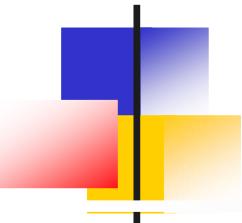


Messagerie distribuée avec Apache Kafka

C# / .NET

David THIBAU – 2024

david.thibau@gmail.com



Agenda

Introduction à Kafka

- Le projet Kafka
- Cas d'usage
- Concepts

Garanties Kafka

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

Cluster Kafka

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

Kafka Stream

- Concepts
- Streamiz
- ksqlDB

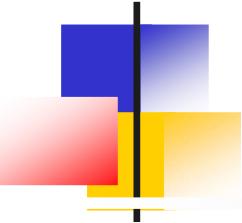
Kafka APIs

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

Sécurité

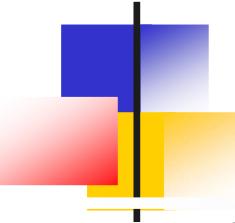
- Configuration des listeners
- SSL/TLS
- Authentification via SASL
- ACLs
- Quotas

Annexes



Introduction à Kafka

Le projet Kafka
Cas d'usage
Concepts



Origine

Initié par *LinkedIn*, mis en OpenSource en 2011

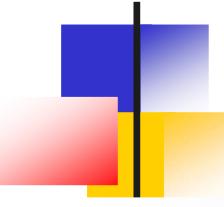
Écrit en *Scala* et *Java*

Au départ, un message broker gérant une file de messages

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

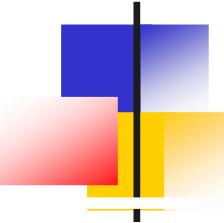
Organisé en cluster, taillé pour le BigData, il est basé sur l'abstraction « d'un journal de commit distribué »

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 !

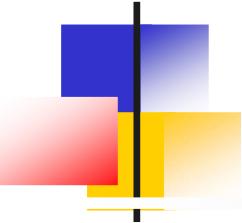


Fonctionnalités

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

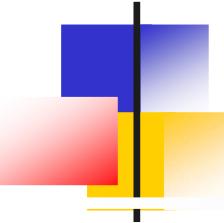
- Gestion des abonnements multiples
- Facilité d'extension du cluster

Très bonne performance

Disponibilité et tolérance aux fautes

Adapté au traitement distribué d'évènements

Intégration avec les autres systèmes



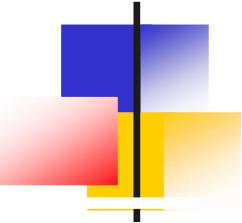
Confluent

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

Mainteneur principal d'Apache Kafka

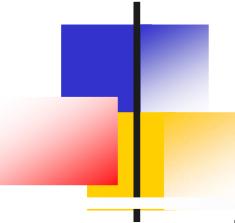
Plate-forme *Confluent* :

- Une distribution de Kafka
- Des librairies clientes
- Fonctionnalités commerciales additionnelles :
Interface d'administration, cloud



