getNext();

ProducerRecord<String, Customer> record =
        new ProducerRecord<>(topic, customer.getId(), customer);
producer.send(record);
```

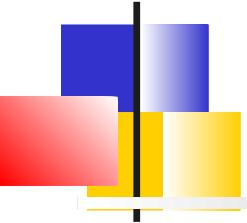


Configuration des producteurs fiabilité

Certains paramètres ont un impact significatif sur l'utilisation de la mémoire, les performances et la fiabilité des producteurs.

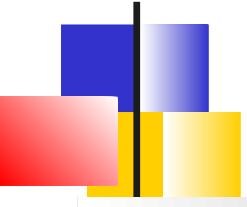
Fiabilité :

- **acks** : contrôle le nombre de réplicas qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie
- **retries** : Si l'erreur renvoyée est de type Retriable, le nombre de tentative de renvoi. Si > 0 possibilité d'envoi en doublon
- **max.in.flight.requests.per.connection** : Maximum de message en cours de transmission (sans réponse obtenu)
- **enable.idempotence** : Livraison unique de message
- **transactional.id** : Mode transactionnel



Configuration des producteurs *performance*

- **batch.size** : La taille du batch en mémoire pour envoyer les messages.
Défaut 16ko
- **linger.ms** : la durée d'attente de messages supplémentaires avant d'envoyer le batch courant. Défaut 0ms
- **buffer.memory** : Taille buffer pour stocker les messages que l'on ne peut pas envoyé. Défaut 32Mo
- **compression.type** : Par défaut, les messages ne sont pas compressés.
Valeurs possibles : *snappy* , *gzip* , ou *lz4*
- **request.timeout.ms**, **metadata.fetch.timeout.ms** et **timeout.ms**:
Timeouts pour la réception d'une réponse à un message, pour obtenir des méta-données (leader, etc..) pour obtenir le ack des répliques.
- **max.block.ms** : *Temps maximum d'attente pour la méthode send(). Dans le cas où le buffer est rempli*
- **max.request.size** : *Taille max d'un message*
- **receive.buffer.bytes** et **send.buffer.bytes**: *Taille des buffers TCP*



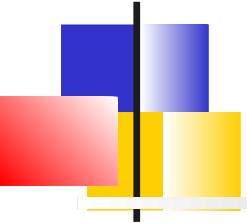
Garantie sur l'ordre

En absence de failure, *Kafka* préserve l'ordre des messages au sein d'une partition.

- Si des messages ont été envoyés par le producteur dans un ordre spécifique, le broker les écrit sur une partition dans cet ordre et tous les consommateurs les liront dans cet ordre ...

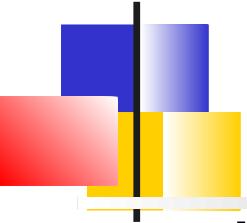
En cas de failure

- Si $\text{retries} > 0$ et $\text{max.in.flights.requests.per.session} > 1$. Il se peut que lors d'un renvoi l'ordre initial soit inversé.
- Pour avoir une garantie sur l'ordre avec tolérance aux fautes, on configure $\text{retries} > 0$ et $\text{max.in.flights.requests.per.session} = 1$ (au détriment du débit global)



APIs

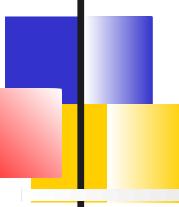
Producer API
Consumer API
Schema Registry
Connect API
Admin et Stream API



Introduction

Les applications qui ont besoin de lire les données de Kafka utilisent un **KafkaConsumer** pour s'abonner aux topics Kafka

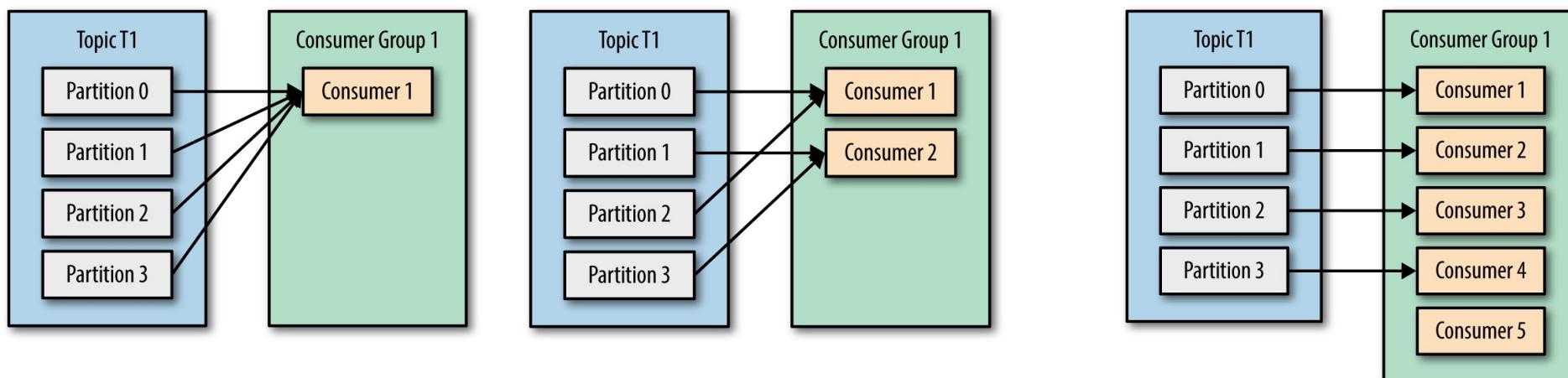
Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relations avec les partitions

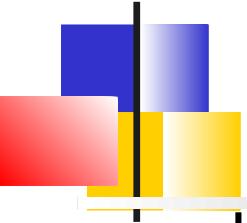


Groupes de consommateurs

Les consommateurs font généralement partie d'un **groupe de consommateurs**.

- Chaque consommateur d'un groupe reçoit les messages d'un sous-ensemble différent des partitions du topic.





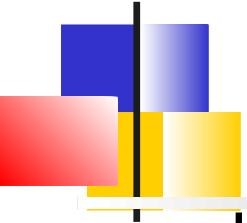
Rééquilibrage dynamique des consommateurs

Lors de l'ajout d'un nouveau consommateur, celui-ci peut se faire affecter une partition consommée précédemment par un autre consommateur du groupe.

Lors de l'arrêt d'un consommateur, la partition qui lui était assignée est réaffectée à un autre consommateur

Cette répartition dynamique offre la scalabilité et la tolérance aux défaillances mais n'est pas spécialement désirable

- car durant le rééquilibrage les messages ne sont pas consommés
- les consommateurs si ils utilisent des caches sont obligés de les rafraîchir

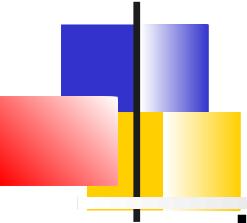


Membership et détection de pannes

Les consommateurs maintiennent leur appartenance à un groupe et leurs attributions de partitions en envoyant régulièrement des *heartbeat* à un broker coordinateur (qui peut être différent en fonction des groupes).

```
kafka-consumer-groups.sh --bootstrap-server  
localhost:9092 --group sample --describe --state
```

Si un consommateur cesse d'envoyer des *heartbeats*, sa session expire et le coordinateur démarre une réaffectation des partitions

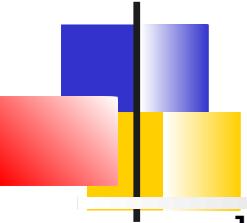


Création de *KafkaConsumer*

L'instanciation d'un ***KafkaConsumer*** est similaire à celle d'une *KafkaProducer*

Plusieurs propriétés doivent être spécifiées dans une classe *Properties* :

- *bootstrap.servers*
- *key.deserializer* , et *value.deserializer*
- *group.id* qui spécifie le groupe de consommateur



Exemple

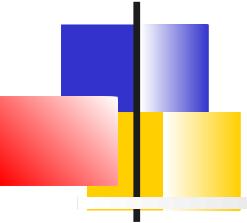
Java

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
    "broker1:9092,broker2:9092");
kafkaProps.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
kafkaProps.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
kafkaProps.put("group.id", "myGroup");

consumer = new KafkaConsumer<String, String>(kafkaProps);
```

Python

```
from confluent_kafka import Consumer
conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': 'foo', 'auto.offset.reset': 'smallest'}
consumer = Consumer(conf)
```



Abonnement à un *topic*

Après la création d'un consommateur, il faut souscrire à un *topic*

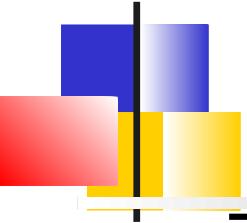
La méthode ***subscribe()*** prend une liste de *topics* comme paramètre.

Ex :

```
consumer.subscribe(Collections.singletonList("myTopic"));
```

Il est également possible d'utiliser *subscribe()* avec une expression régulière

```
consumer.subscribe("test.*");
```

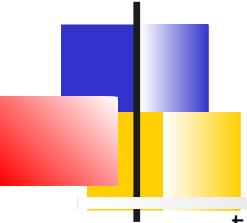


Boucle de Polling

Typiquement, les consommateurs *poll* continuellement les *topics* auxquels ils sont abonnés.

Les objets retournés par *poll* sont une collection de ***ConsumerRecord*** qui encapsule :

- Le message :
- La partition
- L'offset
- Le timestamp

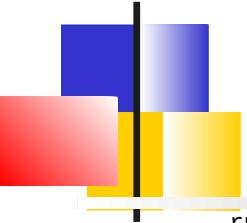


Exemple Java

```
try {
    while (true) {
        // poll envoie le heartbeat, on bloque pdt 100ms pour récupérer les messages
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            log.debug("topic = %s, partition = %s, offset = %d,
                      customer = %s, country = %s\n",
                      record.topic(), record.partition(),
                      record.offset(), record.key(), record.value());

            int updatedCount = 1;
            if (custCountryMap.containsValue(record.value())) {
                updatedCount = custCountryMap.get(record.value()) + 1;
            }
            custCountryMap.put(record.value(), updatedCount) ;

            System.out.println(custCountryMap) ;
        }
    }
} finally {
    consumer.close();
}
```



Exemple Python

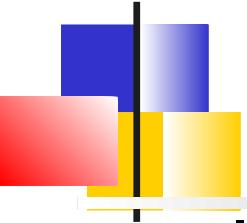
```
running = True

def basic_consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
    finally:
        # Close down consumer to commit final offsets and trigger rebalance.
        consumer.close()

def shutdown():
    running = False
```

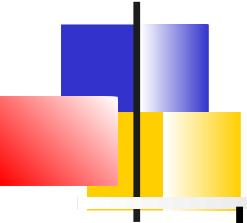


Thread et consommateur

Il n'est pas possible d'avoir plusieurs consommateurs du même groupe dans la même thread et il n'est pas possible d'utiliser le même consommateur dans plusieurs threads

=> 1 consommateur = 1 thread

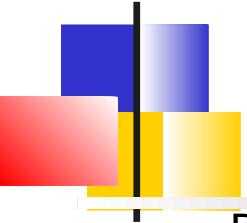
Pour exécuter plusieurs consommateurs d'un même groupe dans une application, il est utile d'utiliser les classes *ExecutorService* de Java.



Configuration des consommateurs

Les propriétés les plus importantes :

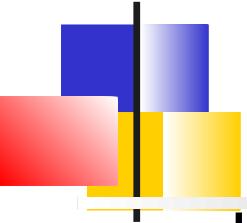
- ***auto.offset.reset*** : *latest* (défaut) ou *earliest*.
Contrôle le comportement du consommateur si il ne détient pas d'offset valide.
Dernier message ou le plus ancien
- ***enable.auto.commit*** : Le consommateur commit les offsets automatiquement ou non.
Par défaut true
Si true :
 - ***auto.commit.interval.ms*** : Intervalle d'envois des commits



Configuration des consommateurs (2)

D'autres propriétés

- **fetch.min.bytes** : Volume minimum de données à recevoir. Permet de réduire la charge sur le broker et le consommateur
- **fetch.max.wait.ms** : Attente maximale avant de récupérer les données
- **max.partition.fetch.bytes** : Maximum de données par partition ramenées lors d'un poll. Par défaut 1Mo
- **max.poll.records** : Maximum de record via un *poll()*
- **session.timeout.ms** : Le temps faisant expirer la session et déclarer le consommateur comme down. Par défaut 10s
- **heartbeat.interval.ms** : L'intervalle d'envoi des heartbeat. Par défaut 3s
- **partition.assignment.strategy** : Stratégie d'affectation des partitions *Range* (défaut), *RoundRobin* ou *Custom*
- **client.id** : Une chaîne de caractère utilisé pour les métriques.
- **receive.buffer.bytes** et **send.buffer.bytes** : Taille des buffers TCP



Offsets et Commits

Les consommateurs peuvent suivre leurs offsets de partition en s'adressant à Kafka.

Kafka appelle la mise à jour d'un offset : un **commit**

Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka :

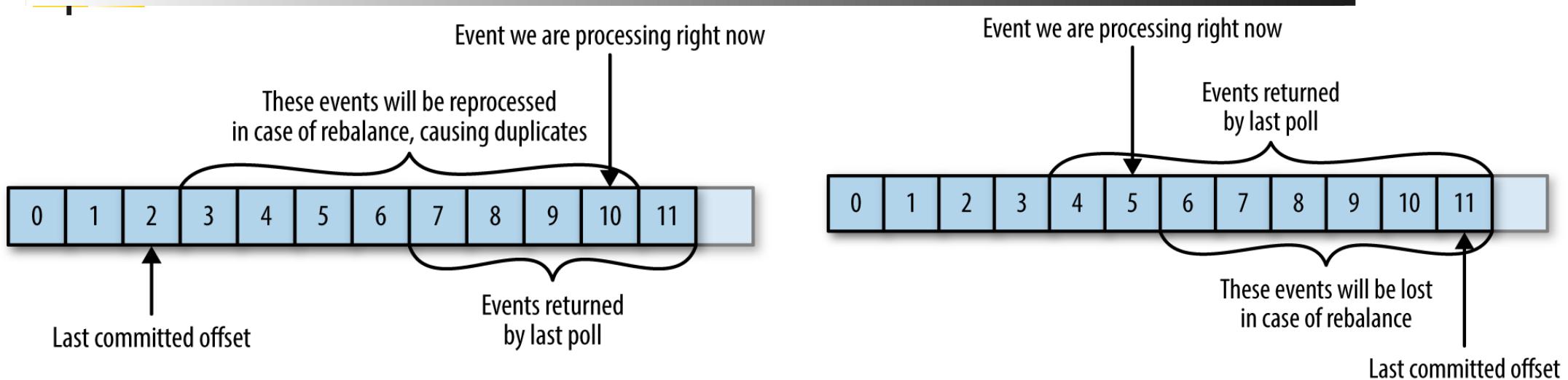
consumer_offsets

- Ce *topic* contient les offsets de chaque partition.

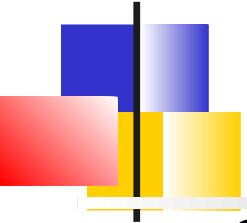
Lors d'une réaffectation de partitions, 2 risques doivent être pris en compte :

- Traiter 2 fois le même message
- Louper des messages

Risques lors d'une réaffectation



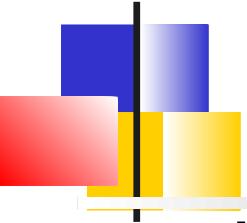
Kafka propose plusieurs façons de gérer les commits



Commit automatique

Si **`enable.auto.commit=true`** ,
le consommateur valide toutes les
`auto.commit.interval.ms` (par défaut
5000), les plus grands offset reçus par *poll()*

=> Cette approche (simple) ne protège pas contre les duplications en cas de ré-affectation
Cela dépend de la grandeur de l'intervalle en fonction du temps de traitement des messages ramenés par *poll()*

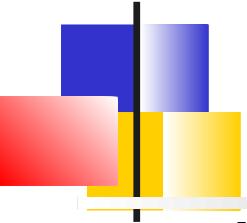


Commit contrôlé

L'API Consumer permet de contrôler le moment du commit plutôt que de se baser sur un timer.

Si ***auto.commit.offset=false*** ,
l'application doit explicitement committer les offsets

- Soit de façon bloquante avec ***commitSync()***
- Soit en asynchrone avec ***commitAsync()***

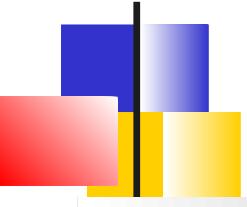


Commit Synchrone

La méthode ***commitSync()*** valide les derniers offsets reçus par *poll()*

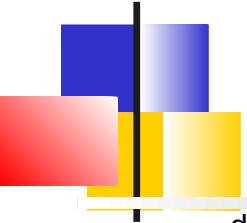
- La méthode est bloquante et retourne lorsque les offsets sont commités
- Elle lance une exception si un commit échoue

=> En cas de réaffectation, il y a toujours un risque pour que le messages soit traités plusieurs fois



Exemple Java

```
while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        // Si réaffectation, possibilité de duplication
        log.info("topic = %s, partition = %s,
                  offset =%d, customer = %s, country = %s\n",
                  record.topic(), record.partition(),
                  record.offset(), record.key(), record.value());
    }
    try {
        consumer.commitSync();
    } catch (CommitFailedException e) {
        log.error("commit failed", e)
    }
}
```

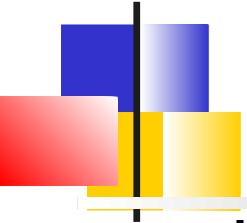


Exemple Python

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

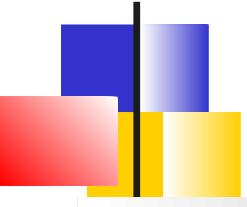
            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(asynchronous=False)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```



Commit asynchrone

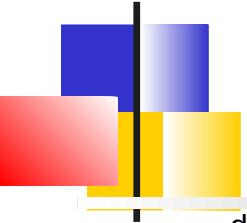
La méthode ***commitAsync()*** est non bloquante

- En cas d'erreur, 2 cas de figure en fonction de la configuration de *retry* :
 - Soit *retry* et erreur de type *Retriable*, en fonction de la configuration, la méthode peut effectuer un renvoi.
Attention, cela peut provoquer des ordres de commits dans le mauvais ordre
 - Si pas de *retry*, alors c'est le prochain appel à commit qui validera les offsets
- Il est possible de fournir une méthode de callback en argument



Exemple Java

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        Log.info("topic = %s, partition = %s,
                  offset = %d, customer = %s, country = %s\n",
                  record.topic(), record.partition(), record.offset(),
                  record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
                               OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
    });
}
```

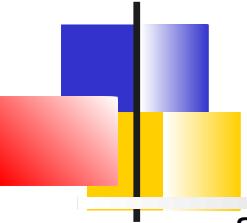


Exemple Python

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(asynchronous=True)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```



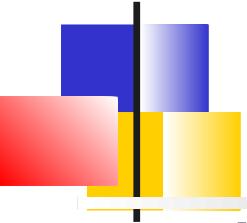
Python Commit avec callback

```
from confluent_kafka import Consumer

def commit_completed(err, partitions):
    if err:
        print(str(err))
    else:
        print("Committed partition offsets: " + str(partitions))

conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': "foo",
        'default.topic.config': {'auto.offset.reset': 'smallest'},
        'on_commit': commit_completed}

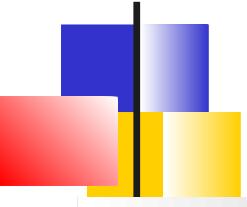
consumer = Consumer(conf)
```



Committer un offset spécifique

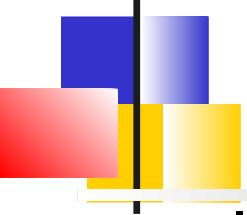
L'API permet également de fournir en argument une *Map* contenant les offsets que l'on veut valider pour chaque partition

- Cela permet de committer sans avoir traité l'intégralité des messages ramenés par *poll()*



Exemple

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new
    HashMap<>();
int count = 0;
...
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        log.info("topic = %s, partition = %s, offset = %d,
                  customer = %s, country = %s\n",
                  record.topic(), record.partition(), record.offset(),
                  record.key(), record.value());
        currentOffsets.put(new
TopicPartition(record.topic(),record.partition()), new
OffsetAndMetadata(record.offset()+1, "no metadata"));
        if (count % 1000 == 0)
            consumer.commitAsync(currentOffsets, null);
        count++;
    }
}
```

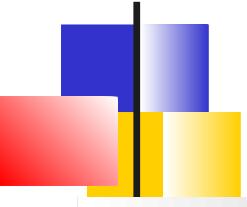


Stocker les offsets hors de Kafka

Il n'y a pas d'obligation à stocker les offsets dans Kafka, un consommateur peut stocker lui même les offsets dans son propre data store.

Si les offsets et le résultat du traitement sont stockés dans la même BD, on peut alors profiter d'une transaction et d'une écriture atomique.

Ce type de scénario permet d'obtenir très facilement des garanties de livraison « *Exactly Once* ».

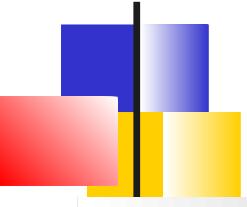


Réagir aux réaffectations

Lors de réaffectations de partitions, les consommateurs peuvent être prévenus afin de prendre les mesures adéquates (commit, fermeture de ressources, ...)