Introduction à Kafka

Le projet Kafka
Cas d'usage
Concepts



Kafka vs Message broker traditionnel

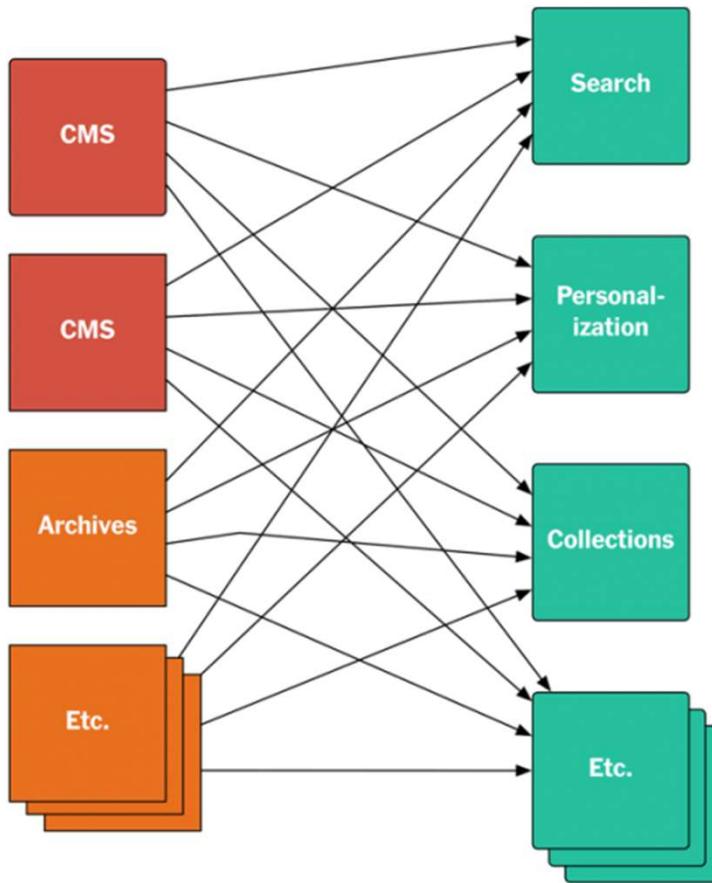
Kafka peut être utilisé comme **message broker** permettant de découpler les cycles de vie des producteurs et des consommateurs

- 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 malgré les défaillances
 - 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

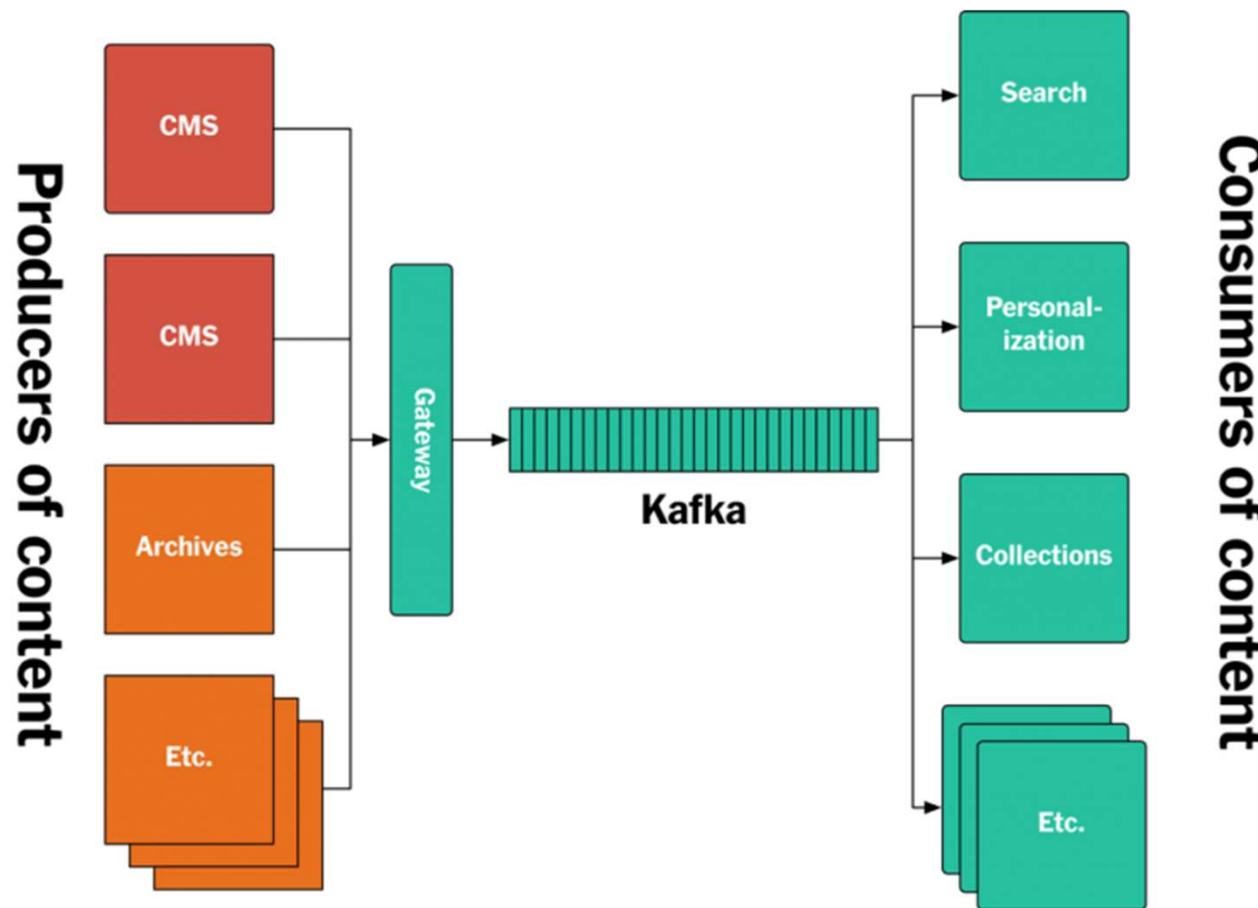
Producers of content

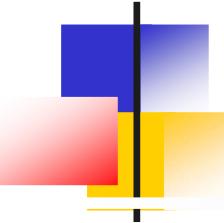


Consumers of content

Exemple ESB (New York Times)

Après