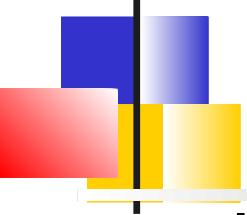
Lors du *subscribe()*, il faut fournir une classe de type ***ConsumerRebalanceListener*** qui implémente 2 méthodes :

- public void onPartitionsRevoked(
 Collection<TopicPartition> partitions)
- public void onPartitionsAssigned(
 Collection<TopicPartition> partitions)



Exemple

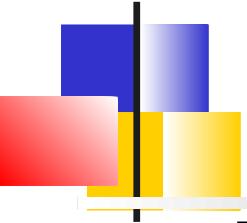
```
private class HandleRebalance implements  
ConsumerRebalanceListener {  
  
    public void onPartitionsAssigned(  
        Collection<TopicPartition> partitions) { }  
  
    public void onPartitionsRevoked(  
        Collection<TopicPartition> partitions) {  
        log.info("Lost partitions in rebalance.  
            Committing current offsets:" + currentOffsets);  
        consumer.commitSync(currentOffsets);  
    }  
}
```



Consommation de messages avec des offsets spécifiques

L'API permet d'indiquer un offset spécifique :

- ***seekToBeginning(TopicPartition tp)*** :
Revenir au début de la partition
- ***seekToEnd(TopicPartition tp)*** :
Se placer à la fin
- ***seek(TopicPartition, long)*** :
Se placer à un offset particulier
Cela permet de stocker les offsets en dehors de Kafka et en cas d'erreur ou redémarrage repartir à partir des offsets sauvegardés

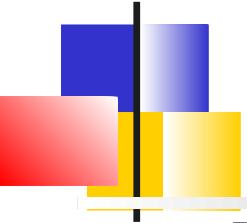


Sortie de boucle

Pour sortir de la boucle de *poll*, il faut qu'une autre thread appelle ***consumer.wakeup()*** qui a pour effet de lancer une *WakeupException* lors de l'appel à *poll*.

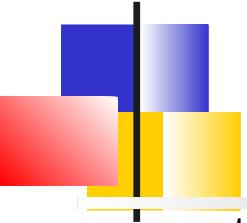
Le consommateur doit alors faire un appel explicite à *close()*

On peut utiliser
Runtime.addShutdownHook(Thread hook)



Exemple

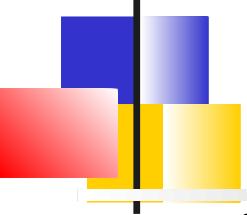
```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {
        System.out.println("Starting exit...");
        consumer.wakeup();
        try {
            mainThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
```



Exemple (2)

```
try {
    // Boucle infinie interruptible via ctrl-c
    while (true) {
        ConsumerRecords<String, String> records =
        movingAvg.consumer.poll(1000);
        log.info(System.currentTimeMillis() + "-- waiting for data...");  

        for (ConsumerRecord<String, String> record : records) {
            log.info("offset = %d, key = %s,value = %s\n",
                    record.offset(), record.key(), record.value());
        }
        for (TopicPartition tp: consumer.assignment())
            log.info("Committing offset
atposition:"+consumer.position(tp));
        movingAvg.consumer.commitSync();
    }
} catch (WakeupException e) {
} finally {
    consumer.close();
    log.info("Closed consumer and we are done");
}
```

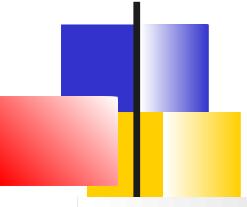


Affectation statique des partitions

Si l'application ne comporte que des consommateurs dédiés statiquement à des partitions, alors la fonctionnalité de groupe n'est pas nécessaire.

Dans ce cas, l'application assigne explicitement les partitions à ses consommateurs.

L'API ***assign()*** est alors utilisée à la place de *subscribe()*

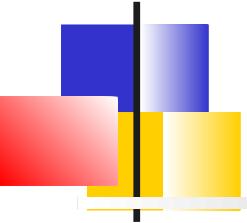


Exemple

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");

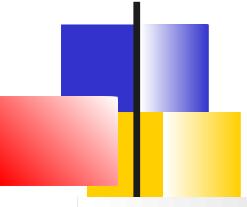
if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(), partition.partition()));

consumer.assign(partitions);
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record: records) {
        log.info("topic = %s, partition = %s, offset = %d,
                 customer = %s, country = %s\n",
                 record.topic(), record.partition(), record.offset(),
                 record.key(), record.value());
    }
    consumer.commitSync();
}
}
```



APIs

Producer API
Consumer API
Schema Registry
Connect API
Admin et Stream API

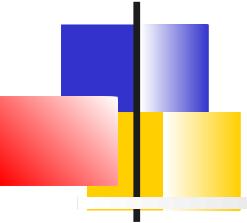


Introduction

Lors d'évolution des applications, le format des messages est susceptible de changer.

=> Afin de s'assurer que ses évolutions ne génèrent pas de problème chez les consommateurs, il est nécessaire d'utiliser un gestionnaire de schéma capable de détecter les problèmes de compatibilité.

C'est le rôle de ***Schema Confluent Registry*** qui supporte les formats de sérialisation JSON, Avro et ProtoBuf



Apache Avro

Apache Avro est un système de sérialisation de données.

- Il utilise une structure JSON pour définir le **schéma**, permettant la sérialisation entre les octets et les données structurées.
- Les outils associés à Avro sont capables de générer les classes Java¹ correspondantes au schéma.

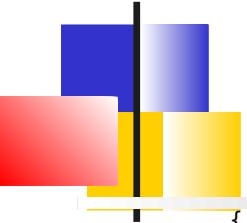
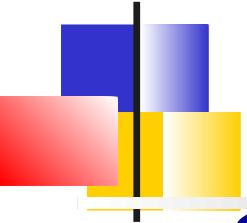


Schéma Avro

```
{  
  "type": "record",  
  "name": "Courier",  
  "namespace": "org.formation.model",  
  "fields": [  
    {  
      "name": "id",  
      "type": "int" },  
    {  
      "name": "firstName",  
      "type": "string" },  
    {  
      "name": "lastName",  
      "type": "string" },  
    {  
      "name": "position",  
      "type": [  
        {  
          "type": "record",  
          "name": "Position",  
          "namespace": "org.formation.model",  
          "fields": [  
            {  
              "name": "latitude",  
              "type": "double" },  
            {  
              "name": "longitude",  
              "type": "double" } ] ] ] } ] }
```



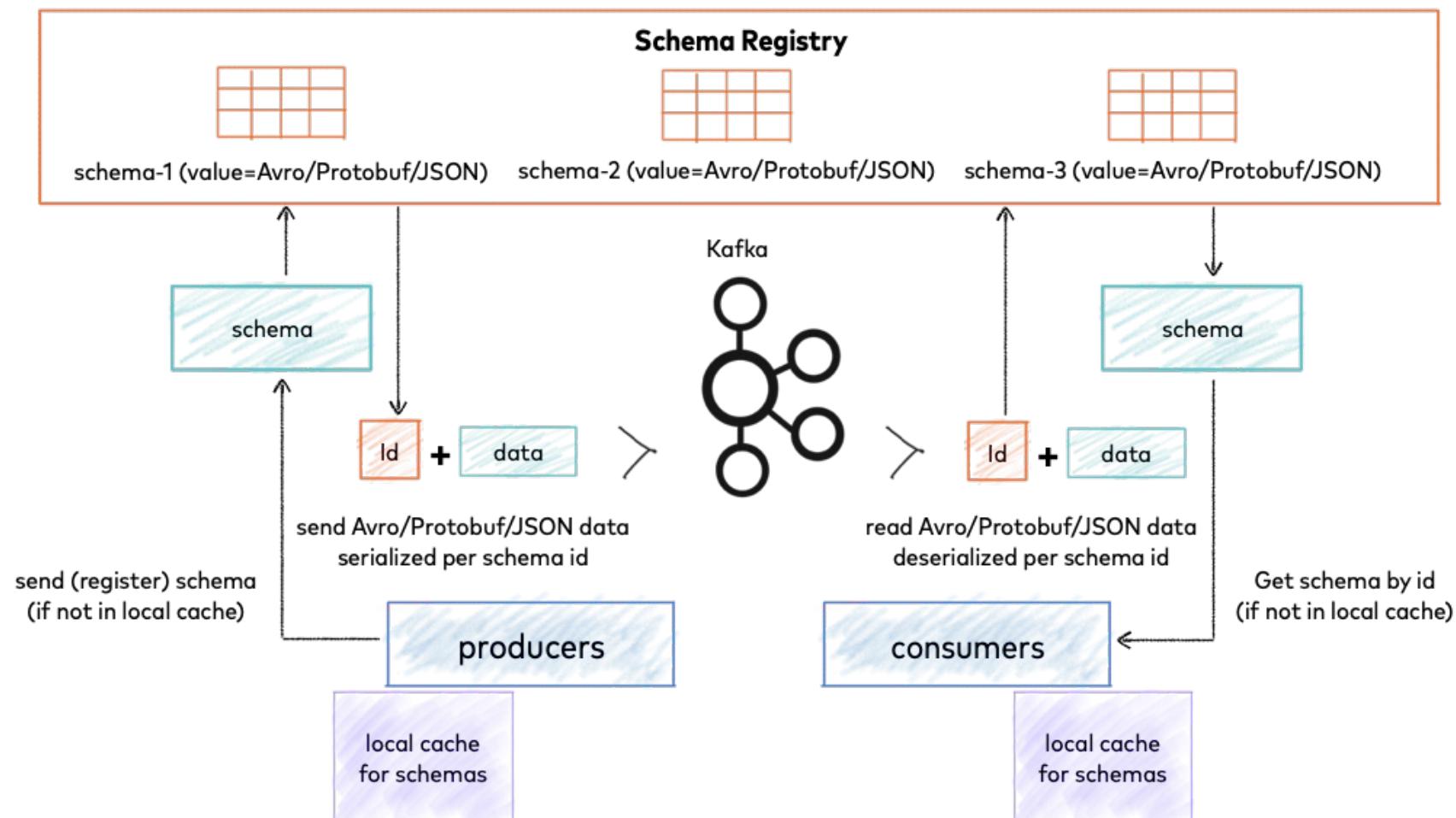
Utilisation du schéma

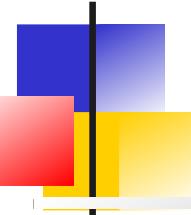
Confluent Schema Registry offre une API Rest

- Permettant de configurer le mode de compatibilité
- Permettant de stocker des Schema
- De détecter les incompatibilités entre schémas
=> Si le schéma est incompatible, le producteur est empêché de produire vers le topic
Le nouveau format de message devra être publié vers un autre topic

Les sérialiseurs inclut l'id du schéma dans les messages et font appel à l'API

Schema Registry



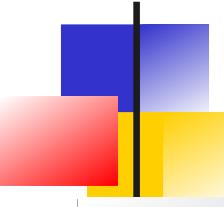


Modes de compatibilité

Trois niveaux de compatibilité sont pris en charge par *Schema Registry* :

- **Compatibilité backward** : Tous les messages de la version précédente du schéma sont également valides selon la nouvelle version
- **Compatibilité forward** : Tous les messages de la nouvelle version sont également valides selon la version précédente du schéma
- **Compatibilité full** : La version précédente du schéma et la nouvelle version sont toutes deux compatibles ascendante et descendante

Les vérifications de compatibilité s'effectuent soit juste avec la précédente version, soit avec toutes les versions précédentes.

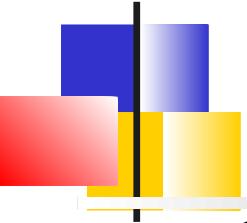


Choix du mode

La compatibilité backward, configuration par défaut, est à privilégier.

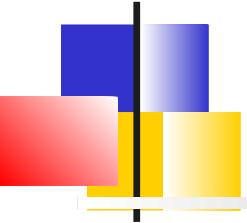
- Elle permet à un consommateur de lire les anciens messages, ce qui est compatible avec la possibilité de Kafka de s'abonner à posteriori.
- Cependant, cela oblige à upgrader les consommateurs en premier afin qu'il s'adapte à la nouvelle version du message.

Avec la compatibilité forward, ce sont les producteurs qui sont mis à jour en premier. Par contre, les possibilités de mises à jour sont très restrictives pour le producteur.



Dépendances Confluent

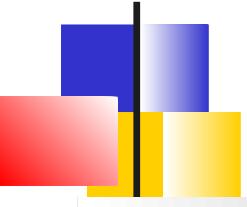
```
<dependencies>
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-schema-registry</artifactId>
    <version>7.2.1</version>
  </dependency>
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.2.1</version>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
  </repository>
</repositories>
```



Maven plug-in Avro

```
<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
    <version>1.8.2</version>
    <executions>
        <execution>
            <id>schemas</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>schema</goal>
                <goal>protocol</goal>
                <goal>idl-protocol</goal>
            </goals>
            <configuration>
                <sourceDirectory>./src/main/resources/</sourceDirectory>
                <outputDirectory>./src/main/java/</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

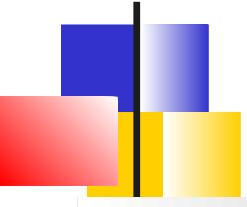
=> mvn compile génère les classes du modèle



Producteur (1)

Le producteur de message peut contenir du code permettant d'enregistrer le schéma en utilisant la librairie cliente de *Schema Registry*

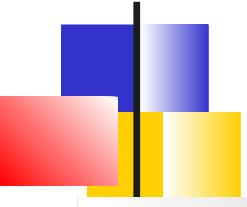
```
// Lecture du schéma
Schema avroSchema = new Schema.Parser().parse(inputStream);
// Instanciation du client
CachedSchemaRegistryClient client = new
    CachedSchemaRegistryClient(REGISTRY_URL, 20);
// Enregistrement du sujet, le nom correspond au nom du topic + suffixe
client.register(topicName +"-value", new AvroSchema(avroSchema);
```



Producteur (2)

Les propriétés du *KafkaProducer* doivent contenir :

- ***schema.registry.url*** :
L'adresse du serveur de registry
- Le sérialiseur de valeur :
io.confluent.kafka.serializers.KafkaAvroSerializer

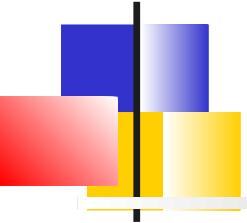


Consommateur

KafkaConsumer doit également préciser l'URL et le désérialiseur à
io.confluent.kafka.serializers.KafkaAvroDeserializer

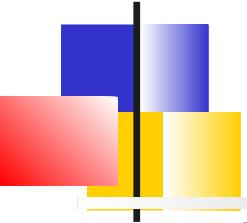
Il peut récupérer les messages sous la forme de ***GenericRecord*** plutôt que des classes spécialisés.

```
ConsumerRecords<String, GenericRecord> records =  
    consumer.poll(Duration.ofMillis(sleep));  
for (ConsumerRecord<String, GenericRecord> record : records)  
{  
    System.out.println("Value is " + record.value());
```



APIs

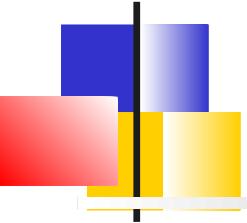
Producer API
Consumer API
Schema Registry
Connect API
Admin et Stream API



Introduction

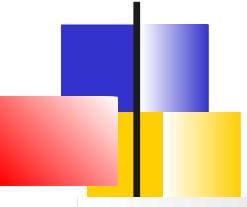
Kafka Connect permet d'intégrer Apache Kafka avec d'autres systèmes en utilisant des connecteurs.

- => Ingérer des bases de données volumineuses, des données de monitoring dans des topics avec des latences minimales
- => Exporter des topics vers des supports persistants



Fonctionnalités

- Un cadre commun pour les connecteurs qui standardisent l'intégration
- Mode distribué ou standalone
- Une interface REST permettant de gérer facilement les connecteurs
- Gestion automatique des offsets
- Distribué et scalable : Possibilité de scaler les workers, basé sur la gestion de groupe



Mode standalone

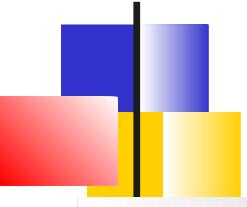
Pour démarrer *Kafka Connect* en mode standalone

```
> bin/connect-standalone.sh config/connect-standalone.properties  
    connector1.properties [connector2.properties ...]
```

Le premier paramètre contient la configuration pour le worker. Il inclut entre autres :

- ***bootstrap.servers***
- ***key.converter, value.converter***
- ***offset.storage.file.filename*** : Fichier pour stocker les offsets
- ***plugin.path*** : Une liste de chemins qui contiennent les plugins KafkaConnect (connecteurs, convertisseurs, transformations).

Les autres paramètres définissent les configurations des différents connecteurs



Mode distribué

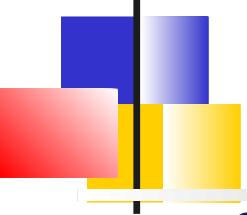
Le mode distribué gère le scaling dynamique et offre la tolérance aux pannes

Pour démarrer en mode distribué :

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

En mode distribué, *Kafka Connect* stocke les offsets, les configuration et les statuts des taches dans des topics.

- Pour contrôler, le nombre de partitions et le facteur de réPLICATION utilisés, il est recommandé de créer manuellement ces topics



Configurations supplémentaires en mode distribué

group.id (*connect-cluster* par défaut) : nom unique pour former le groupe

config.storage.topic (*connect-configs* par défaut) : *topic* utilisé pour stocker la configuration.

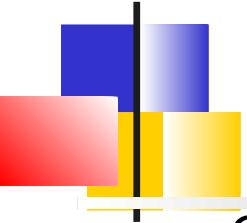
Doit s'agir d'une rubrique à partition unique, hautement répliquée et compactée.

offset.storage.topic (*connect-offsets* par défaut) : *topic* utilisé pour stocker les offsets;

Doit avoir de nombreuses partitions, être répliquée et être configurée pour le compactage

status.storage.topic (*connect-status* par défaut) : topic utilisé pour stocker les états;

Peut avoir plusieurs partitions et doit être répliquée et configurée pour le compactage



Configuration des connecteurs

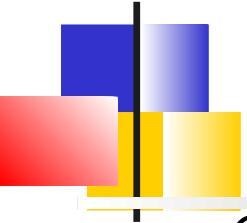
Configuration :

- En mode standalone, via un fichier *.properties* ou l'*API Rest*
- En mode distribué, via une API Rest .

Les valeurs sont très dépendantes du type de connecteur.

Comme valeur communes, on peut citer :

- ***name*** : Nom unique.
- ***connector.class*** : Classe Java du connecteur
- ***tasks.max*** : Maximum de tâches créés pour le connecteur. (degré de parallélisme)
- ***key.converter, value.converter***
- ***topics, topics.regex, topic.prefix*** : Liste, expression régulière ou gabarit spécifiant les topics



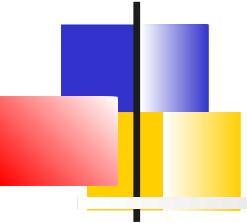
Connecteurs

Confluent offre de nombreux connecteurs¹ :

- Active MQ Source Connector
- Amazon S3 Sink Connector
- Elasticsearch Sink Connector
- FileStream Connector (Development and Testing)
- IBM MQ Source Connector
- JDBC Connector (Source et Sink)
- JMS Source Connector
- ...

De nombreux éditeurs fournissent également leur connecteurs Kafka
(Amazon, Azure, Google, Salesforce, TIBCO, MongoDB, ...)

1. <https://www.confluent.io/fr-fr/product/connectors/>



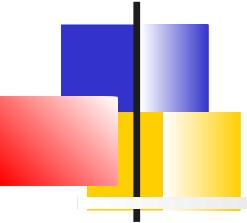
Exemple Connecteur JDBC

```
name=mysql-whitelist-timestamp-source
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
tasks.max=10

connection.url=jdbc:mysql://mysql.example.com:3306/my_database?
    user=alice&password=secret
table.whitelist=users,products,transactions

# Pour détecter les nouveaux enregistrements
mode=timestamp+incrementing
timestamp.column.name=modified
incrementing.column.name=id

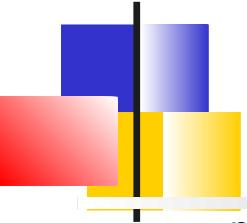
topic.prefix=mysql-
```



Transformations

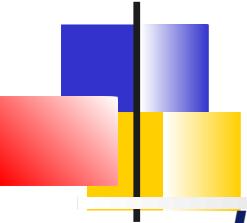
Une chaîne de transformation, s'appuyant sur des **transformers** prédéfinis, est spécifiée dans la configuration d'un connecteur.