Exemple : micro-services

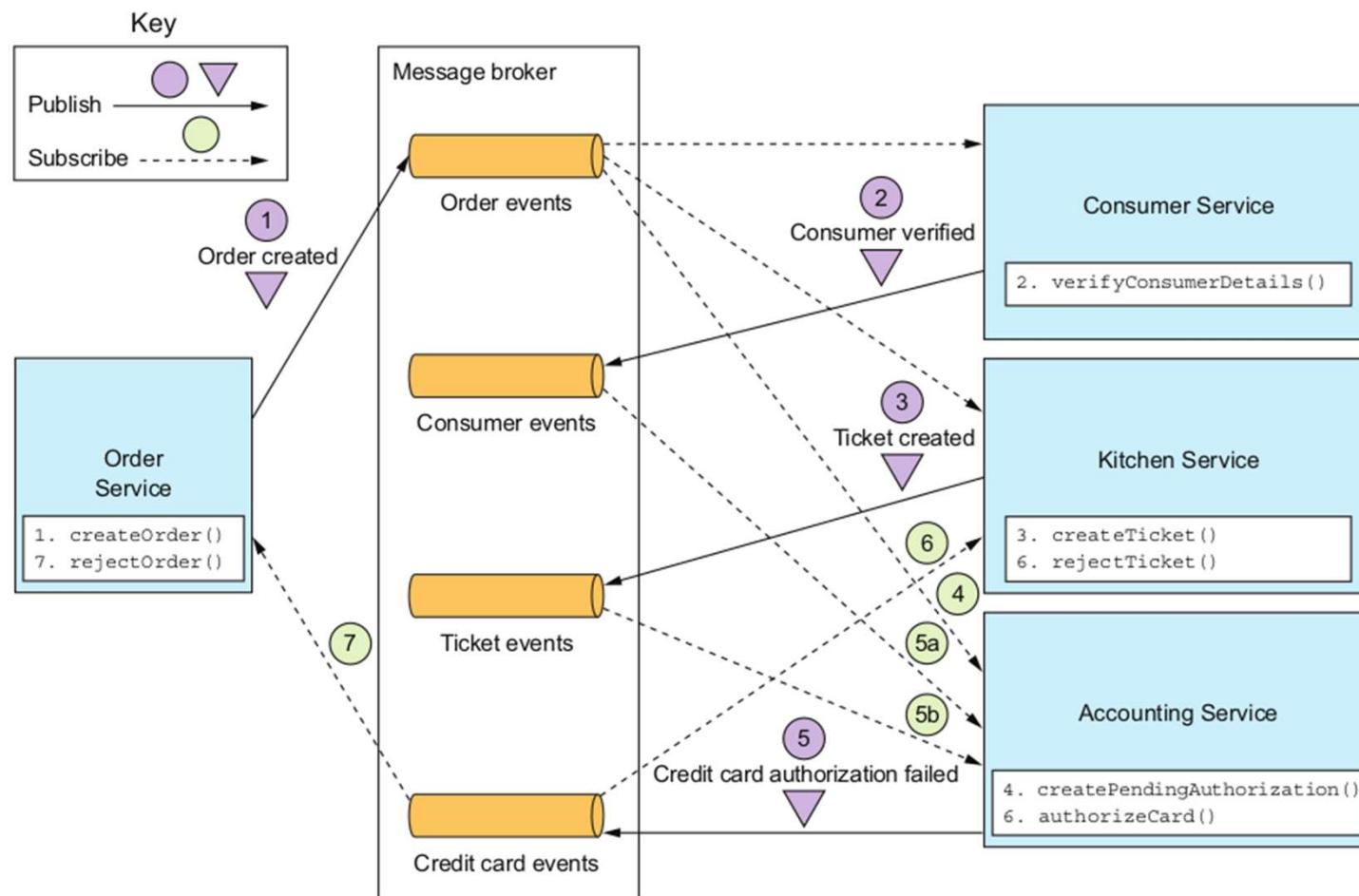
Kafka est souvent utilisé pour permettre des communications asynchrones entre les services d'une architecture micro-services