- **transforms** : Liste d'aliases spécifiant la séquence des transformations
- **transforms.\$alias.type** : La classe utilisée pour la transformation.
- **transforms.\$alias.\$transformationSpecificConfig** : Propriété spécifique d'un *Transformer*



Exemple de configuration

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
# Un sink défini un topic de destination
topic=connect-test
# 2 transformers nommés MakeMap et InsertSource
transforms=MakeMap, InsertSource
# La ligne du fichier devient le champ line
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
# Le champ data-source est ajouté avec une valeur statique
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```



Quelques *transformers*

HoistField : Encapsule l'événement entier dans un champ unique de type Struct ou Map

InsertField : Ajout d'un champ avec des données statiques ou des métadonnées de l'enregistrement

ReplaceField : Filtrer ou renommer des champs

MaskField : Remplace le champ avec une valeur nulle

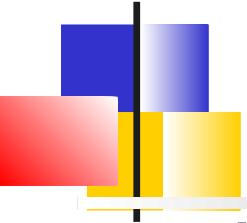
ValueToKey : Échange clé/valeur

ExtractField : Construit le résultat à partir de l'extraction d'un champ spécifique

SetSchemaMetadata : Modifie le nom du schéma ou la version

TimestampRouter : Route le message en fonction du timestamp

RegexRouter : Modifie le nom du topic le destination via une regexp

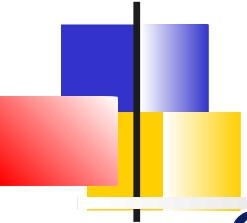


API Rest

L'adresse d'écoute de l'API REST peut être configuré via la propriété `listeners`
`listeners=http://localhost:8080, https://localhost:8443`

Ces listeners sont également utilisés par la communication intra-cluster

- Pour utiliser d'autres ips pour le cluster, positionner :
`rest.advertised.listener`



API

GET /connectors : Liste des connecteurs actifs

POST /connectors : Création d'un nouveau connecteur

GET /connectors/{name} : Information sur un connecteur (config et statuts et tasks)

PUT /connectors/{name}/config : Mise à jour de la configuration

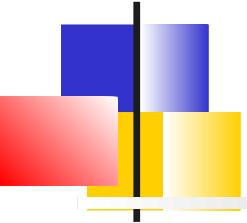
GET /connectors/{name}/tasks/{taskid}/status : Statut d'une tâche

PUT /connectors/{name}/pause : Mettre en pause le connecteur

PUT /connectors/{name}/resume : Réactiver un connecteur

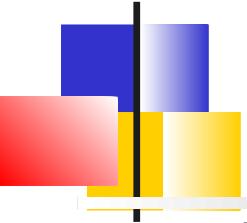
POST /connectors/{name}/restart : Redémarrage après un plantage

DELETE /connectors/{name} : Supprimer un connecteur



APIs

Producer API
Consumer API
Schema Registry
Connect API
Admin API



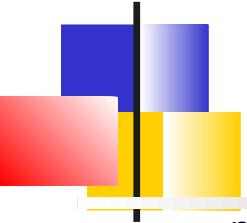
Introduction

Kafka propose 2 autres APIs :

- **Admin API¹** : Client d'administration permettant de gérer et inspecter les topics, brokers, configurations et ACLs
- **Streams API²** : Librairie cliente pour des micro-services dont les entrées/sorties sont des *topics* Kafka

1. API existante dans les autres langages chez Confluent

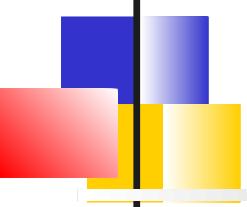
2. Équivalent Python : Faust, Équivalent C# : Streamiz



Exemple Admin : Lister les configurations

```
public class ListingConfigs {

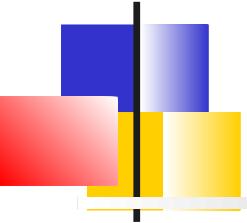
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        Properties config = new Properties();
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        AdminClient admin = AdminClient.create(config);
        for (Node node : admin.describeCluster().nodes().get()) {
            System.out.println("-- node: " + node.id() + " --");
            ConfigResource cr = new ConfigResource(ConfigResource.Type.BROKER,
                "0");
            DescribeConfigsResult dcr =
                admin.describeConfigs(Collections.singleton(cr));
            dcr.all().get().forEach((k, c) -> {
                c.entries()
                    .forEach(configEntry -> {
                        System.out.println(configEntry.name() + " = " +
                            configEntry.value());
                    });
            });
        }
    }
}
```



Exemple Admin

Créer un topic

```
public class CreateTopic {  
    public static void main(String[] args) throws ExecutionException,  
        InterruptedException {  
        Properties config = new Properties();  
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,  
        "localhost:9092");  
        AdminClient admin = AdminClient.create(config);  
        //Créer un nouveau topics  
        System.out.println("-- creating --");  
        NewTopic newTopic = new NewTopic("my-new-topic", 1, (short) 1);  
        admin.createTopics(Collections.singleton(newTopic));  
  
        //lister  
        System.out.println("-- listing --");  
        admin.listTopics().names().get().forEach(System.out::println);  
    }  
}
```



Garanties Kafka

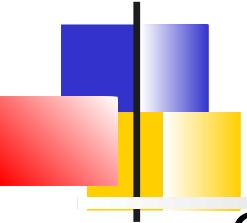
Mécanismes de réPLICATION

At Most One, At Least One

Exactly Once

Débit, latency, durabilité

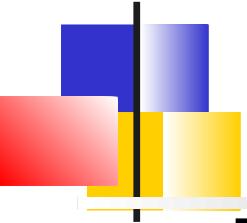
Stockage et rétention



Garanties Kafka

Garanties offertes par Kafka via les partitions et la réPLICATION :

- **Garantie de l'ordre à l'intérieur d'une partition.**
=> Le consommateur d'une partition lit dans l'ordre d'écriture des messages
=> Par contre l'ordre de la production n'est pas garantie si $\text{max_in_flight_request} > 1$ et $\text{retries} > 0$
- **Durabilité** : Les messages produits et répliquées sont disponibles tant qu'au moins une réplique reste en vie.
- **Garantie de livraison.** Selon les configurations, Kafka garantit At Most Once, At Least Once, Exactly Once malgré un certaine défaillance des brokers ou consommateurs .



Rôles des brokers

Différents brokers participent à la gestion distribuée et répliquée d'un topic.

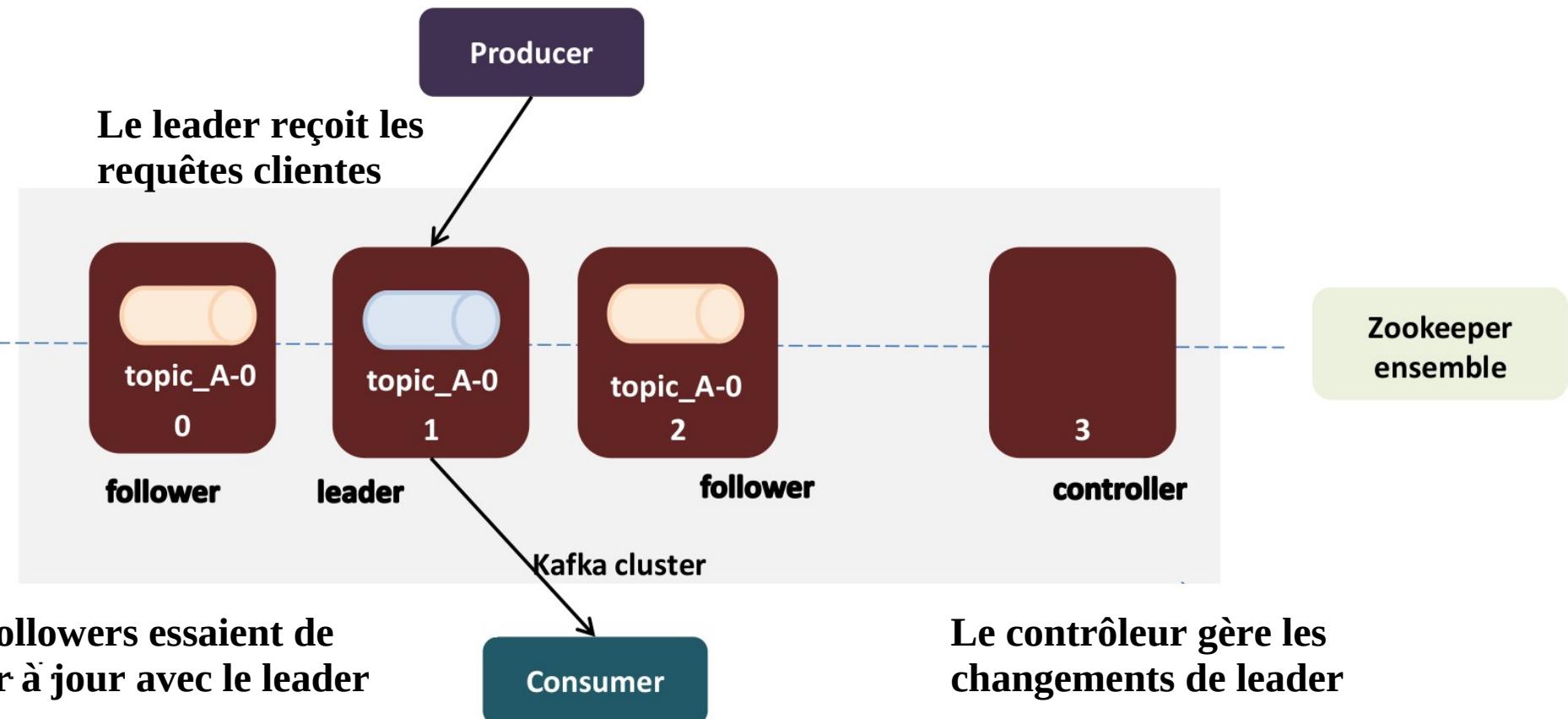
Pour chaque partition répliquée :

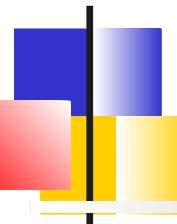
- **1 broker leader** : Détient la partition leader responsable de la validation des écritures
- **N followers** : Suivent les écritures effectuées par le leader. Un décalage est permis mais si il n'arrive plus à suivre la cadence d'écriture, ils sont éliminés

Au niveau du cluster :

- Un **contrôleur** est responsable des élections de leader

Rôle des brokers gérant un topic





Synchronisation des répliques

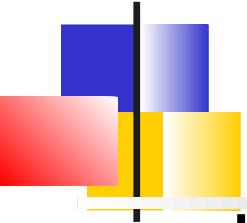
Contrôlé par la propriété :
replica.lag.time.max.ms (défaut 30 000)

Si pendant ce délai, le follower

- N'envoie pas de requêtes fetch
- N'atteint pas l'offset de fin du leader

Alors, le follower est considéré comme désynchronisé

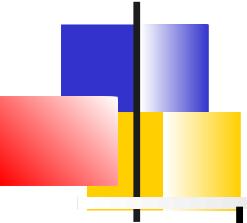
- Il est supprimé de la liste des **ISR (In Sync Replica)**
- Il peut ensuite rattraper son retard et être réintégré aux ISRs



Validation de la production

Le producteur peut considérer que le message produit est « committé » :

- Dès que le message est envoyé (acks=0 et prise maximale de risque)
- Lorsque le leader a écrit le message et renvoyé un acquittement (acks=1 et prise de risque de la défaillance du leader)
- Lorsque le message a été répliqué sur un minimum de réplique (acks=all, réduction du risque)

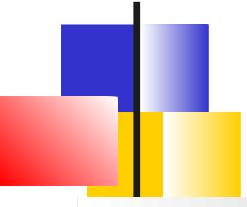


min.insync.replica

La propriété ***min.insync.replica***, spécifiée au niveau cluster ou topic, indique le minimum de répliques de l'ISR qui doivent avoir écrit un message afin que celui-ci soit considéré comme *committed/validé*

Tout cela pour un producteur configuré avec *acks=all*

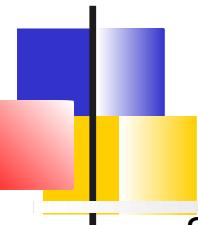
- A la réception d'un message, le leader vérifie si il y a assez d'ISR pour écrire le message, sinon il envoie une *NotEnoughReplicasException*
- Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



Conséquences

Un réplique synchronisée légèrement en retard peut ralentir l'acquittement du message ; ce qui peut ralentir le débit.

Une réplique désynchronisée n'a plus d'impact sur les performances mais augmente le risque d'un temps d'arrêt ou d'une perte de données.



Rejet de demande d'émission

Si le **nombre de ISR < *min.insync.replicas*** :

- Kafka empêche l'acquittement du message.
En fonction de la configuration du producteur, celui-ci peut être bloqué.

Si le **nombre de répliques disponibles < *min.insync.replicas***

- Mauvaise configuration, Kafka bloque les émissions de message

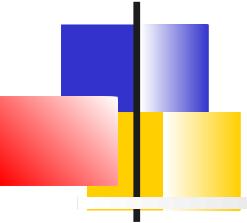
En conséquences :

n répliques

=> tolère $n-1$ failures pour que la partition soit disponible

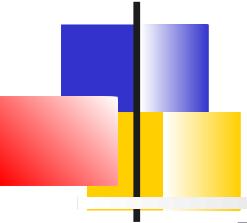
n répliques, $\text{min.insync.replicas} = m$

=> Tolère $n-m$ failures pour accepter les envois



Garanties Kafka

Mécanismes de réPLICATION
At Most Once, At Least Once
Exactly Once
Débit, latency, durabilité
Stockage et rétention



Côté producteur

Du côté producteur, 2 premiers facteurs influencent la fiabilité de l'émission

- La configuration des **acks** en fonction du *min.insync.replica* du topic
3 valeurs possibles : *0,1,all*
- Le paramètre **retries** qui permet de tolérer les défaillances temporaires
(un broker qui redémarre par exemple)

acks=0

acks=0 : Le producteur considère le message écrit au moment où il l'a envoyé sans attendre l'acquittement du broker
=> Perte de message potentielle (*At Most Once*)



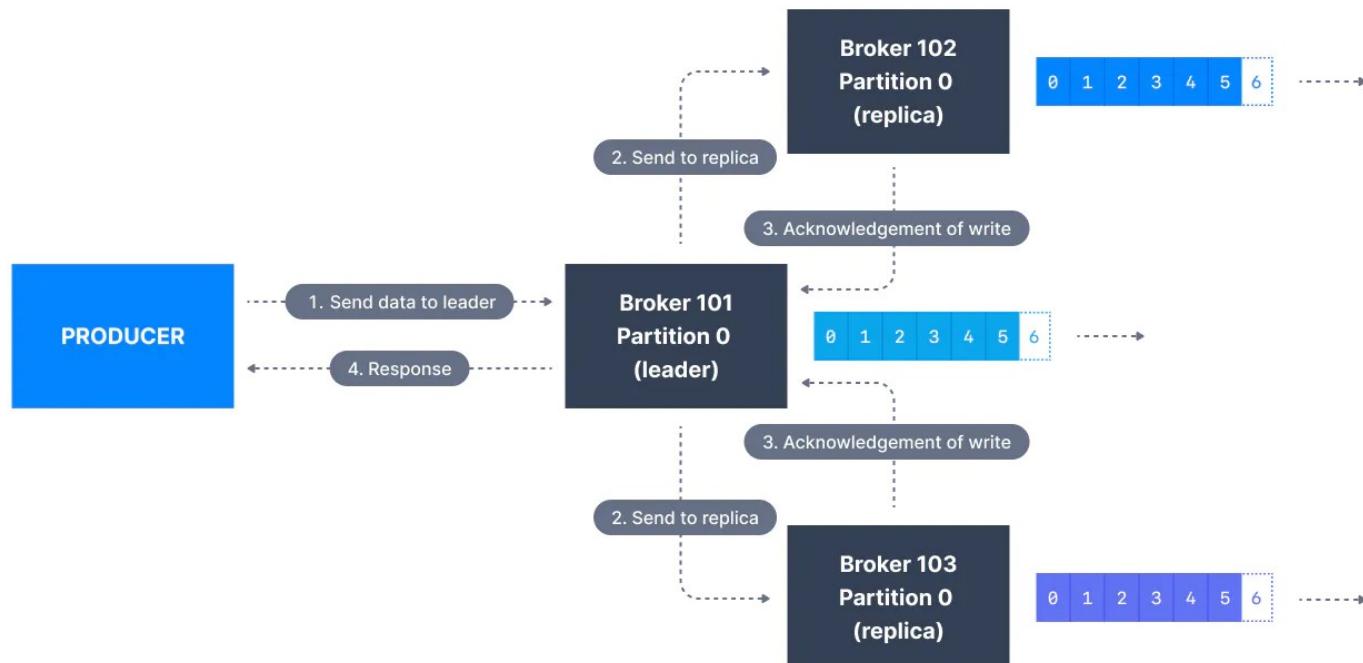
acks=1

acks=1 : Le producteur considère le message écrit lorsque le leader a acquitté l'écriture
=> Si le leader s'arrête et que les répliques n'ont pas eu le temps d'écrire le message, perte de données

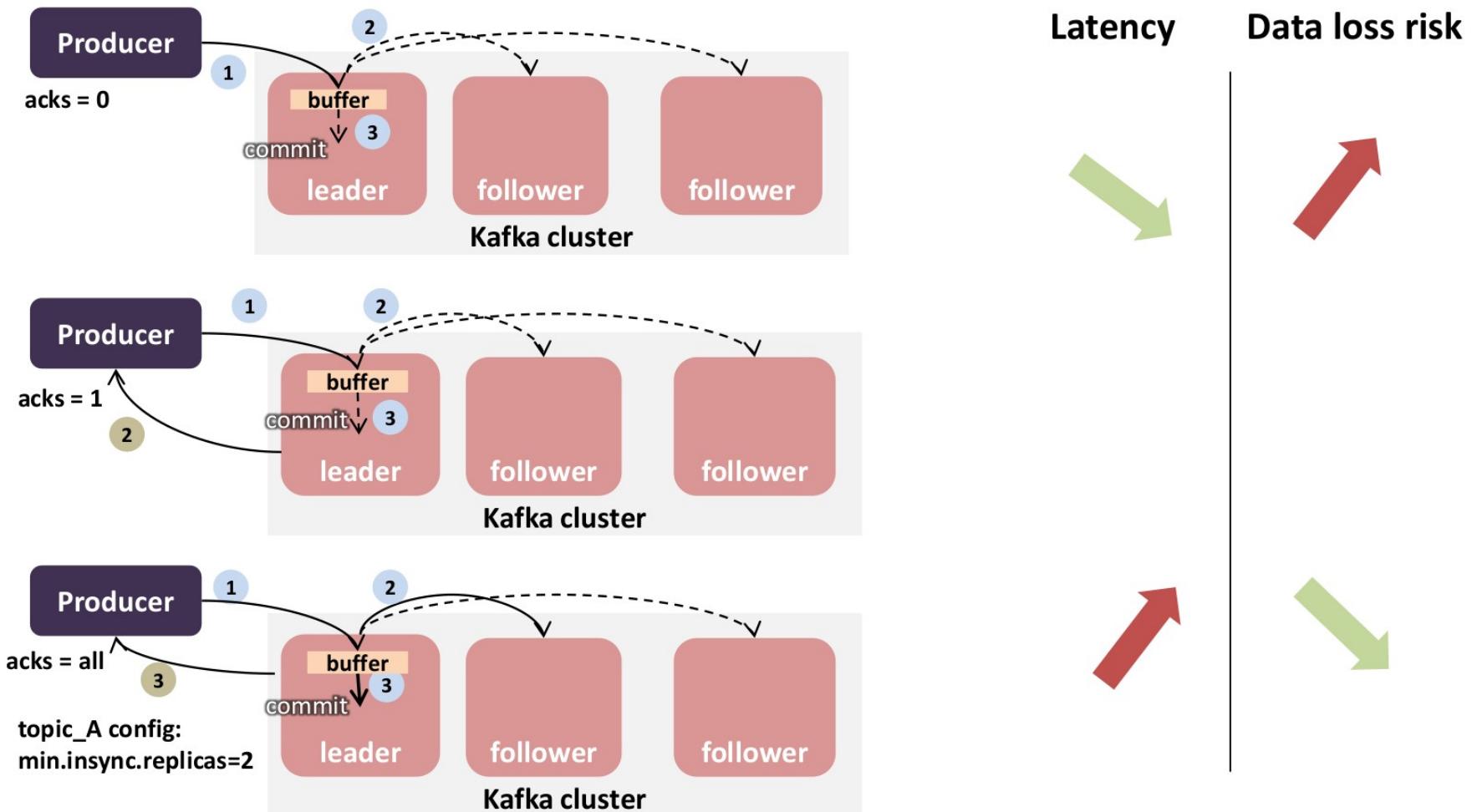


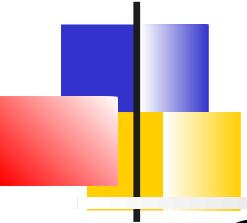
acks=all

acks=all : Le producteur considère le message comme écrit lorsque il a été répliqué par *min.insync.replica* des ISR.
=> Assure le maximum de durabilité
(Nécessaire pour *At Least Once* et *Exactly Once*)



Acquittement et durabilité

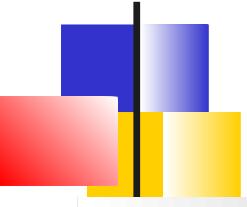




Gestion des erreurs

2 types d'erreur pour le producteur:

- les erreurs que les producteurs gèrent automatiquement .
Ce sont les erreurs ré-essayable (ex :
`LEADER_NOT_AVAILABLE,`
`NOT_ENOUGH_REPLICA`)
Nombre d'essai configurable via ***retries***.
=> Attention, peut générer des doublons
- les erreurs qui doivent être traitées par le code. (ex : `INVALID_CONFIG`,
`SERIALIZATION_EXCEPTION`)

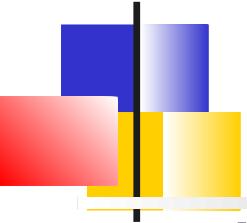


Côté consommateur

Du point de vue de la fiabilité, la seule chose que les consommateurs ont à faire est de s'assurer qu'ils gardent une trace des offsets qu'ils ont traités en cas de rebalancering.

Pour cela, ils commettent leur offset auprès du cluster Kafka qui stocke les informations dans le topic **consumer_offsets**

=> La seule façon de perdre des messages est alors de committer des offsets de messages lus mais pas encore traités



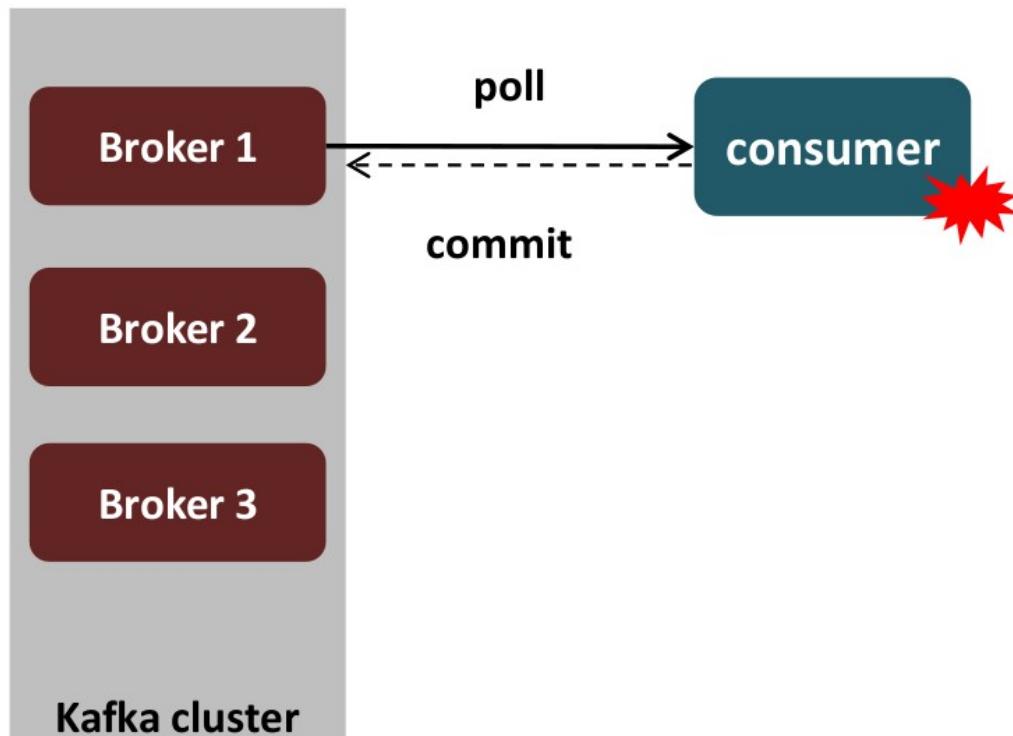
Gestion des commits

Les commits peuvent être automatiquement géré par Kafka

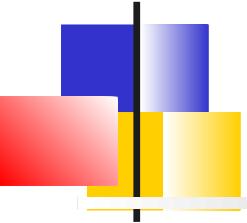
Si ***enable.auto.commit=true*** ,

Lors de l'appel à `poll()` et si ***auto.commit.interval.ms*** s'est écoulée, il valide les plus grands offset reçus par le `poll()` précédent

Consommation: At Most Once



- L'offset est committé
- Traitement d'un ratio de message puis plantage



Configuration At Most Once

enable.auto.commit = true.

+ traitement asynchrone dans la boucle de poll

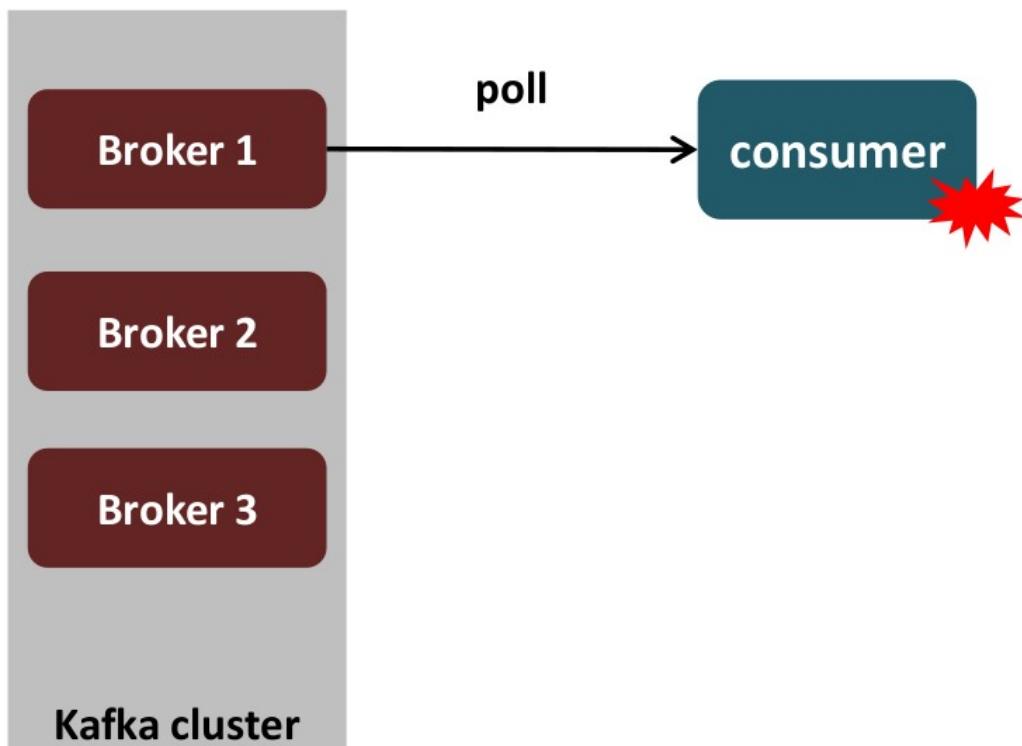
OU

Commit manuel avant le traitement des messages

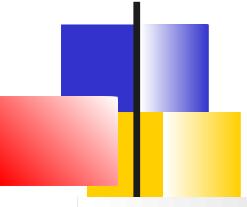
Exemple : Auto-commit et traitement asynchrone

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records) {  
        // Traitement asynchrone  
        asyncProcess(record);  
    }  
}
```

Réception : *At Least Once*



- Traitement d'un ratio de message puis plantage
- L'offset n'est pas commité



Configuration At Least Once

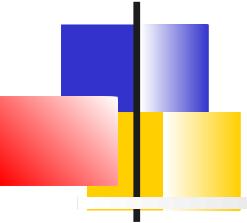
Configuration par défaut et traitement synchrone dans la boucle de poll

Ou

enable.auto.commit à false ET Commit explicite après traitement via
consumer.commitSync();

Exemple : Commit manuel après traitement

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
  
    for (ConsumerRecord<String, String> record : records) {  
        syncProcess(record) ;  
    }  
  
    consumer.commitSync();  
}
```

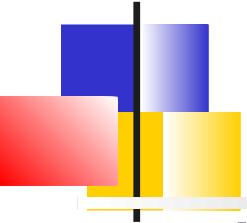


Garanties Kafka

Mécanismes de réPLICATION
At Most Once, At Least Once

Exactly Once

Débit, latency, durabilité
Stockage et rétention

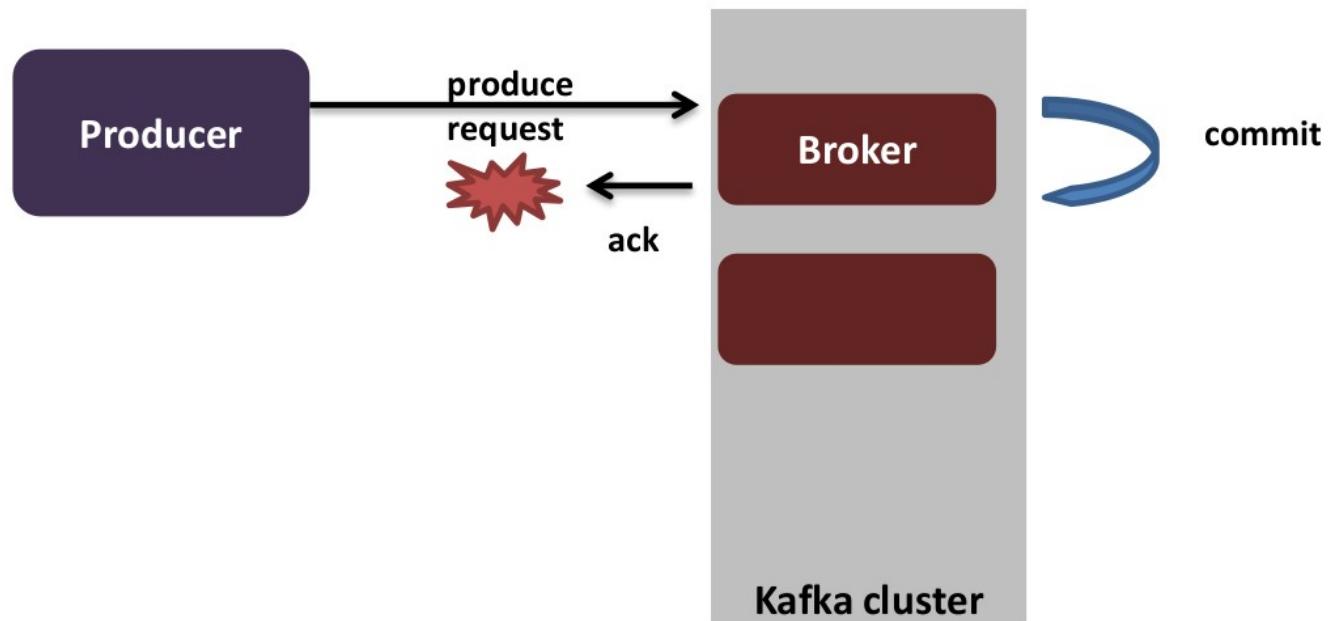


Introduction

La sémantique *Exactly Once* est basée sur *At least Once* et empêche les messages en double en cours de traitement par les applications clientes

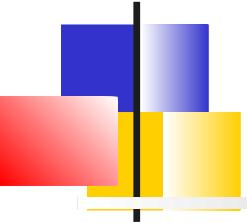
Pris en charge par les APIs *Kafka Producer, Consumer et Streams*

Producteur idempotent



Le producteur ajoute une
nombre séquentiel et un ID
de producteur

Le broker détecte le
doublon
=> Il envoie un ack sans le
commit



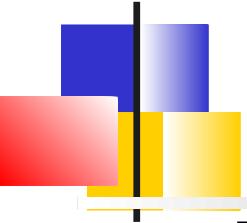
Configuration du producteur

enable.idempotence

L'idempotence nécessite les configurations suivantes :

- *max.in.flight.requests.per.connection <= 5*
- *retries > 0*
- *acks = all*

Si des valeurs incompatibles sont configurées, une *ConfigException* est lancée

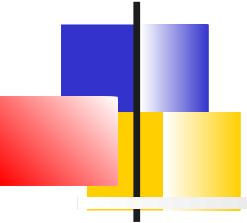


Consommateur

Du côté du consommateur, traiter une et une seule fois les messages consiste à :

- gérer manuellement les offsets des partitions dans un support de persistance transactionnel et partagé par tous les consommateurs et gérer les rééquilibrages
- Si le traitement consiste à produire un message vers le même cluster Kafka, on peut utiliser les transactions¹.

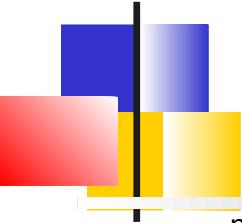
1. C'est le cas de KafkaStream



Exemple Gestion des offsets (1)

enable.auto.commit=false

```
while (true) {  
    ConsumerRecords<String, String> records =  
        consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("offset = %d, key = %s, value = %s\n",  
            record.offset(),  
            record.key(), record.value());  
  
        // Sauvegarder l'offset traité .  
        offsetManager.saveOffsetInExternalStore(record.topic(),  
            record.partition(), record.offset());  
    }  
}
```



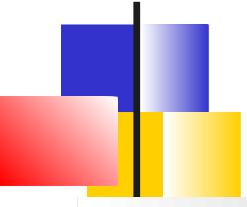
Exemple Gestion des offsets (2)

```
public class MyConsumerRebalancerListener implements
    org.apache.kafka.clients.consumer.ConsumerRebalanceListener {

    private OffsetManager offsetManager = new OffsetManager("storage2");
    private Consumer<String, String> consumer;

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            offsetManager.saveOffsetInExternalStore(partition.topic(),
                partition.partition(), consumer.position(partition));
        }
    }

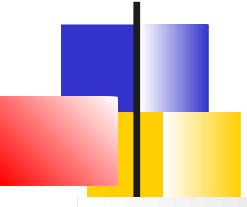
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            consumer.seek(partition,
                offsetManager.readOffsetFromExternalStore(partition.topic(),
                    partition.partition()));
        }
    }
}
```



Transaction

Pour utiliser le mode transactionnel et l'API correspondante, il faut positionner la propriété ***transactional.id*** à une chaîne de caractères unique par producteur

- Le mode transactionnel inclut le mode idempotent
- Les topics impliqués doivent être configurés avec *replication.factor >= 3* et *min.insync.replicas = 2*
- Les consommateurs doivent avoir la propriété *isolation.level* à *read committed*
- L'API *send()* devient bloquante

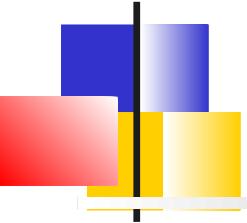


Transaction

Permet la production atomique de messages sur plusieurs partitions

Les producteurs produisent l'ensemble des messages ou aucun

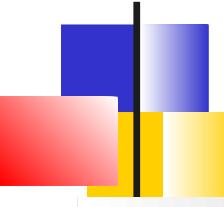
```
// initTransaction démarre une transaction avec l'id.  
// Si une transaction existe avec le même id, les messages sont roll-backés  
producer.initTransactions();  
try {  
    producer.beginTransaction();  
    for (int i = 0; i < 100; ++i) {  
        ProducerRecord record = new ProducerRecord("topic_1", null, i);  
        producer.send(record);  
    }  
    producer.commitTransaction();  
} catch(ProducerFencedException e) { producer.close(); } catch(KafkaException  
e) { producer.abortTransaction(); }
```



Configuration du consommateur

Afin de lire les messages transactionnels validés :

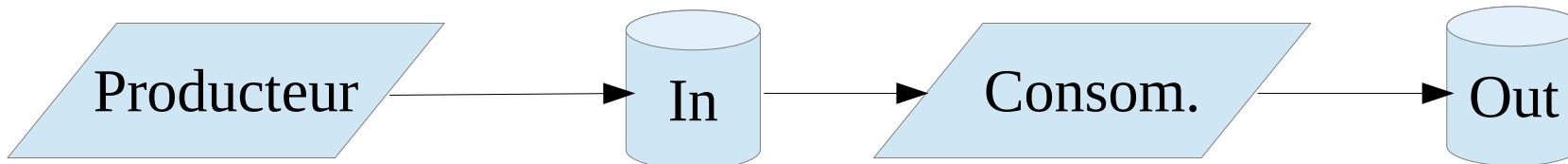
- ***isolation.level=read_committed***
(Défaut: *read_uncommitted*)
 - *read_committed*: Messages (transactionnels ou non) validés
 - *read_uncommitted*: Tous les messages (même les messages transactionnels non validés)

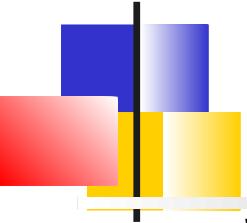


Exactly Once pour le transfert de messages entre topics

Lorsque un consommateur produit vers un autre topic, on peut utiliser les transactions afin d'écrire l'offset vers Kafka dans la même transaction que le topic de sortie

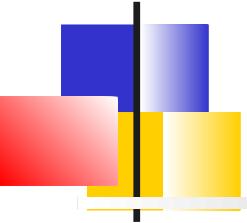
Si la transaction est abandonnée, la position du consommateur reviendra à son ancienne valeur et les données produites sur les topics de sortie ne seront pas visibles pour les autres consommateurs, en fonction de leur "niveau d'isolement".





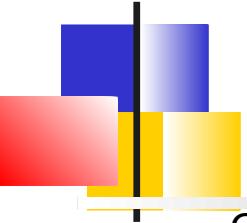
Consommateur

```
while (true) {  
  
    ConsumerRecords<String, String> records = consumer.poll(ofSeconds(60));  
  
    // Transform message  
    Map<String, Integer> wordCountMap = ...  
  
    producer.beginTransaction();  
  
    wordCountMap.forEach((key, value) ->  
        producer.send(new ProducerRecord<String, String>(OUTPUT_TOPIC, key, value.toString())));  
  
    Map<TopicPartition, OffsetAndMetadata> offsetsToCommit = new HashMap<>();  
  
    // Retreive offsets for each partition  
    for (TopicPartition partition : records.partitions()) {  
        List<ConsumerRecord<String, String>> partitionedRecords = records.records(partition);  
        long offset = partitionedRecords.get(partitionedRecords.size() - 1).offset();  
        offsetsToCommit.put(partition, new OffsetAndMetadata(offset + 1));  
    }  
    // Commit Offset for consumer associated with the commit of the transaction  
    producer.sendOffsetsToTransaction(offsetsToCommit, CONSUMER_GROUP_ID);  
    producer.commitTransaction();  
  
}
```



Garanties Kafka

Mécanismes de réPLICATION
At Most Once, At Least Once
Exactly Once
Débit, latence, durabilité
Stockage et rétention



Configuration pour favoriser le débit

Côté producteur :

Augmenter

- *batch.size*: ex : 100000 (défaut 16384)
- *linger.ms*: ex : 100 (default 0)
- *buffer.memory*: si il y a beaucoup de partitions (défaut 32 Mb)

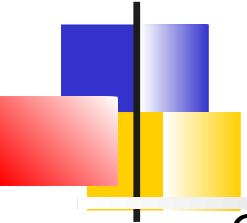
Puis

- *compression.type=lz4* (défaut none)
- *acks=1* (défaut all)

Côté consommateur

Augmenter

- *fetch.min.bytes*: ex : 100000 (défaut 1)



Configuration pour favoriser la latence

Cluster

Si followers pas suffisamment rapide, augmenter leur nombre

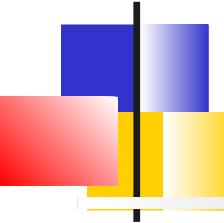
- *num.replica.fetchers* : (défaut 1)

Côté producteur

- *linger.ms: 0*
- *compression.type=none*
- *acks=1*

Côté consommateur

- *fetch.min.bytes: 1*



Configuration pour la durabilité

Cluster

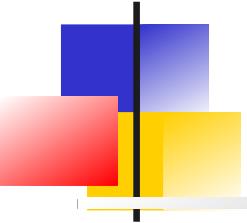
- *replication.factor: 3*
- *min.insync.replicas: 2* (défaut 1)
- *unclean.leader.election.enable : false* (défaut false)
- *broker.rack: rack du broker* (défaut null)

Producteur

- *acks:all* (défaut all)
- *enable.idempotence:true* (défaut false)
- *max.in.flight.requests.per.connection: <=5*

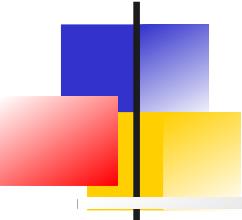
Consommateur

- *isolation.level: read_committed*



Garanties Kafka

Mécanismes de réPLICATION
At Most Once, At Least Once
Exactly Once
Débit, latency, durabilité
Stockage et rétention

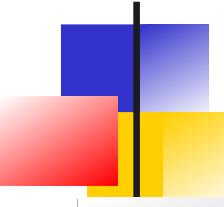


Introduction

L'unité de stockage de Kafka est une réplique de partition.