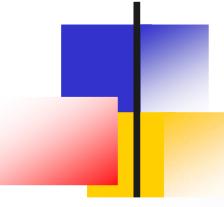
- Permet tous les styles d'interaction :
Requête/Réponse synchrone asynchrone,
One way notification,
Publish and Subscribe, Publish et réponse asynchrones
- Indispensable pour certains patterns micro-services, par exemple SAGA¹ (~ Transaction distribuée)

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

Exemple Micro-services

Message Broker et Design pattern SAGA





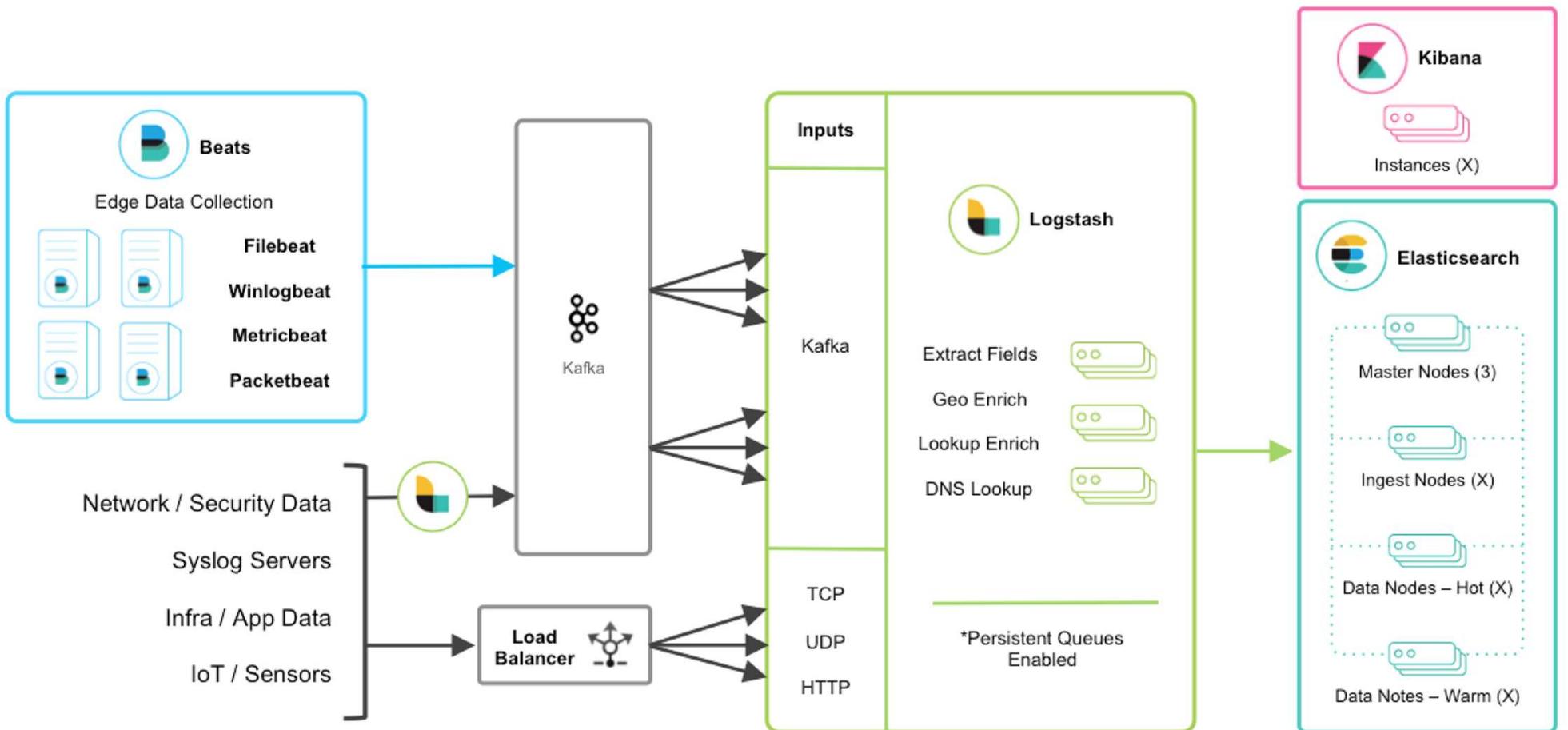
Kafka : Data Buffering

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

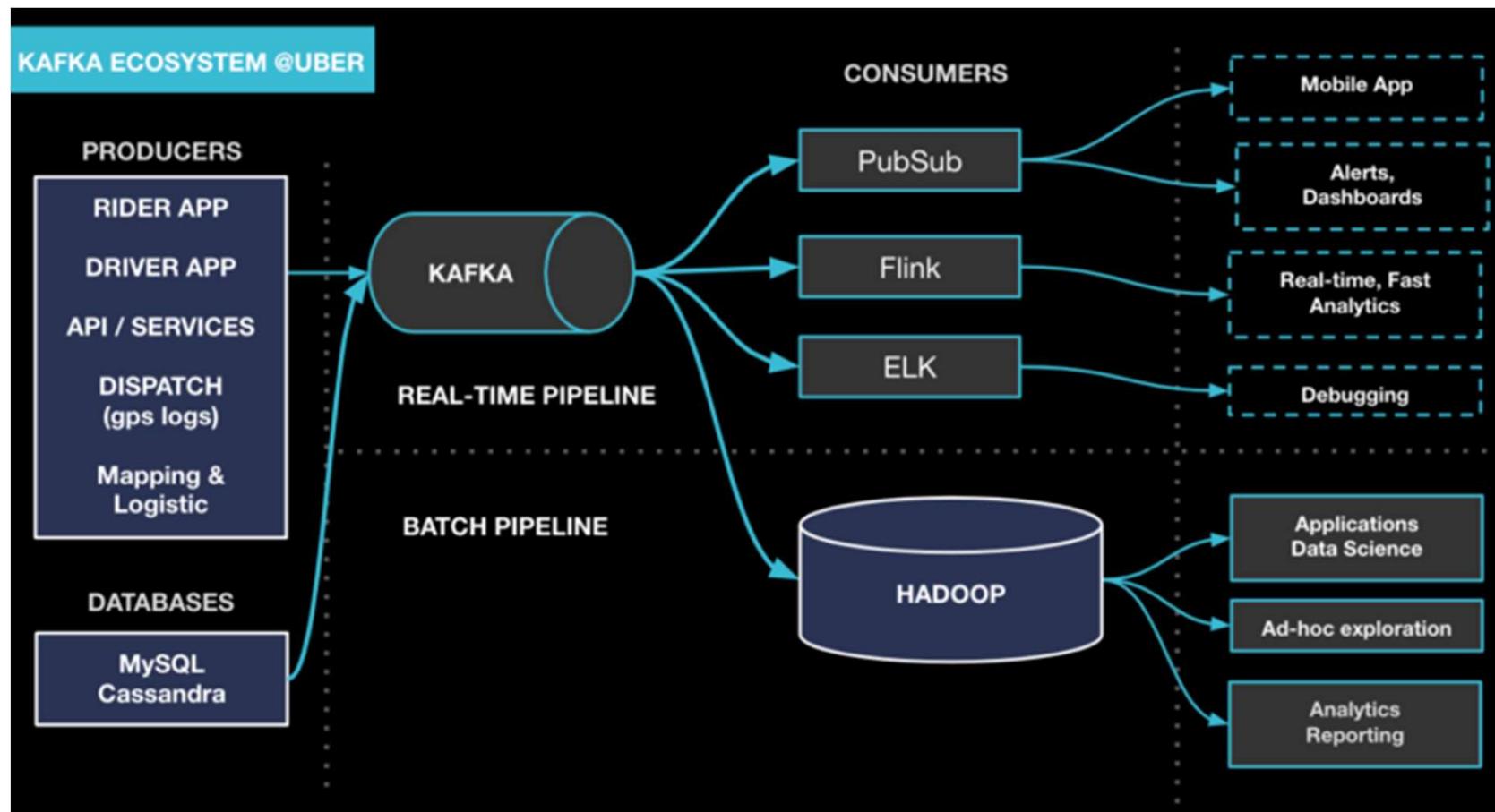
Exemple Architecture ELK

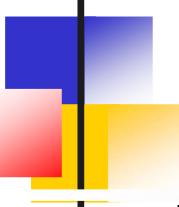
Bufferisation des événements



Ingestion massive de données

Exemple Uber





Architecture Event-driven

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