- => Les partitions ne peuvent pas être divisées entre plusieurs brokers ni entre plusieurs disques du même broker

La propriété ***log.dirs*** définit les répertoires de stockage des partitions

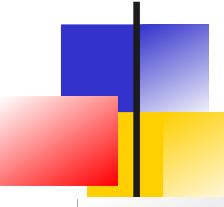


Allocation des partitions

A la création des *topics*, Kafka décide comment allouer les partitions sur les brokers

Ses objectifs sont :

- Répartir uniformément les répliques entre les brokers
- S'assurer que chaque réplica d'une partition se trouve sur un broker différent
- Si les brokers ont des informations sur le rack, s'assurer que les répliques sont affectés à des racks différents si possible



Rétention des données

Pour accélérer la purge des messages obsolètes, Kafka utilise les **segments**

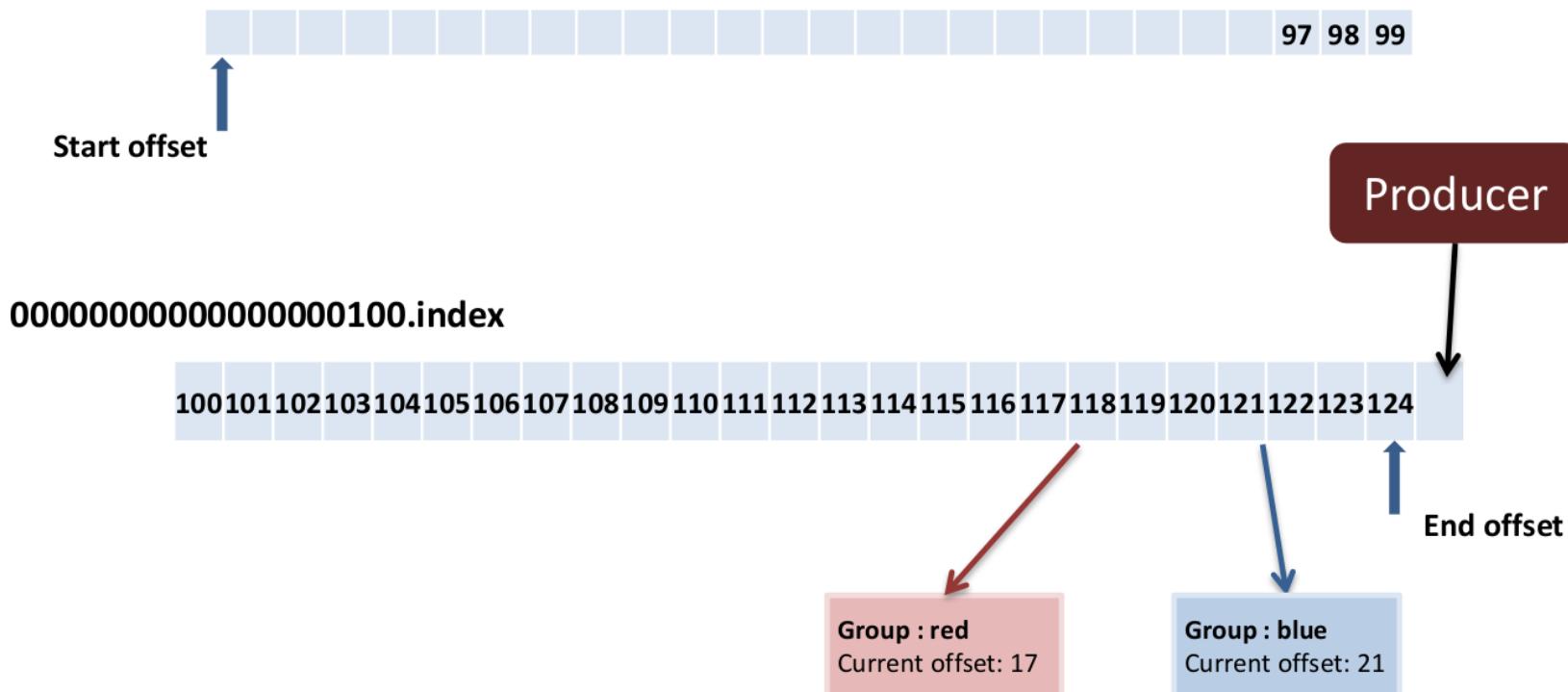
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite de taille est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas être purgé

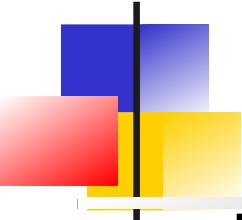
Segments

Log segment

closing segment parameters :

- log.roll.ms (ignored if log.roll.hours is set)
- log.roll.hours (default 168)
- log.segment.bytes (default 1073741824)



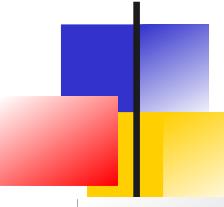


Indexation

Kafka permet de récupérer des messages à partir de n'importe quel offset disponible.

Pour accélérer cette opération, Kafka maintient un **index** pour chaque partition

- L'index associe les offsets aux segments et aux positions dans le fichier
- Les index sont également divisés en segments
- En cas de suppression, ils peuvent être régénérés automatiquement par Kafka



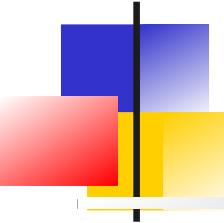
Nettoyage des logs

La propriété ***log.cleanup.policy*** détermine la stratégie de purge :

Les valeurs possibles sont :

- ***delete*** (défaut) : Suppression des vieux segments en fonction de l'âge ou de la taille de la partition
- ***compact*** : Suppression basée sur les clés des messages quelque soit l'âge et la taille des données

Possibilité de combiner les 2 stratégies (delete et compact)

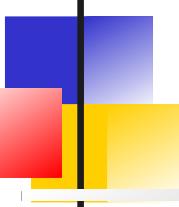


Stratégie *delete*

La stratégie *delete* s'appuie sur :

- ***log.retention.bytes*** (défaut -1 : infinite)
- ***log.retention.ms*** (défaut null)
- ***log.retention.minutes*** (défaut null)
- ***log.retention.hours*** (défaut 168, 1 semaine)

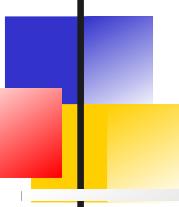
=> Contrôle de l'usage disque



Compactage

Avec la stratégie de rétention ***compact***, kafka ne conserve que la valeur la plus récente pour chaque clé du *topic*.

- Propriétés ***cleanup.policy=compact*** et ***log.cleaner.enabled=true***
- Les messages doivent alors contenir une clé
- Le compactage est effectué par une thread séparé qui périodiquement purge les messages *dirty*



Stratégie compact

2 propriétés de configuration importantes pour cette stratégie :

- ***cleaner.min.compaction.lag.ms*** : Le temps minimum qu'un message reste non compacté
- ***cleaner.max.compaction.lag.ms*** : Le temps maximum qu'un message reste inéligible pour la compactage

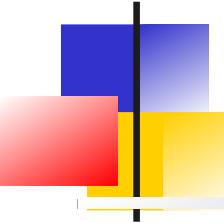
Le *nettoyeur (log cleaner)* est implémenté par un pool de threads configurable :

- ***log.cleaner.threads*** : Le nombre de threads
- ***log.cleaner.backoff.ms*** : Le temps de pause lorsqu'il n'y pas de travail (défaut 15 secondes)
-

Lors du nettoyage, le segment actif est fermé (un nouveau segment est créé)

Sauvegarde la dernière valeur (dernière mise à jour) pour chaque clé

=> Consomme CPU et RAM

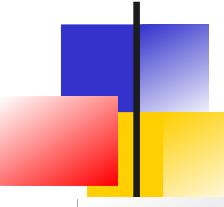


Exemple compact

1	2	3	4	5	6	7
K1	K2	K3	K4	K4	K5	K1
6	3	2	2	1	3	2



2	3	5	6	7
K2	K3	K4	K5	K1
3	2	1	3	2



Conséquences stratégie compact

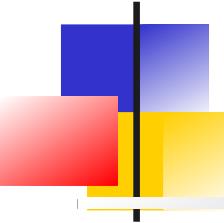
Consommateurs «à jour» (offset courant dans le segment actif), récupèrent tous les messages

- Le segment actif n'est pas compacté
 - => Le compactage n'empêche pas le consommateur de lire les données en double

L'ordre des messages est sauvegardé

Les offsets des messages sont sauvegardés

Les messages supprimés sont toujours disponibles pour les consommateurs actifs jusqu'à ce que *delete.retention.ms* soit expiré (24h par défaut)



Paramètres de compactage

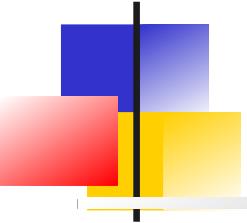
segment.ms (7 jours)

segment.byte (1G)

min.compaction.lag.ms (défaut 0)

delete.retention.ms (défaut 1 jour)

min.cleanable.dirty.ratio (défaut 0.5) : réduire pour un nettoyage plus efficace



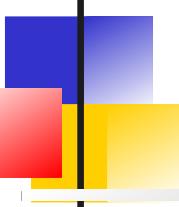
KafkaStream

Concepts

Opérateurs stateless

Opérateurs stateful

ksqldb



Application KafkaStream

Une application KafkaStream définit sa logique de traitement à travers une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

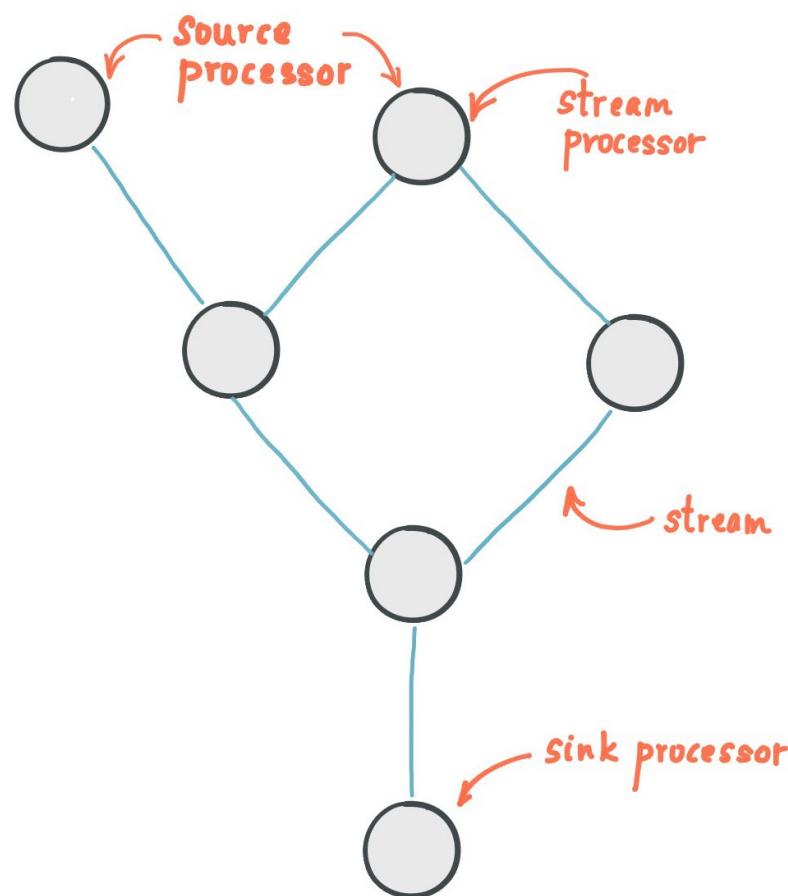
Un processeur représente une étape de traitement qui prend en entrée un événement, le transforme puis produit un ou plusieurs événements

Certains processeurs :

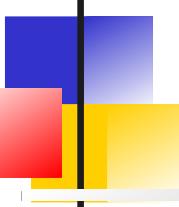
- n'ont pas de connexions entrantes : **Source**
- d'autres n'ont pas de connexions sortantes : **Sink**
- Les autres ont des connexions entrantes et sortantes

La topologie peut être spécifiée programmatiquement ou par un DSL offrant les opérateurs classiques (filter, map, join, ...)

Topologie processeurs

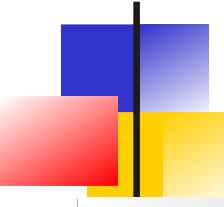


PROCESSOR TOPOLOGY



Apports de *KafkaStream*

- Abstractions ***KStream*** et ***KTable*** permettant la transformation de flux d'évènements, des jointures entre fluix, des agrégations et des requêtes
- Peut garantir que chaque évènement soit traité une et une seule fois, même en cas de défaillance.
(Par défaut, configuration *At Least Once*)
- Temps de latence des traitements en millisecondes, modèle énormément scalable
- Un ensemble d'opérateurs stateless ou stateful permettant de filtrer, transformer les messages
- Supporte des opérations de fenêtrage temporel avec l'arrivée des événements dans le désordre.



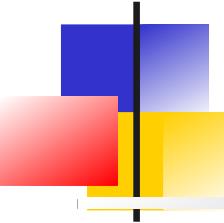
Définitions

Un **KStream** est une abstraction représentant un ensemble de données illimité, c'est à dire infini et sans cesse croissant car de nouveaux enregistrements continuent d'arriver

Les caractéristiques de ce flux :

- Les événements sont partitionnés
- A l'intérieur d'une partition, les évènement sont ordonnés
- Les événements sont immuables et ont un horodatage

Si il est persisté, il correspond à un topic Kafka partitionné avec une stratégie d'archivage *delete*



KStream et KTable

KafkaStream fournit également l'abstraction ***KTable*** qui représente un ensemble de faits qui évoluent.

L'arrivée d'une valeur avec une clé connue met à jour la valeur précédente.

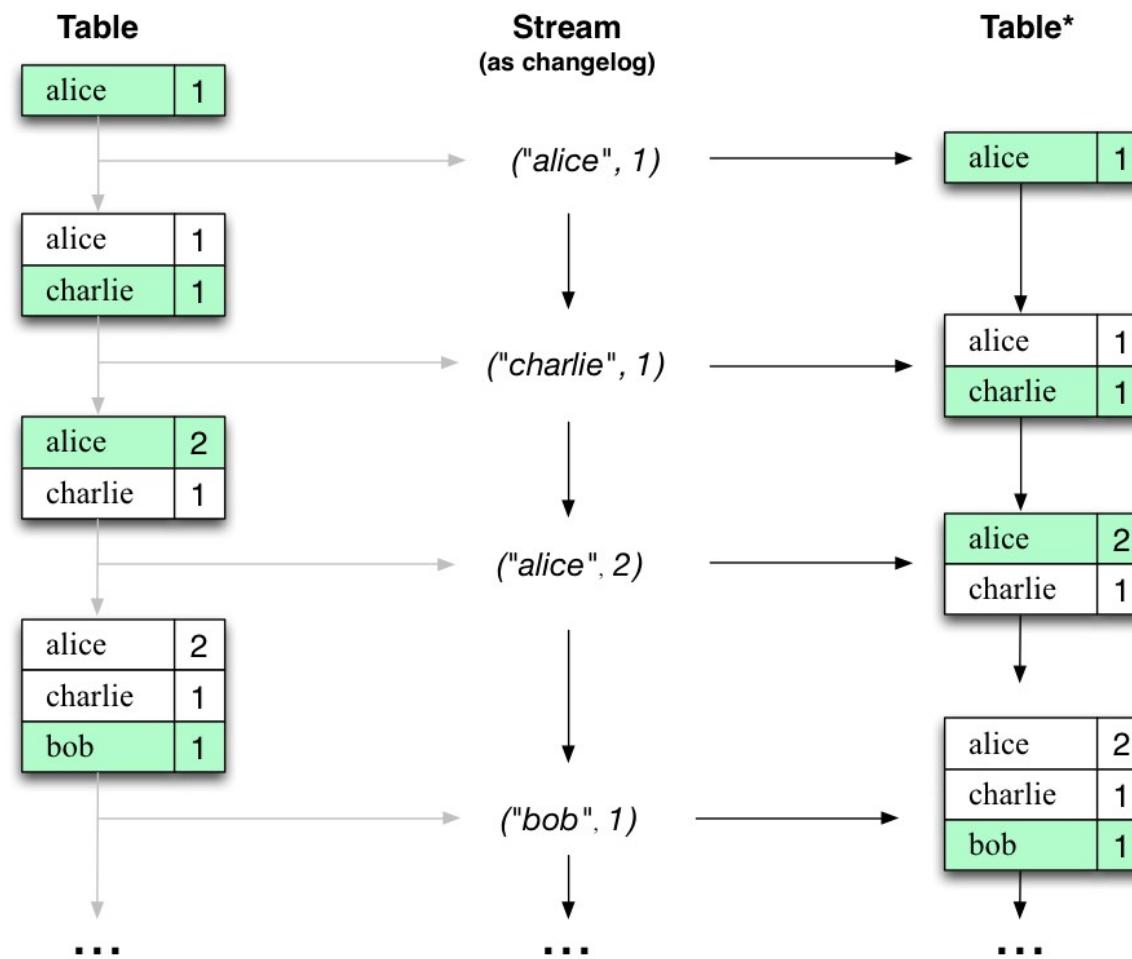
Cela peut être vu comme une table d'une base de données ou il n'existe qu'une valeur pour une clé donné.

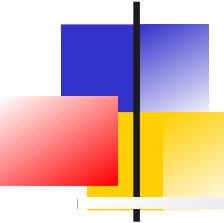
Cela correspond à un topic Kafka avec un archivage compact

A table

key1	value1
key2	value2
key3	value3

Dualité KStream / KTable



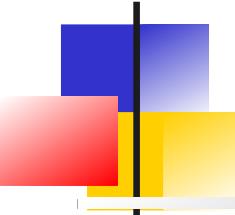


KGlobalTable

KGlobalTable représente une table distribuée globale.

Contrairement à une *CTable*, qui est partitionnée et contient donc un sous-ensemble des clés des évènements, Une *KGlobalTable* contient une copie complète des données.

Elle est accessible par chaque instance de l'application *KafkaStream*



Opérateurs stateless / stateful

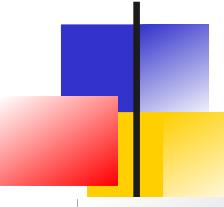
KafkaStream propose de nombreux opérateurs permettant de traiter les enregistrements entrants.

Certains sont **stateless**, d'autres **stateful**.

- Exemple stateless : filter, map, ...
- Exemple stateful : Agrégation, Jointure

Les opérateurs stateful doivent maintenir un état stocké dans un **StateStore**

- On peut interroger ces états via des *interactive queries*
- Les state store sont généralement partitionnés

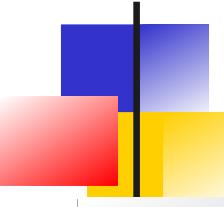


Tâches et partitions

En se basant sur les partitions Kafka et les clés des messages, KafkaStream implémenter le parallélisme via des **Task**

Les tâches permettent la scalabilité.

- *KafkaStream* crée un nombre fixe de tâches en fonction des partitions du flux d'entrée
- Chaque tâche est affectée à une liste de partitions. L'affectation ne change que lorsqu'une instance de l'application disparaît ou apparaît
- Les tâches instancient leur propre topologie de processeurs et ont leur propre StateStore



Modèle de threads

KafkaStream permet de configurer le nombre de threads utilisées pour paralléliser le traitement au sein d'une instance d'application.

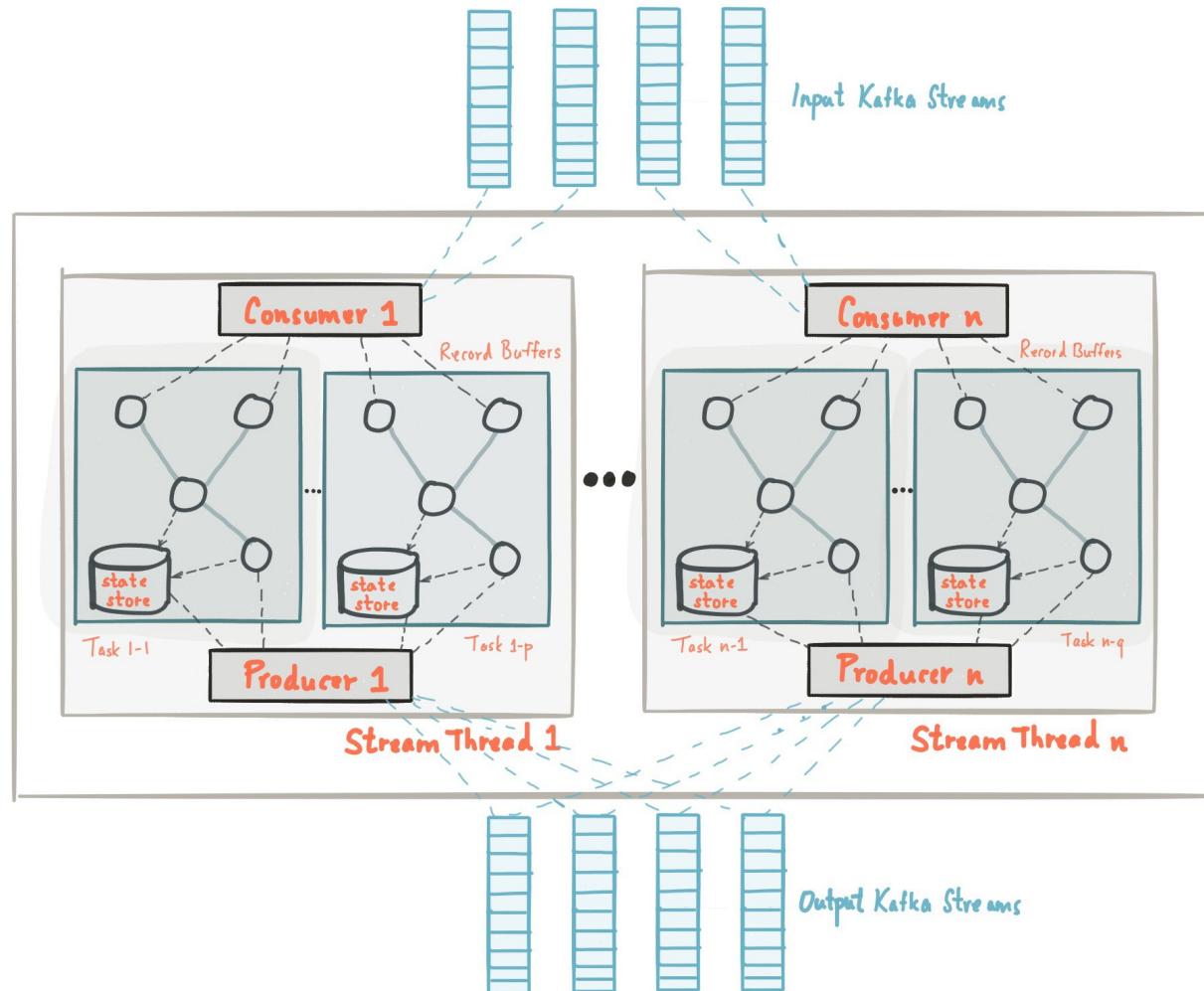
- Chaque thread peut exécuter une ou plusieurs tâches avec leurs topologies de processeur de manière indépendante.

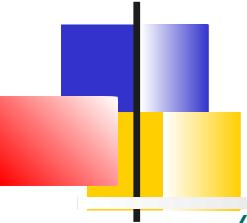
=> Le scaling horizontal consiste donc à augmenter le nombre de threads.

=> Le scaling vertical consiste à démarrer plusieurs fois la même instance.

Dans les 2 cas, Kafka Streams se charge de distribuer les partitions entre les tâches qui s'exécutent dans les instances de l'application.

Architecture et scalabilité





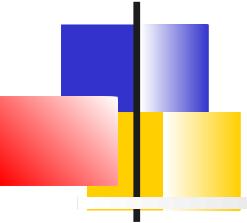
Exemple

```
// Propriétés : ID, BOOTSTRAP, Sérialiseur/Désérialiseur
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

// Création d'une topologie de processeurs
final StreamsBuilder builder = new StreamsBuilder();
builder.<String, String>stream("streams-plaintext-input")
    .flatMapValues(value -> Arrays.asList(value.split("\\w+")))
    .to("streams-linesplit-output");

final Topology topology = builder.build();

// Instanciation du Stream à partir d'une topologie et des propriétés
final KafkaStreams streams = new KafkaStreams(topology, props);
```

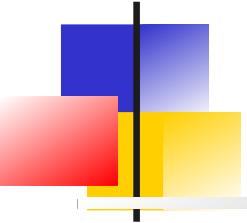


Exemple (2)

```
final CountDownLatch latch = new CountDownLatch(1);

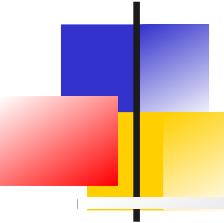
// attach shutdown handler to catch control-c
Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});

// Démarrage du stream
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
System.exit(0);
```



KafkaStream

Concepts
Opérateurs stateless
Opérateurs stateful
ksqldb

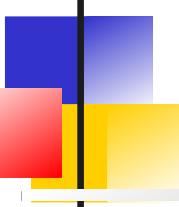


Processseurs stateless

Les processseurs stateless ne nécessitent pas de *StateStore*.

Cependant le résultat de la transformation peut être matérialisé sous forme de *KTable* permettant des requêtes interactives

Les processseurs prennent donc un argument optionnel, le nom du store associé à la Table



Les processeurs stateless

filter, filterNot : Filtre à partir d'une fonction booléenne

map, mapValues : Transformation du message

flatMap, flatMapValues : Un message produit 0, 1 ou plusieurs messages transformés

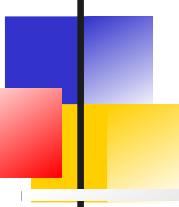
branch : Split le flux d'entrée en plusieurs flux

merge : Fusionne 2 flux

foreach : Opération terminale ne retournant pas de messages mais permettant de faire un traitement sur chaque message

print : Opération terminale affichant le message sur la console

peek : Idem que ForEach mais opération non terminale renvoyant le même flux



Les processeurs stateless (2)

groupBy : Regroupe les enregistrements par une nouvelle clé.
Pré-requis pour faire de l'agrégation

groupByKey : Regroupe en utilisant la clé

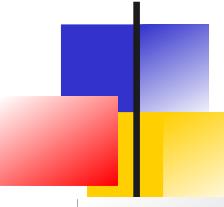
coGroup : Agrégation de plusieurs stream ayant les mêmes clés

selectKey : Permet de définir une nouvelle clé

mapAsync, mapValuesAsync, flatMapAsync,
FlatMapValuesAsync, ForEachAsync : Opérations
asynchrones sur le modèle requête réponse. Cas d'usage :
Enrichir les données avec un système externe

tableToStream, streamToTable :

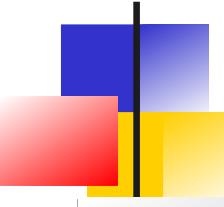
repartition : Repartitionne le flux avec le nombre de partitions
voulus



Exemples

```
// Branch
Map<String, KStream<String, Long>> branches =
    stream.split(Named.as("Branch-"))
        .branch((key, value) -> key.startsWith("A"), /* first predicate */
                 Branched.as("A"))
        .branch((key, value) -> key.startsWith("B"), /* second predicate */
                 Branched.as("B"))
        .defaultBranch(Branched.as("C"))                  /* default branch */
);

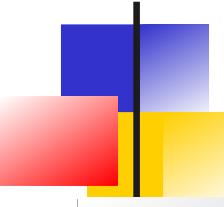
// KStream branches.get("Branch-A") contains all records whose keys start with "A"
// KStream branches.get("Branch-B") contains all records whose keys start with "B"
// KStream branches.get("Branch-C") contains all other records
// Filtre Kstream, KTable
KStream<String, Long> onlyPositives = stream.filter((key, value) -> value > 0);
// Map, MaValues
KStream<String, Integer> transformed = stream.map(
    (key, value) -> KeyValue.pair(value.toLowerCase(), value.length()));
KStream<byte[], String> uppercased = stream.mapValues(value -> value.toUpperCase());
```



Exemples

```
// FlatMap
KStream<String, Integer> transformed = stream.flatMap(
    // Here, we generate two output records for each input record.
    // We also change the key and value types.
    // Example: (345L, "Hello") -> ("HELLO", 1000), ("hello", 9000)
    (key, value) -> {
        List<KeyValue<String, Integer>> result = new LinkedList<>();
        result.add(KeyValue.pair(value.toUpperCase(), 1000));
        result.add(KeyValue.pair(value.toLowerCase(), 9000));
        return result;
    }
);

// Merge
KStream<byte[], String> merged = stream1.merge(stream2);
// Re-partitionnement
KStream<byte[], String> repartitionedStream =
    stream.repartition(Repartitioned.numberOfPartitions(10));
```

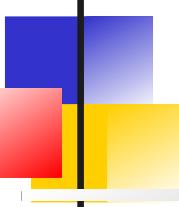


Exemples Side Effect

```
// ForEach : Kstream → void, KTable → void
stream.foreach((key, value) -> System.out.println(key + " => " + value));

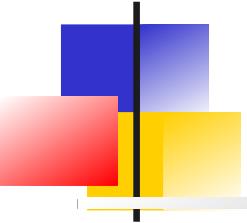
// Print : Kstream → void
stream.print(Printed.toFile("streams.out").withLabel("streams"));

// Peek : Kstream → Kstream
KStream<byte[], String> unmodifiedStream = stream.peek(
    (key, value) -> System.out.println("key=" + key + ", value=" +
value));
```



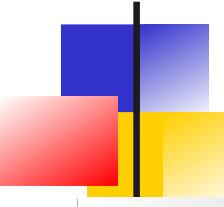
Exemples, groupes

```
// SelectKey : KStream → KStream
KStream<String, String> rekeyed = stream.selectKey((key, value) -> value.split(" ")[0])
// GroupByKey : Kstream → KgroupedStream
KGroupedStream<byte[], String> groupedStream = stream.groupByKey();
// GroupBy : KStream → KgroupedStream ou KTable → KGroupedTable
KGroupedTable<String, Integer> groupedTable = table.groupBy(
    (key, value) -> KeyValue.pair(value, value.length()),
    Grouped.with(
        Serdes.String(), /* key (note: type was modified) */
        Serdes.Integer()) /* value (note: type was modified) */
);
// Cogroup : KGroupedStream → CogroupedKStream, CogroupedKStream → CogroupedKStream
KGroupedStream<byte[], String> groupedStream = stream.groupByKey();
KGroupedStream<byte[], String> groupedStream2 = stream2.groupByKey();
CogroupedKStream<byte[], String> cogroupedStream =
    groupedStream.cogroup(aggregator1).cogroup(groupedStream2, aggregator2);
```



KafkaStream

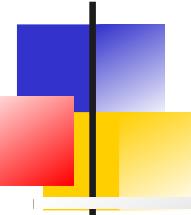
Concepts
Opérateurs stateless
Opérateurs stateful
ksqldb



Processseurs stateful

Les opérateurs stateful disponibles sont :

- Les agrégations
 - En continue
 - Utilisant des fenêtres temporelles
- Les jointures
 - Entre 2 Stream
 - Entre un Stream et une table
 - Entre 2 tables



State Store associés

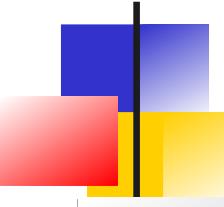
Les opérateurs stateful nécessitent un StateStore associé ou pas à une fenêtre temporelle

Les StateStore sont spécifiés via le paramètre ***materialized***.

3 types de StateStore sont possibles :

- les agrégations non fenêtrées et les KTables non fenêtrées utilisent des ***TimestampedKeyValueStores*** ou des ***VersionedKeyValueStores***¹
 - Les agrégations fenêtrées temporelles et les jointures *KStream-KStream* utilisent des ***TimestampedWindowStores***
 - Les agrégations fenêtrées de session utilisent des ***SessionStores***
- Les stateStore sont fault-tolerant

1. Les tables versionnées apportent des restrictions



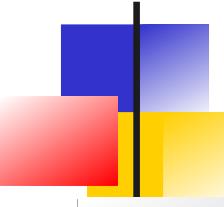
Agrégations

Pour appliquer une agrégation, il est généralement nécessaire de grouper les enregistrement avec les opérateurs stateless **GroupBy***

Ces opérateurs génèrent :

- Un **KGroupedStream** si appliqués sur un KStream
- Un **KGroupedTable** si appliqués sur une KTable

Ces opérateurs ont pour effet de repartitionner le flux en fonction de la nouvelle clé.



Opérations d'agrégation

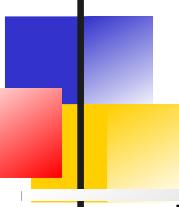
count : A partir d'un *Grouped**, compte les enregistrements

aggregate : A partir d'un *Grouped**, agrège les enregistrements.
Prend en arguments :

- La valeur initiale de l'agrégation
- La fonction d'agrégation (lambda)
- Le store via Materialized et le Serde permettant de sérialiser la valeur agrégée

reduce : A partir d'un *GroupedStream*, L'enregistrement courant est combiné avec la dernière valeur réduite et une nouvelle valeur réduite est renvoyée. Le type de valeur de résultat ne peut pas être modifié.

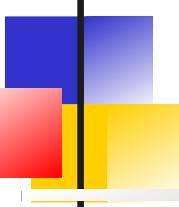
Lorsque l'on applique à un KGroupedTable, il faut en plus fournir un substractor qui s'applique lorsque un enregistrement change de groupe



Exemple count

```
KStream<String, String> textLines = ...;

KStream<String, Long> wordCounts = textLines
    .flatMapValues(value ->
        Arrays.asList(value.toLowerCase().split("\\\\w+")))
    // Groupe le flux, la clé est word.
    .groupByKey((key, word) -> word)
    // Count l'occurrence de chaque mot
    // `KGroupedStream<String, String>` devient `KTable<String, Long>` .
    .count()
    // Convertit the `KTable<String, Long>` into a `KStream<String, Long>` .
    .toStream();
```

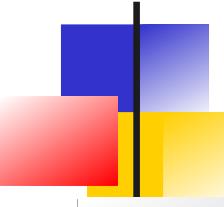


Agrégation en continu

```
KGroupedStream<byte[], String> groupedStream = ...;
```

```
// Aggregating a KGroupedStream

KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
    () -> 0L, /* Initialisation */
    (aggKey, newValue, aggValue) ->
        aggValue + newValue.length(), /* Ajout */
    Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>
        as("aggregated-stream-store") /* Nom du stateStore */
        .withValueSerde(Serdes.Long()); /* serde pour la valeur agrégée*/
```



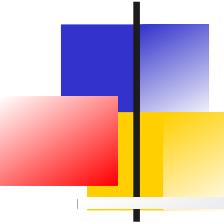
Fenêtrage

Le regroupement par clé garantit que les données sont correctement partitionnées.

Une fois groupées par clé, le fenêtrage permet de sous-regrouper davantage les enregistrements d'une même clé.

Les informations fenêtres valeurs sont stockées dans un **WindowedStateStore**

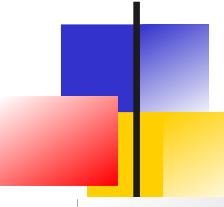
- Dans les opérations de jointure, il est utilisé pour stocker tous les enregistrements reçus jusqu'à présent dans la limite de la fenêtre définie.
- Dans les opérations d'agrégation, il est utilisé pour stocker les derniers résultats d'agrégation par fenêtre.



Types de fenêtres

Le DSL supporte différents types de fenêtres :

- **Hopping time** : Fenêtres de taille fixe qui se chevauchent définie par une taille et un pas d'avancement
- **Tumbling time** : Fenêtres de taille fixe, sans chevauchement et sans espace définies juste par leur taille
- **Sliding time** : Fenêtres de taille fixe se chevauchant et qui fonctionnent sur les différences entre les horodatages d'enregistrement.
2 enregistrements sont dans la même fenêtre si le différence de timestamp est inférieure à la taille de la fenêtre
- **Session** : Fenêtres de taille dynamique, sans chevauchement et pilotées par les données.
Timeout d'inactivité



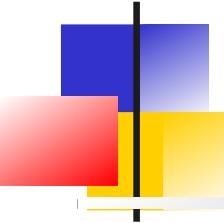
Agrégation avec fenêtrage

```
KGroupedStream<byte[], String> groupedStream = ...;
```

```
// Aggregation avec des fenêtres basculant tous les 5 minutes
KTable<Windowed<String>, Long> timeWindowedAggregatedStream =
    groupedStream.windowedBy(Duration.ofMinutes(5))

    .aggregate(
        () -> 0L,
        (aggKey, newValue, aggValue) -> aggValue + newValue,
        Materialized.<String, Long, WindowStore<Bytes, byte[]>>
            as("time-windowed-aggregated-stream-store")
        .withValueSerde(Serdes.Long())));

```



Reduce

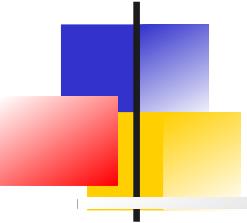
```
KGroupedTable<String, Long> groupedTable = ...;

// KGroupedStream

KTable<String, Long> aggregatedStream = groupedStream.reduce(
    (aggValue, newValue) -> aggValue + newValue /* adder */);

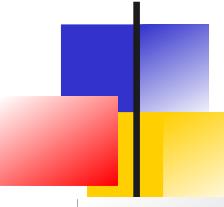
// KgroupedTable, utilisation d'un substractor pour mettre à jour un
// changement de groupe

KTable<String, Long> aggregatedTable = groupedTable.reduce(
    (aggValue, newValue) -> aggValue + newValue, /* adder */
    (aggValue, oldValue) -> aggValue - oldValue /* substractor */);
```



KafkaStream

Concepts
Opérateurs stateless
Opérateurs stateful
ksqIDB

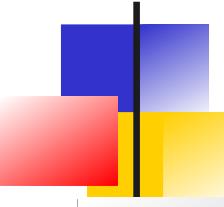


ksqldb

ksqldb fournit donc une couche d'abstraction SQL permettant de manipuler les *Kstream*, *KTable* de KafkaStream

ksqldb convertit les requêtes SQL en tâches Kafka Streams en coulisses.

- Chaque requête ksqldb (par exemple, une CREATE STREAM ou CREATE TABLE) est compilée en un flux Kafka Streams qui est ensuite exécuté dans le cadre de l'infrastructure ksqldb.

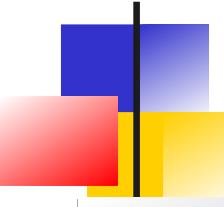


ksqldb

ksqldb est donc un serveur

L'interaction avec le serveur peut se faire via :

- Son *API REST*
- *KsqlDb-cli : Une commande en ligne qui appelle l'API*



Démarrage

Une fois le serveur ksqlDB démarré, on peut se connecter à l'interface de ligne de commande et exécuter des requêtes SQL.

Lister les flux disponibles :

```
SHOW STREAMS;
```

Créer un flux à partir d'un topic :

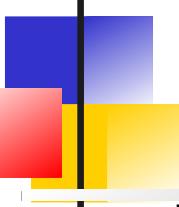
```
CREATE STREAM pageviews (viewtime BIGINT, userid VARCHAR, pageid  
VARCHAR)  
WITH (KAFKA_TOPIC='pageviews', VALUE_FORMAT='JSON');
```

Renseigner un flux :

```
INSERT INTO pageviews (viewtime, userid, pageid) VALUES (1000, 1, 1);
```

Lister les données d'un flux :

```
SELECT * FROM pageviews EMIT CHANGES;
```



Equivalence

L'équivalent de l'opérateur Filter() de KafkaStream est la clause WHERE

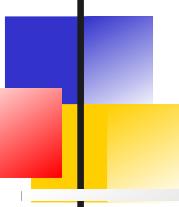
```
CREATE STREAM important_events AS SELECT * FROM source_stream WHERE event_type =  
    'important';
```

L'opérateur Map s'effectue généralement avec des fonctions prédéfinies appliquées sur les colonnes

```
CREATE STREAM uppercase_events AS  
SELECT UCASE(event_type)  
AS event_type_upper  
FROM source_stream;
```

L'opérateur flatMap peut s'implémenter via des fonctions EXPLODE

```
CREATE STREAM exploded_events AS  
SELECT EXPLODE(SPLIT(event_ids, ',')) AS event_id  
FROM source_stream;
```



Equivalence (2)

Les agrégations se font avec GROUP BY et les méthodes d'agrégations classiques

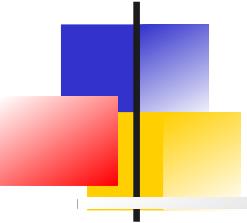
```
SELECT user_id, COUNT(*) AS event_count FROM event_stream GROUP BY user_id;
```

Les jointures :

```
CREATE STREAM enriched_stream AS
  SELECT s.event_id, u.user_name
  FROM events_stream s
  LEFT JOIN users_table u ON s.user_id = u.user_id;
```

Les fenêtres temporelles :

```
SELECT COUNT(*), WINDOWSTART AS start_time, WINDOWEND AS end_time
  FROM events_stream
  WINDOW TUMBLING (SIZE 10 MINUTES)
  GROUP BY event_type;
```



Sécurité

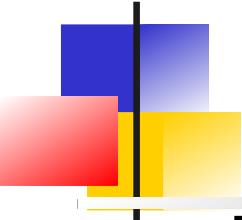
Configuration des listeners

SSL/TLS

Authentification via SASL

ACLs

Quotas

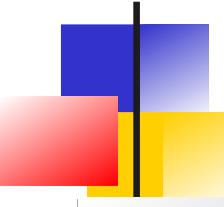


Introduction

Différentes alternatives supportées pour la sécurité :

- Authentification des connections clients vers les brokers via SSL ou SASL
- Authentification des connections entre contrôleurs Kraft OU des brokers vers Zookeeper
- Cryptage des données transférées avec les clients/brokers via TLS/SSL
- Autorisation des opérations read/write/create/delete/... par client
- Possibilité d'intégration avec d'autres mécanismes d'authentification et d'autorisation

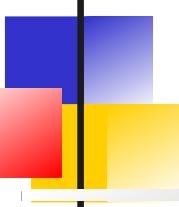
Naturellement, dégradation des performances avec SSL



Listeners

Chaque serveur doit définir l'ensemble des *listeners* utilisés pour recevoir les requêtes des clients ainsi que des autres serveurs.

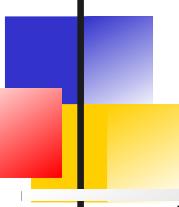
Chaque listener peut être configuré pour authentifier les clients et pour garantir que le trafic entre le serveur et le client est crypté.



Combinaisons des protocoles

Chaque protocole combine une couche de transport (PLAINTEXT ou SSL) avec une couche d'authentification optionnelle (SSL ou SASL) :

- **PLAIN_TEXT** : En clair sans authentification. Ne convient qu'à une utilisation au sein de réseaux privés pour le traitement de données non sensibles
- **SSL** : Couche de transport SSL avec authentification client SSL en option. Convient pour une utilisation dans réseaux non sécurisés car l'authentification client et serveur ainsi que le chiffrement sont prise en charge.
- **SASL_PLAINTEXT** : Couche de transport PLAINTEXT avec authentification client SASL. Ne prend pas en charge le cryptage et convient donc uniquement pour une utilisation dans des réseaux privés.
- **SASL_SSL** : Couche de transport SSL avec authentification SASL. Convient pour une utilisation dans des réseaux non sécurisés.