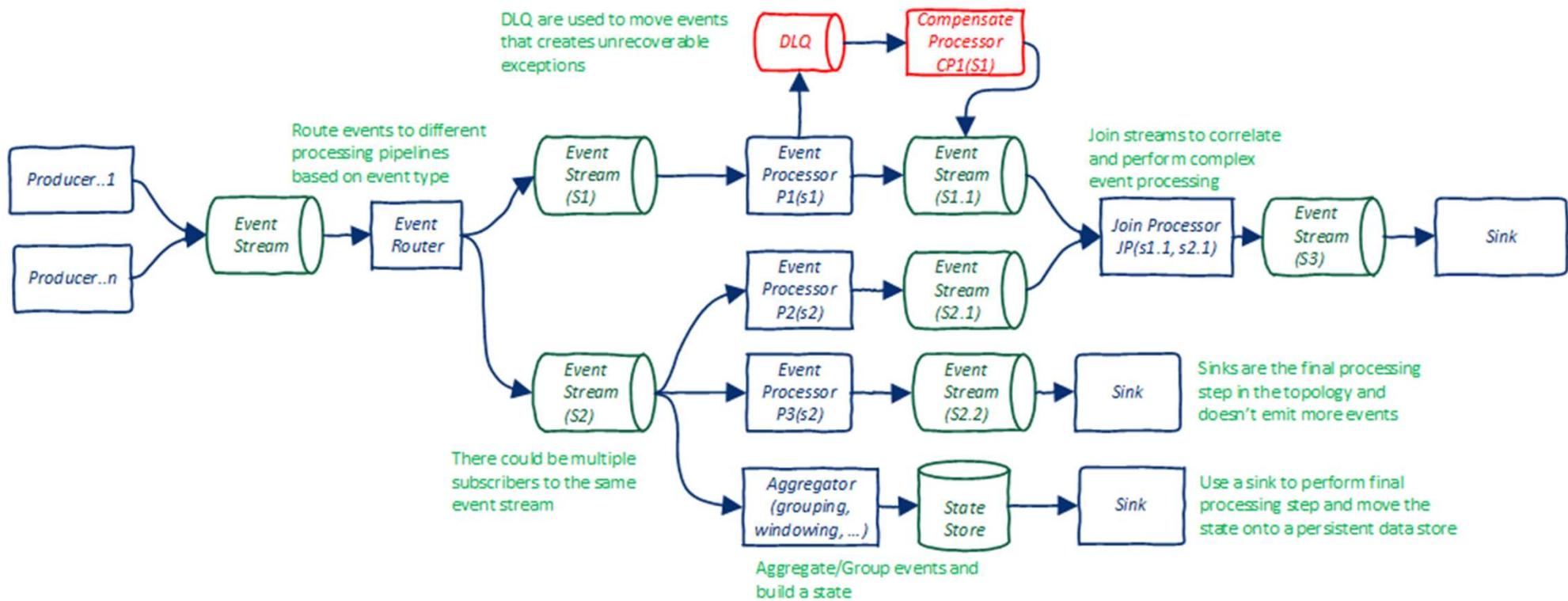
- Produit généralement des architectures plus souples et plus réactives.

Chaque micro-service consomme en continu des événements :

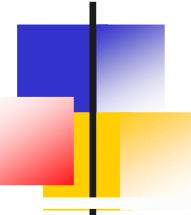
- Lit un ou plusieurs topics Kafka en entrée
- Effectue un traitement
- Écrit vers un ou plusieurs topics de sortie

L'API Kafka Stream et son pendant .NET Streamiz facilitent ce type d'architecture