Configuration des listeners

Les listeners sont déclarés via la propriété ***listeners*** :

{LISTENER_NAME}://:{port}

Ou LISTENER_NAME est un nom descriptif

Exemple :

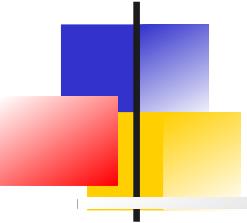
```
listeners=CLIENT://:9092,BROKER://:9095
```

Le protocole utilisé pour chaque listener est spécifié dans la propriété
listener.security.protocol.map

Exemple : `listener.security.protocol.map=CLIENT:SSL,BROKER:PLAINTEXT`

Les protocoles supportés sont : *PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL*

Il faut déclarer les listener utilisés pour la communication inter broker via
inter.broker.listener.name et ***controller.listener.names***

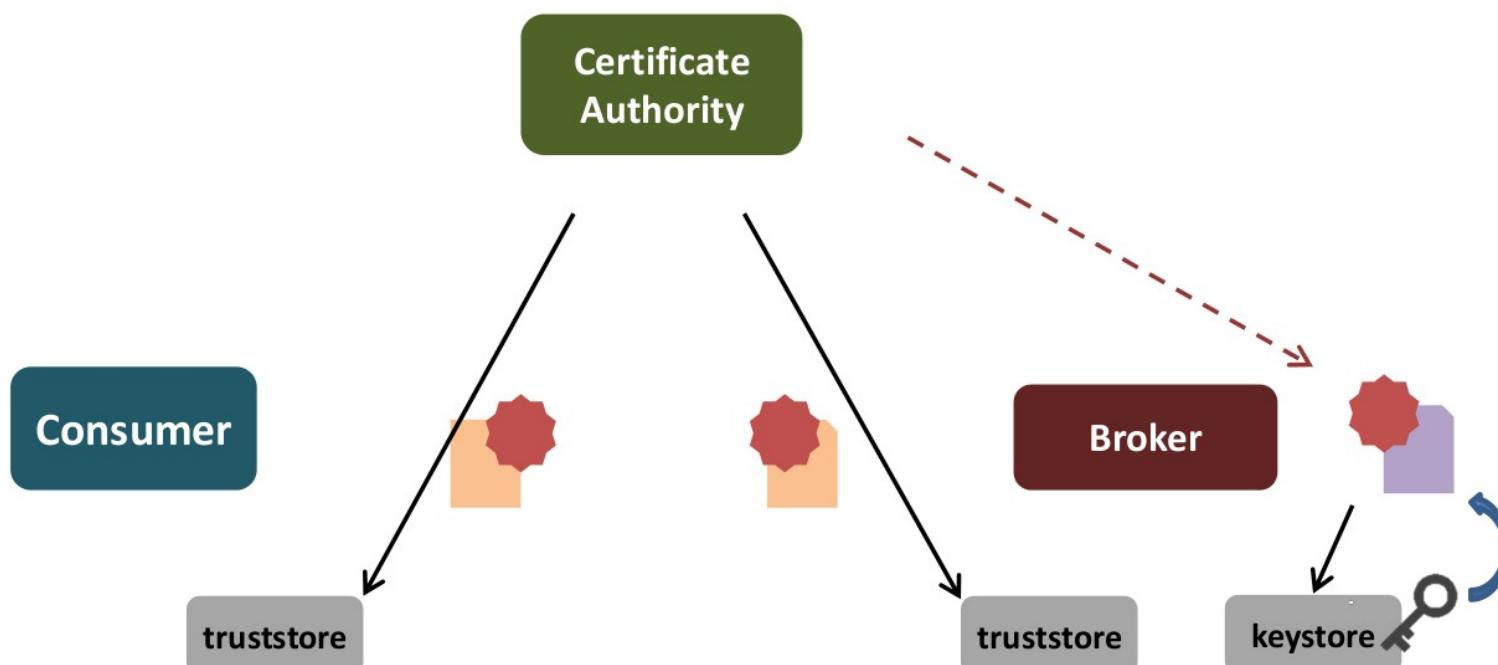


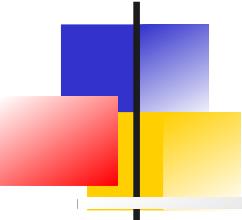
Sécurité

Configuration des listeners
SSL/TLS
Authentification via SASL
ACLs
Quotas

Certificats

SSL pour le cryptage et l'authentification

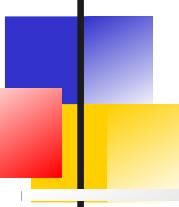




Configuration TLS et Vérification d'hôte

Les certificats des brokers doivent contenir le nom d'hôte du broker en tant que nom alternatif du sujet (SAN) extension ou comme nom commun (CN) pour permettre aux clients de vérifier l'hôte du serveur.

Les certificats génériques utilisant les wildcards peuvent être utilisés pour simplifier l'administration en utilisant le même keystore pour tous les brokers d'un domaine

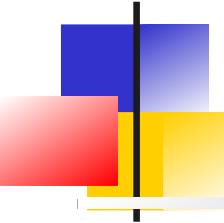


Configuration du broker

server.properties :

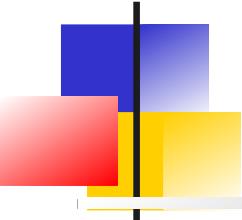
```
listeners=PLAINTEXT://host.name:port,SSL://  
host.name:port
```

```
ssl.keystore.location=/home/ubuntu/ssl/server.keystore.jks  
ssl.keystore.password=servpass  
ssl.key.password=servpass  
ssl.truststore.location=/home/ubuntu/ssl/server.truststore.jks  
ssl.truststore.password=servpass
```



Configuration des clients

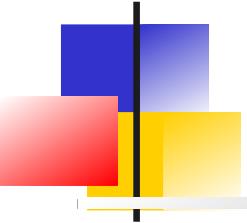
```
security.protocol=SSL  
ssl.truststore.location=/var/private/ssl/  
  client.truststore.jks  
ssl.truststore.password=clipass
```



Authentification des clients via SSL

Si l'authentification du client est requise, un *keystore* doit également être créé et la configuration de *client.properties* doit contenir :

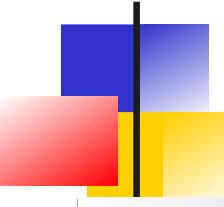
```
ssl.keystore.location=/var/private/client.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
```



Sécurité

Configuration des listeners
SSL/TLS

Authentification via SASL
ACLs
Quotas



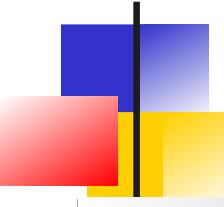
SASL

Simple Authentication and Security Layer pour l'authentification

Kafka utilise JAAS pour la configuration SASL.

Mécanismes:

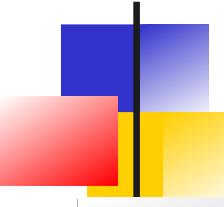
- GSSAPI: Kerberos
- SCRAM-SHA-256, SCRAM-SHA-512: hashed passwords
- PLAIN: username/password en clair
- OAUTHBEARER (Depuis Kafka 3.x)



Configuration JAAS

La configuration JAAS s'effectue :

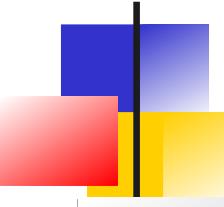
- Soit via un **fichier jaas**
 - La propriété JVM
java.security.auth.login.config référence l'emplacement du fichier:
 - Le fichier contient une section **KafkaServer** pour l'authentification auprès d'un broker
- Soit par la propriété :
listener.name.<listener-name>.<mechanism>.sasl.jaas.config



Fichier JAAS

L'exemple suivant définit 2 utilisateurs admin et alice qui pourront accéder au broker et l'identité avec laquelle le broker initiera les requêtes inter-broker

```
KafkaServer {  
    org.apache.kafka.common.security.plain.PlainLoginModule  
    required  
    username="admin"  
    password="admin-secret"  
    user_admin="admin-secret"  
    user_alice="alice-secret";  
};
```

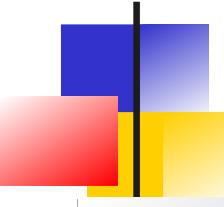


Propriété *sasl.jaas.config*

La configuration JAAS peut également s'effectuer dans les propriétés Kafka

L'exemple suivant configure le listener **SASL_SSL** utilisant le mécanisme **scram-sha-256**

```
listener.name.sasl_ssl.scram-sha-256.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="admin" \
    password="admin-secret";
```



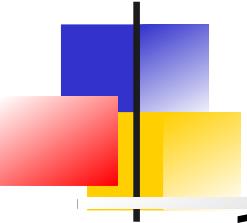
Mécanismes

SASL/Kerberos : Nécessite un serveur (Active Directory par exemple), d'y créer les Principals représentant les brokers. Tous les hosts kafka doivent être atteignables via leur FQDNs

SASL/PLAIN est un mécanisme simple d'authentification par login/mot de passe. Il doit être utilisé avec TLS. Kafka fournit une implémentation par défaut qui peut être étendue pour la production

SASL/SCRAM (256/512) (Salted Challenge Response Authentication Mechanism) : Mot de passe haché stocké dans Zookeeper

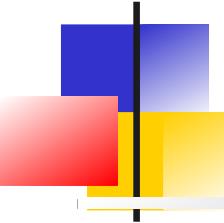
SASL/OAUTHBEARER : Basé sur oAuth2 mais pas adapté à la production



SASL PLAIN

1. Configurer le fichier JAAS
2. Passer l'emplacement du fichier de configuration JAAS en tant que paramètre JVM à chaque broker
 - Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
3. Configurer SASL dans *server.properties*

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
```



Configuration du client

Créer le fichier Jaas

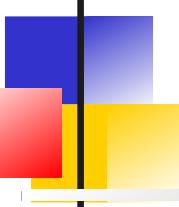
```
KafkaClient {  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
        username="alice"  
        password="alice-secret"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/client_jaas.conf
```

client.properties

```
security.protocol=SASL_SSL  
sasl.mechanisms=PLAIN
```



SASL / PLAIN en production

Doit être utilisé avec SSL

Possibilité d'éviter de stocker des mots de passe clairs en configurant des *callback handler* qui obtiennent le nom d'utilisateur et le mot de passe d'une source externe via :

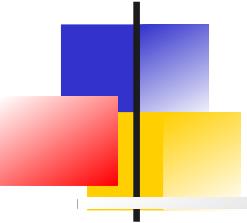
sasl.server.callback.handler.class

et

sasl.client.callback.handler.class.

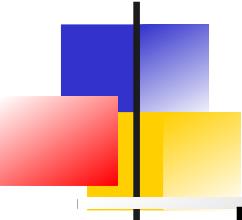
Possibilité d'intégrer des *callback handler* utilisant des serveurs d'authentification externes pour la vérification du mot de passe via

sasl.server.callback.handler.class



Sécurité

Configuration des listeners
SSL/TLS
Authentification via SASL
ACLs
Quotas



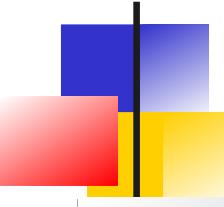
Introduction

Les brokers Kafka gèrent le contrôle d'accès à l'aide d'une API définie par l'interface `org.apache.kafka.server.authorizer.Authorizer`

L'implémentation est configurée via la propriété : **`authorizer.class.name`**

Kafka fournit 2 implémentations

- Pour Zookeeper,
`kafka.security.authorizer.AclAuthorizer`
- En kraft mode
`org.apache.kafka.metadata.authorizer.StandardAuthorizer`



Contrôleurs

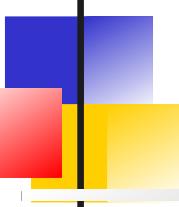
Dès lors que les ACLs sont activées, les contrôleurs doivent s'identifier pour communiquer.

Par exemple pour SASL_PLAIN :

```
controller.listener.names=CONTROLLER
listener.security.protocol.map=CONTROLLER:SASL_SSL, ...
sasl.mechanism.controller.protocol=PLAIN
```

Le fichier JAAS contient une section pour le contrôleur préfixé avec le nom du listener

```
controller.KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="admin"
    password="admin-secret"
    user_admin="admin-secret";
};
```



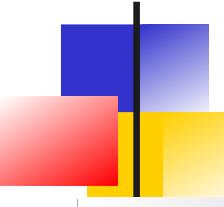
Autorisation

Les ACLs sont stockées dans les méta-données gérées par les contrôleurs et peuvent être gérées par l'utilitaire **kafka-acls.sh**

Chaque requête Kafka est autorisée si le *KafkaPrincipal* associé à la connexion a les autorisations pour effectuer l'opération demandée sur les ressources demandées.

Les règles peuvent être exprimées comme suit :

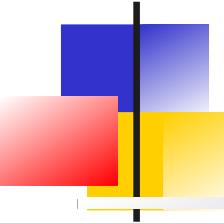
Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP



Exemple

Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1

```
bin/kafka-acls.sh --bootstrap-servers  
localhost:9092 --add --allow-principal  
User:Bob --allow-principal User:Alice --  
allow-host 198.51.100.0 --allow-host  
198.51.100.1 --operation Read --operation  
Write --topic Test-topic
```



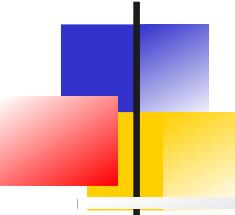
Ressources et opérations

Chaque ACL consiste en :

- **Type de ressource** : *Cluster | Topic | Group | TransactionalId*
- **Type de pattern** : *Literal | Prefixed*
- **Nom de la ressource** : Possibilité d'utiliser les wildcards
- **Opération** : *Describe | Create | Delete | Alter | Read | Write | DescribeConfigs | AlterConfigs*
- **Type de permission** : *Allow | Deny*
- **Principal** : De la forme `<principalType>:<principalName>`
Exemple : *User:Alice, Group:Sales, User :**
- **Host** : Adresse IP du client, * si tout le monde

Exemple Complet :

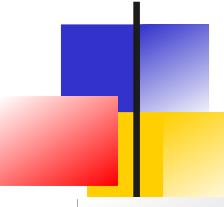
User:Alice has Allow permission for Write to Prefixed Topic:customer from 192.168.0.1



Règles

AclAuthorizer autorise une action s'il n'y a pas d'ACL de DENY qui corresponde à l'action et qu'il y a au moins une ACL ALLOW qui correspond à l'action.

- L'autorisation Describe est implicitement accordée si l'autorisation Read, Write, Alter ou Delete est accordée.
- L'autorisation Describe Configs est implicitement accordée si l'autorisation AlterConfigs est accordée.



Permissions clientes

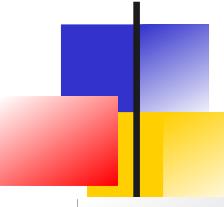
Brokers : **Cluster:ClusterAction** pour autoriser les requêtes de contrôleur et les requêtes de fetch des répliques.

Producteurs simples : **Topic:Write**

- idempotents sans transactions : **Cluster:IdempotentWrite**.
- Transactionnels : **TransactionId:Write** à la transaction et **Group:Read** pour que les groupes de consommateurs valident les offsets.

Consommateurs : **Topic:Read** et **Group:Read** s'ils utilisent la gestion de groupe ou la gestion des offsets.

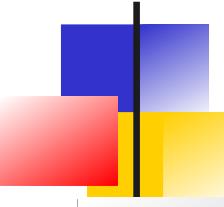
Clients admin : **Create, Delete, Describe, Alter, DescribeConfigs, AlterConfigs** .



Exceptions

2 options de configuration permettant d'accorder un large accès aux ressources permet de simplifier la mise en place d'ACL à des clusters existants :

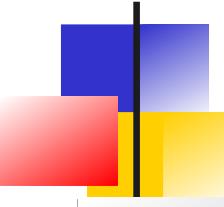
- ***super.users*** : Permet de définir les utilisateurs ayant droit à tout
- ***allow.everyone.if.no.acl.found=true*** : Tous les utilisateurs ont accès aux ressources sans ACL.



Exemple

Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1

```
bin/kafka-acls.sh --authorizer-properties
zookeeper.connect=localhost:2181 --add --
allow-principal User:Bob --allow-principal
User:Alice --allow-host 198.51.100.0 --
allow-host 198.51.100.1 --operation Read
--operation Write --topic Test-topic
```

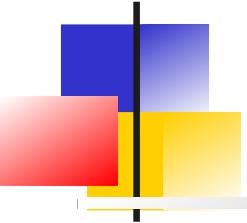


Audit

Les brokers peuvent être configurés pour générer des traces d'audit.

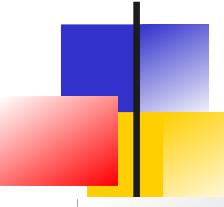
La configuration s'effectue dans
conf/log4j.properties

- Le fichier par défaut est *kafka-authorizer.log*
- Le niveau INFO trace les entrées pour chaque refus
- Le niveau DEBUG pour chaque requête acceptée



Sécurité

Configuration des listeners
SSL/TLS
Authentification via SASL
ACLs
Quotas



Introduction

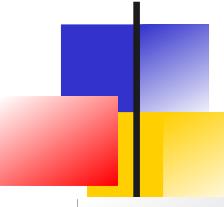
Kafka permet d'appliquer des quotas sur les requêtes pour contrôler les ressources du broker utilisées par les clients.

Deux types de quotas peuvent être appliqués par groupe de client :

- Les quotas de bande passante réseau définissent des seuils de débit
- Les quotas de taux de requête définissent les seuils d'utilisation du processeur en pourcentage du réseau et des threads d'E/S
- Quotas du taux de connexion par IP

L'identité du client correspond au KafkaPrincipal dans un cluster sécurisé ou la propriété applicative client-id.

Tous les clients ayant la même identité partagent leur configuration de quotas

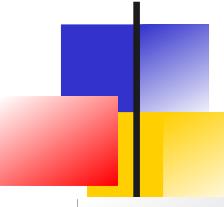


Quota de bande passante

Le seuil de débit d'octets pour chaque groupe de clients.

Chaque groupe peut publier/consommer un maximum de X octets/sec par broker avant d'être limités.

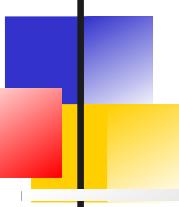
```
kafka-configs.sh --bootstrap-server $BOOT --alter --  
add-config  
'producer_byte_rate=1024,consumer_byte_rate=1024' --  
entity-type users --entity-name <authenticated-  
principal> --command-config /tmp/client.properties
```



Taux de requêtes

Définis en pourcentage de temps sur les threads d'I/O et de réseau de chaque broker dans une fenêtre temporelle.

```
kafka-configs.sh --bootstrap-server $BOOT --  
alter --add-config 'request_percentage=150'  
--entity-type users --entity-name big-time-  
tv-show-host --command-config  
/tmp/client.properties
```



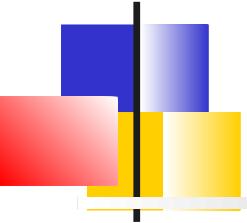
Application des quotas

Lorsqu'un broker constate une violation de quota, il calcule une estimation du délai nécessaire pour ramener le client sous son quota.

Le broker peut alors :

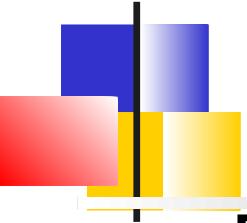
- Répondre au client avec une demande de délai
- Désactiver le canal de la socket

Selon le client (récent ou non), le client peut soit respecter ce délai, soit l'ignorer.



Frameworks Java

Vert.x
Microprofile Messaging
Spring Kafka
Spring Cloud Stream

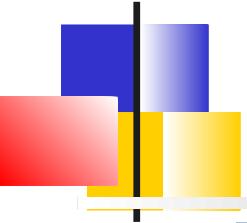


Introduction

De nombreux frameworks permettent une intégration avec Kafka

Ils facilitent la production/consommation de messages, la configuration des niveaux de fiabilité des topics et proposent des modèles de programmation réactif

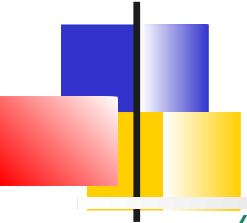
Généralement, ils implémentent la boucle de polling du consommateur



Eclipse Vert.x

Vert.x se définit comme une boîte à outil dédiée aux applications réactive basées sur une JVM

Vert.x propose des clients réactifs pour des APIs Web, des Bds, ... et des systèmes de messagerie comme Kafka



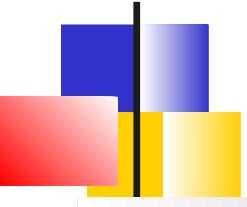
Vert.x Consommateur

```
// Création du consommateur : identique à KafkaConsumer API
...
// Définition du traitement
consumer.handler(record -> {
    System.out.println("Processing key=" + record.key() + ",value=" + record.value() +
        ",partition=" + record.partition() + ",offset=" + record.offset());
});

// S'abonner à plusieurs topics
Set<String> topics = new HashSet<>();
topics.add("topic1");
topics.add("topic2");
consumer.subscribe(topics);

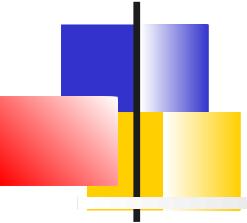
// Avec une regex
Pattern pattern = Pattern.compile("topic\\d");
consumer.subscribe(pattern);

// or un simple topic
consumer.subscribe("a-single-topic");
```



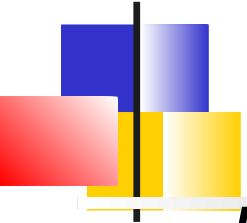
Vert.x Producteur

```
for (int i = 0; i < 5; i++) {  
  
    // Round-robin sur les partitions de destination  
    KafkaProducerRecord<String, String> record =  
        KafkaProducerRecord.create("test", "message_" + i);  
  
    // Envoi asynchrone  
    producer.send(record).onSuccess(rMeta ->  
        System.out.println(  
            "Message " + record.value() + " écrit sur" +  
            rMeta.getTopic() +  
            ", partition=" + rMeta.getPartition() +  
            ", offset=" + rMeta.getOffset()  
        )  
    );  
}
```



Frameworks Java

Vert.x
Microprofile Messaging
Spring Kafka
Spring Cloud Stream



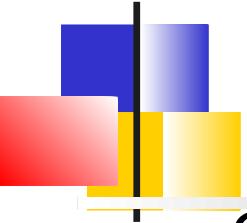
Eclipse MicroProfile

Eclipse MicroProfile est un sous-ensemble de la spécification *Jakarta EE (JavaEE)* dédié aux micro-services déployés dans le cloud.

La spécification inclut **MP Reactive Messaging** qui fournit une abstraction d'un Message Broker.

Les frameworks implémentant cette API proposent des intégrations avec Kafka.

Par exemple, la librairie réactive *SmallRye*, présente dans *Quarkus*, *Wildfly*, *Open Liberty*, propose une intégration Kafka (qui s'appuie sur Eclipse Vert.x)

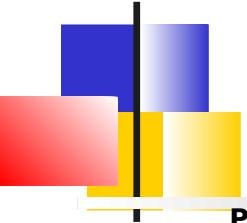


MP Reactive Messaging

Objectif : Fournir un support pour le messaging asynchrone basé sur Reactive Streams, indépendamment du message Broker.

Concepts :

- Les applications échangent des **messages** : Payload et méta-données.
Les messages sont acquittés après traitement
- Les messages transitent dans des **canaux** (channels).
Les applications se connectent aux canaux.
- Certains canaux sont internes à l'application d'autres sont associés à des système de messagerie sous-jacent via des **connecteurs (connector)**.



Exemple *SmallRye*

Producteur

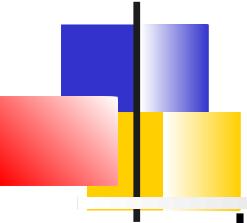
```
@ApplicationScoped
public class KafkaPriceProducer {
    private Random random = new Random();

    @Outgoing("prices")
    public Multi<Double> generate() {
        // It emits a price every second
        return Multi.createFrom().ticks().every(Duration.ofSeconds(1))
            .map(x -> random.nextDouble());
    }
}
```

Consommateur

```
@ApplicationScoped
public class KafkaPriceMessageConsumer {

    @Incoming("prices")
    public CompletionStage<Void> consume(Message<Double> price) {
        // Traitement
        // Puis Commit du offset
        return price.ack();
    }
}
```

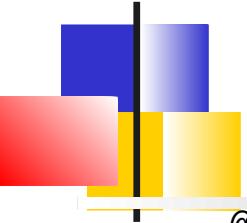


Envoi avec *Emitter*

La signature des méthodes de `@Outgoing` ne permet pas le passage de paramètre.

L'autre façon d'envoyer des messages est de se faire injecter par le framework un bean ***Emitter***.

Le bean propose la méthode `send()` qui retourne un `CompletionStage`, terminé lorsque le message est acquitté.

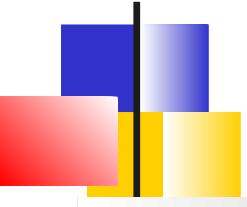


Envoi avec *Emitter*

```
@Path("/prices")
public class PriceResource {

    // La configuration par défaut associe le canal "price-create"
    // au topic Kafka "price-create"
    // La surcharge de la config par défaut s'effectue
    // via application.properties
    @Inject
    @Channel("price-create") Associé au topic price-create
    Emitter<Double> priceEmitter;

    @POST
    @Consumes(MediaType.TEXT_PLAIN)
    public void addPrice(Double price) {
        // Exception si nack
        CompletionStage<Void> ack = priceEmitter.send(price);
    }
}
```



Connecteurs

Les **connecteurs** sont responsables d'associer un canal à un *sink* ou *source* de messages spécifique à une technologie

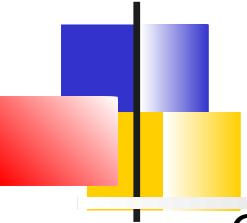
La configuration s'effectue via *MP Config*

Elle suit la structure suivante :

```
mp.messaging.incoming.[channel-name].[attribute]=[value]  
mp.messaging.outgoing.[channel-name].[attribute]=[value]  
mp.messaging.connector.[connector-name].[attribute]=[value]
```

Exemples :

```
mp.messaging.outgoing.prices-out.connector=smallrye-kafka  
mp.messaging.outgoing.prices-out.topic=prices
```



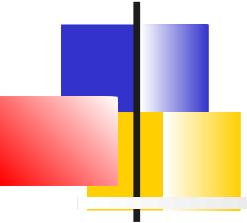
Configuration Kafka

Canal d'entrée (Incoming)

- *topic, key.deserializer, value.deserializer, group.id, auto.offset.reset, partitions*
- *fetch.min.bytes, poll-timeout, batch*
- *enable.auto.commit, commit-strategy*
- *retry, retry-attempts, retry-max-wait, failure-strategy*
-

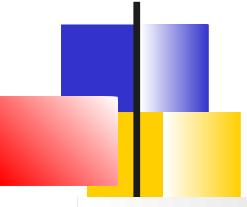
Canal de sortie (Outgoing)

- *topic, key, key.deserializer, value.deserializer,*
- *max-inflight-messages, retries*
- *acks, waitForWriteCompletion*
- ...



Frameworks Java

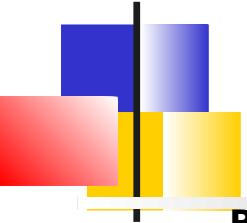
Vert.x
Microprofile Messaging
Spring Kafka
Spring Cloud Stream



Spring Boot

L'écosystème *SpringBoot* propose des starters permettant l'intégration avec Kafka :

- ***org.springframework.kafka:spring-kafka***:
Production/Consommation d'enregistrements Kafka
- ***org.apache.kafka :kafka-streams***:
Intégration de Kafka Streams.
- ***org.springframework.cloud :spring-cloud-stream***:
Abstraction pour développer des micro-services basés sur les évènements (Compatible avec Apache Kafka, RabbitMQ ou Solace PubSub+ via des binders)
équivalent à *MicroProfile Messaging*



Exemple *spring-kafka*

Producteur :

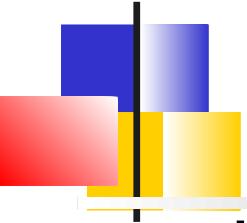
```
@RestController
public class Controller {

    @Autowired
    private KafkaTemplate<Object, Object> template;

    @PostMapping(path = "/send/foo/{what}")
    public void sendFoo(@PathVariable String what) {
        this.template.send("topic1", new Foo1(what));
    }
}
```

Consommateur :

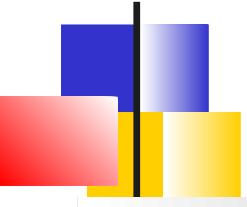
```
@KafkaListener(id = "fooGroup", topics = "topic1")
public void listen(Foo2 foo) {
    logger.info("Received: " + foo);
    if (foo.getFoo().startsWith("fail")) { throw new RuntimeException("failed"); }
    this.new SimpleAsyncTaskExecutor().execute(
        () -> System.out.println("Hit Enter to terminate..."))
};
```



Apports *SpringKafka*

Le starter *SpringKafka* apporte :

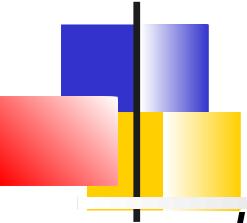
- Les concepts Spring Ioc et DI
- Les auto-configurations SpringBoot
- Un classe utilitaire *KafkaTemplate* pour envoyer les messages
- **@*KafkaListener*** sur des POJOs pour recevoir les messages
- Des similarités avec les autres intégrations de *MessageBroker*, en particulier RabbitMQ et JMS



Configuration des topics

Si un bean *KafkaAdmin* est disponible (automatique avec SpringBoot), il est possible de définir des beans **NewTopic** qui permettent la création automatique de topic

```
@Bean
public NewTopic topic2() {
    return TopicBuilder.name("thing2")
        .partitions(10)
        .replicas(3)
        .config(TopicConfig.COMPRESSION_TYPE_CONFIG, "zstd")
        .build();
}
```

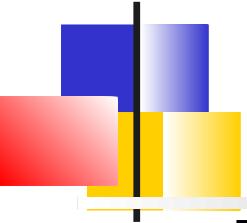


KafkaAdmin et AdminClient

KafkaAdmin fournit les méthodes pour créer et examiner des topics : *createOrModifyTopics()*, *describeTopics()*

On peut également accéder directement à la classe *KafkaAdmin* pour des fonctionnalités plus avancées

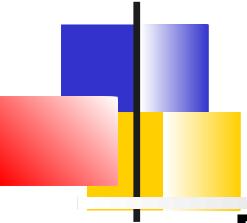
```
@Autowired  
private KafkaAdmin admin;  
  
...  
AdminClient client =  
    AdminClient.create(admin.getConfigurationProperties());  
...  
client.close()
```



Production de messages

Différents objets *Template* sont fournis pour l'émission de messages

- **KafkaTemplate** encapsulant un *KafkaProducer*
- **RoutingKafkaTemplate** permet de sélectionner un *KafkaProducer* en fonction du nom du *Topic*
- **ReplyingKafkaTemplate** permettant des interactions Request/Response
- **DefaultKafkaProducerFactory** pouvant être utilisé pour gérer plus finement les templates

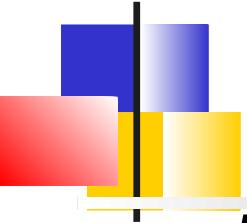


Consommation de messages

Pour consommer des messages, il est nécessaire de définir :

- **MessageListenerContainer** : Définit le modèle de threads des listeners
- **MessageListener** : Mode de réception des messages

L'annotation **@KafkaListener** sur une méthode permet d'associer une méthode d'un POJO à un *MessageListener*

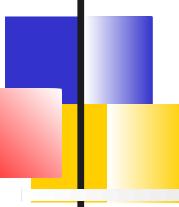


MessageListener

MessageListener est l'interface permettant de consommer les *ConsumerRecords* de Kafka retournés par la boucle *poll*

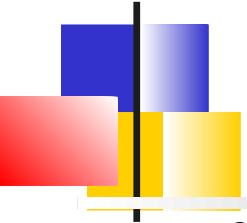
Il offre différentes alternatives :

- Traitement individuel ou par lot
- Commit géré par le container ou manuel via la classe **Acknowledgment**
- Accès au *KafkaConsumer* sous-jacent



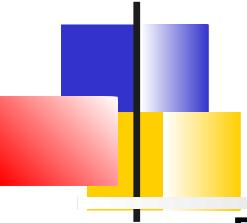
Interfaces MessageListener

```
// Traitement individuel des ConsumerRecord retournée par poll() de Kafka :  
// commit gérées par le container  
MessageListener<K, V> : onMessage(ConsumerRecord<K, V> );  
ConsumerAwareMessageListener<K, V> : onMessage(ConsumerRecord<K, V>,  
Consumer<?, ?> ); }  
  
// Traitement individuel avec commit manuel (Acknowledgment)  
AcknowledgingMessageListener<K, V> : onMessage(ConsumerRecord<K, V>,  
Acknowledgment);  
AcknowledgingConsumerAwareMessageListener<K, V> : onMessage(ConsumerRecord<K,  
V>, Acknowledgment, Consumer<?, ?>);  
  
// Traitement par lot avec commit géré par container  
BatchMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>);  
BatchConsumerAwareMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>,  
Consumer<?, ?>);  
  
// Traitement par lot avec commit manuel  
BatchAcknowledgingMessageListener<K, V> : onMessage(List<ConsumerRecord<K, V>>,  
Acknowledgment);  
BatchAcknowledgingConsumerAwareMessageListener<K, V> :  
onMessage(List<ConsumerRecord<K, V>>, Acknowledgment, Consumer<?, ?>); }
```



Exemple Configuration

```
spring:  
  kafka:  
    consumer:  
      group-id: consumer  
      bootstrap-servers:  
        - localhost:9092  
        - localhost:9192  
      auto-offset-reset: earliest  
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer  
      value-deserializer: org.formation.model.JsonDeserializer  
      enable-auto-commit: true  
  
    listener:  
      concurrency: 3 # Container concurrent  
      log-container-config: true
```



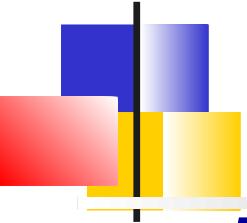
Configuration du traitement d'erreur

Par défaut, le message fautif est sauté (skip) et la consommation continue

Il est possible de configurer des gestionnaires d'exception au niveau container ou listener

- Ils peuvent alors retenter un certain nombre de fois la consommation du message
 - De façon synchrone : les messages suivants sont bloqués pendant les tentatives
 - De façon asynchrone : les messages suivants sont consommés pendant les tentatives. (L'ordre n'est plus respecté)

Il est possible de configurer un déserialiseur Spring gérant les erreurs de sérialisation



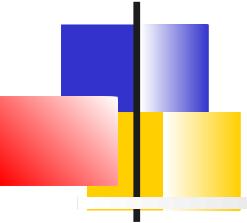
Exemple : *DefaultErrorHandler*

DefaultErrorHandler est utilisé pour rejouer un message en erreur. Le traitement est synchrone

- **N** tentatives sont essayées,
si elles échouent toutes :
 - Soit un simple trace
 - Soit un *DeadLetterPublishingRecoverer* est configuré,
permettant d'envoyer le message vers un topic dead Letter
(même nom avec le suffixe DLT)

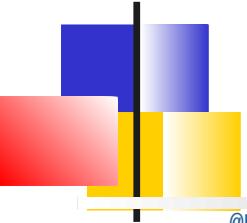
si une réussit : la consommation continue

```
// Configuration automatique dans le container factory grâce à SpringBoot
@Bean
public CommonErrorHandler errorHandler(KafkaTemplate<Object, Object> template) {
    return new DefaultErrorHandler(
        new DeadLetterPublishingRecoverer(template), new FixedBackOff(1000L, 2));
}
```



Frameworks Java

Vert.x
Microprofile Messaging
Spring Kafka
Spring Cloud Stream



cloud-stream : production

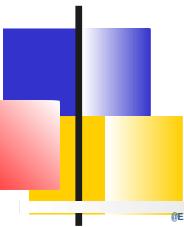
```
@EnableBinding(SampleSource.Source.class)
public class SampleSource {

    private final Log logger = LoggerFactory.getLog(getClass());

    @Bean
    @InboundChannelAdapter(value = Source.SAMPLE, poller = @Poller(fixedDelay = "1000", maxMessagesPerPoll = "1"))
    public MessageSource<String> timerMessageSource() {
        return new MessageSource<String>() {
            public Message<String> receive() {
                logger.info("*****\nAt the Source\n*****");
                String value = "{\"value\":\"hi\"}";
                logger.info("Sending value: " + value);
                return MessageBuilder.withPayload(value).build();
            }
        };
    }

    public interface Source {
        String SAMPLE = "sample-source";

        @Output(SAMPLE)
        MessageChannel sampleSource();
    }
}
```



cloud-stream : consommation

```
@EnableBinding(SampleSink.Sink.class)

public class SampleSink {

    private final Log logger = LogFactory.getLog(getClass());

    // Sink application definition

    @StreamListener(Sink.SAMPLE)

    public void receive(Foo foo) {

        logger.info("*****\nAt the Sink\n*****");

        logger.info("Received transformed message " + foo.getValue() + " of type " + foo.getClass());

    }

    public interface Sink {

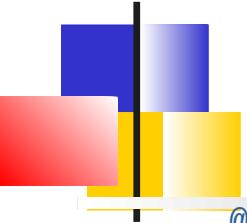
        String SAMPLE = "sample-sink";

        @Input(SAMPLE)

        SubscribableChannel sampleSink();

    }

}
```



cloud-stream : function

@Configuration

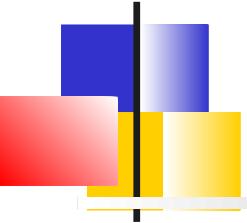
```
public class SampleSinkConfiguarition {

    private final Log logger = LoggerFactory.getLog(getClass());

    // Sink function definition

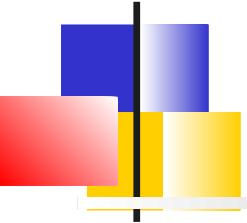
    @Bean
    public Consumer<Message<Foo>> sample-sink() {
        return foo -> {
            logger.info("*****\nAt the Sink\n*****");
            logger.info("Received transformed message " + foo.getValue() + " of type " +
                foo.getClass());
        }
    }

}
```



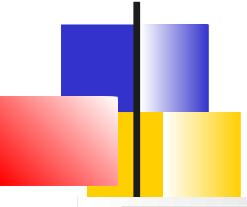
Binding vers topics Kafka

```
spring:  
  cloud:  
    stream:  
      bindings:  
        consumer-in-0 :  # Binding sur le nom de méthode  
          destination : input  
        sample-source:  
          destination: teststock  
        input:  
          destination: teststock  
        output:  
          destination: xformed  
        sample-sink:  
          destination: xformed
```



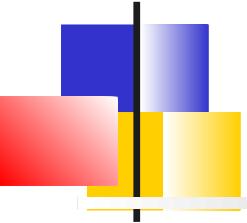
Annexes

Monitoring et JMX



Principaux métriques brokers accessibles via JMX

Métrique	Description	Alerte
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	Nombre de partition sous-répliquée	Si > 0
kafka.controller:type=KafkaController,name=OfflinePartitionsCount	Nombre de partitions qui n'ont pas de leader actif	Si > 0
kafka.controller:type=KafkaController,name=ActiveControllerCount	Nombre de contrôleur actif dans le cluster	Si!= 1
kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce Fetch Consumer FetchFollower}	Nombre de requêtes par seconde, pour produire et récupérer	Si changement significatif
kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	Retard maximal des messages entre les répliques et le leader	
kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	Cadence de shrink des ISR	
kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	Cadence d'expansion des ISR	



Métriques clients

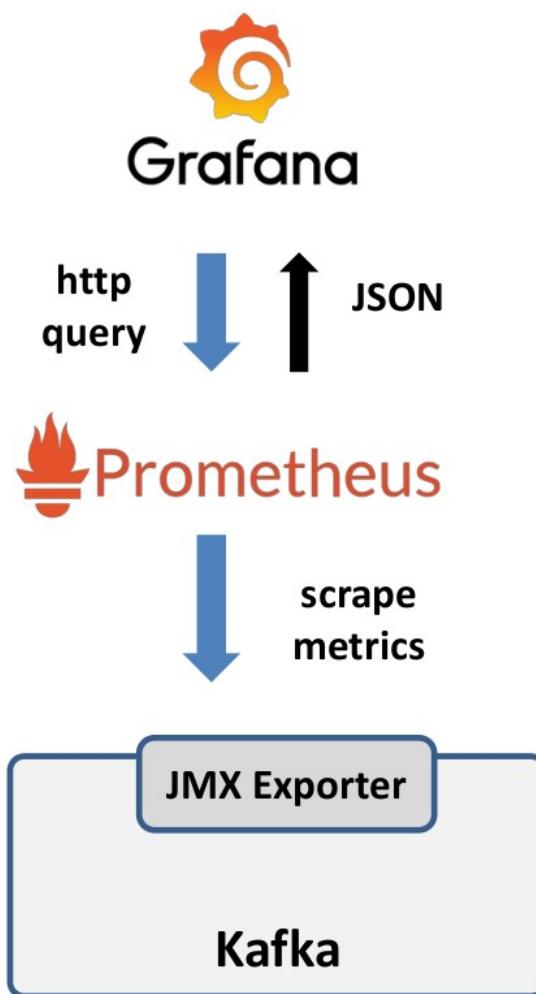
Producteur

Métrique	Description
kafka.producer:type=producermetrics,client-id=(-.w]+),name=io-ratio	Fraction de temps de la thread passé dans les I/O
kafka.producer:type=producer-metrics,client-id=(-.w]+),name=io-wait-ratio	Fraction de temps de la thread passé en attente

Consommateur

Métrique	Description
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=(-.w]+),records-lag-max	Le décalage maximum en termes de nombre d'enregistrements pour n'importe quelle partition

Outil de visualisation

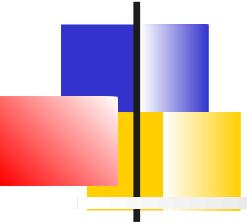


Tdb dispo :

<https://grafana.com/grafana/dashboards/721>

Exportateur JMX

https://github.com/prometheus/jmx_exporter/blob/master/example_configs/kafka-2_0_0.yml



Autres outils

Confluent Control Center

JConsole

Graphite

CloudWatch

DataDog

SMM of Data Flow (Hortonworks)

...