Event-driven architecture

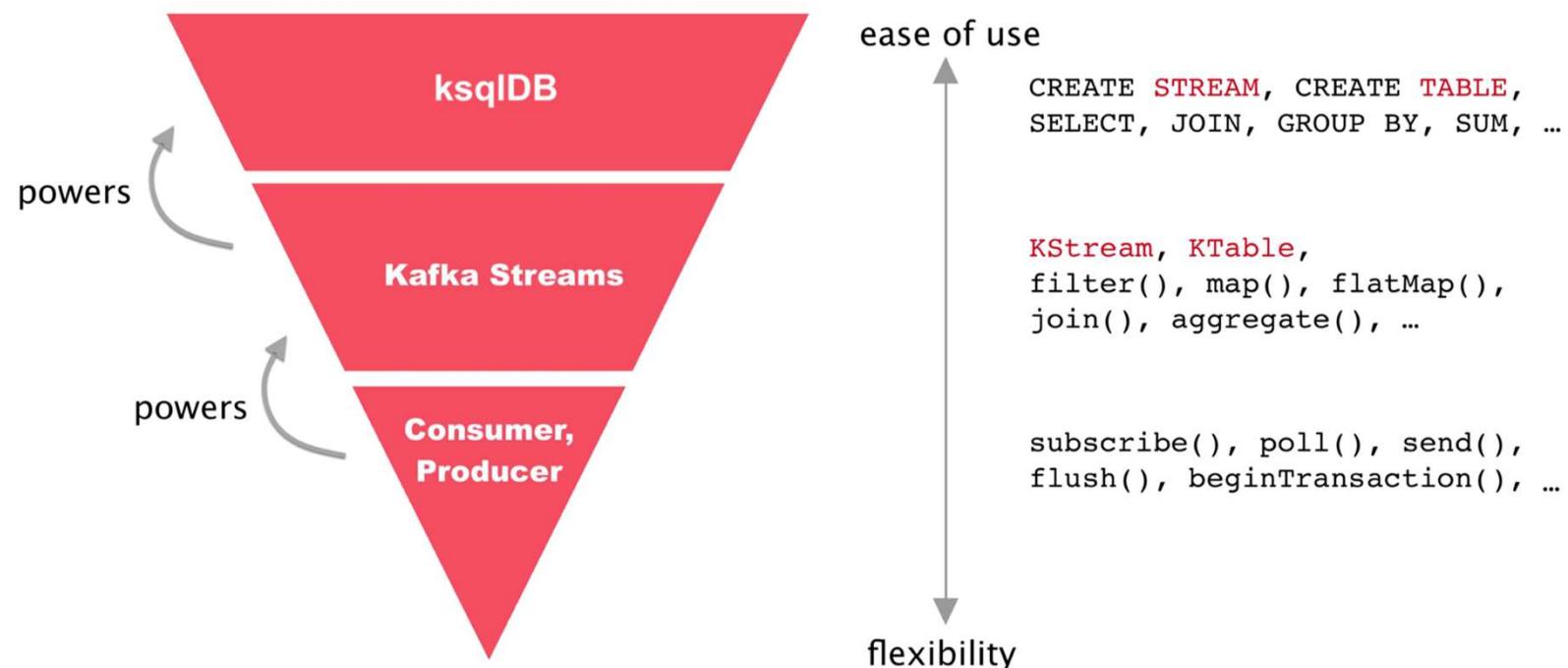


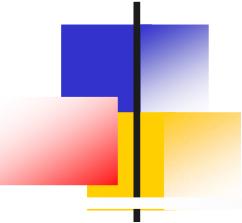
Event Processor – could be a microservice, serverless function, etc – which implements the logic of a particular processing step



ksqldb

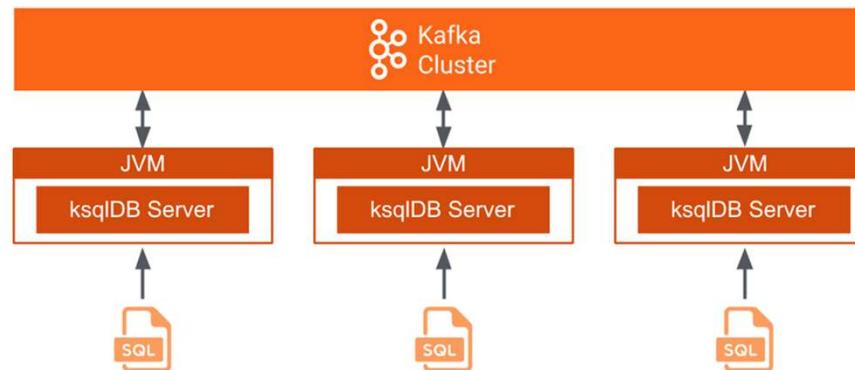
ksqldb est une abstraction au dessus de KafkaStream permettant de bâtir ces applications via des instructions SQL





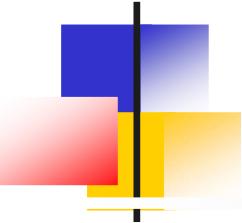
Architecture ksqlDB

ksqlDB Standalone Application (Headless Mode)



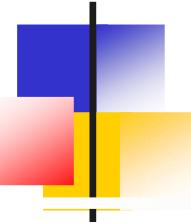
Intégration via

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



Introduction à Kafka

Le projet *Kafka*
Cas d'usage
Concepts

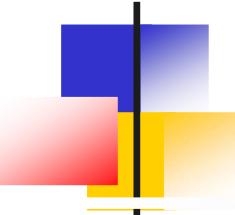


Concepts de base

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 et d'un horodatage.

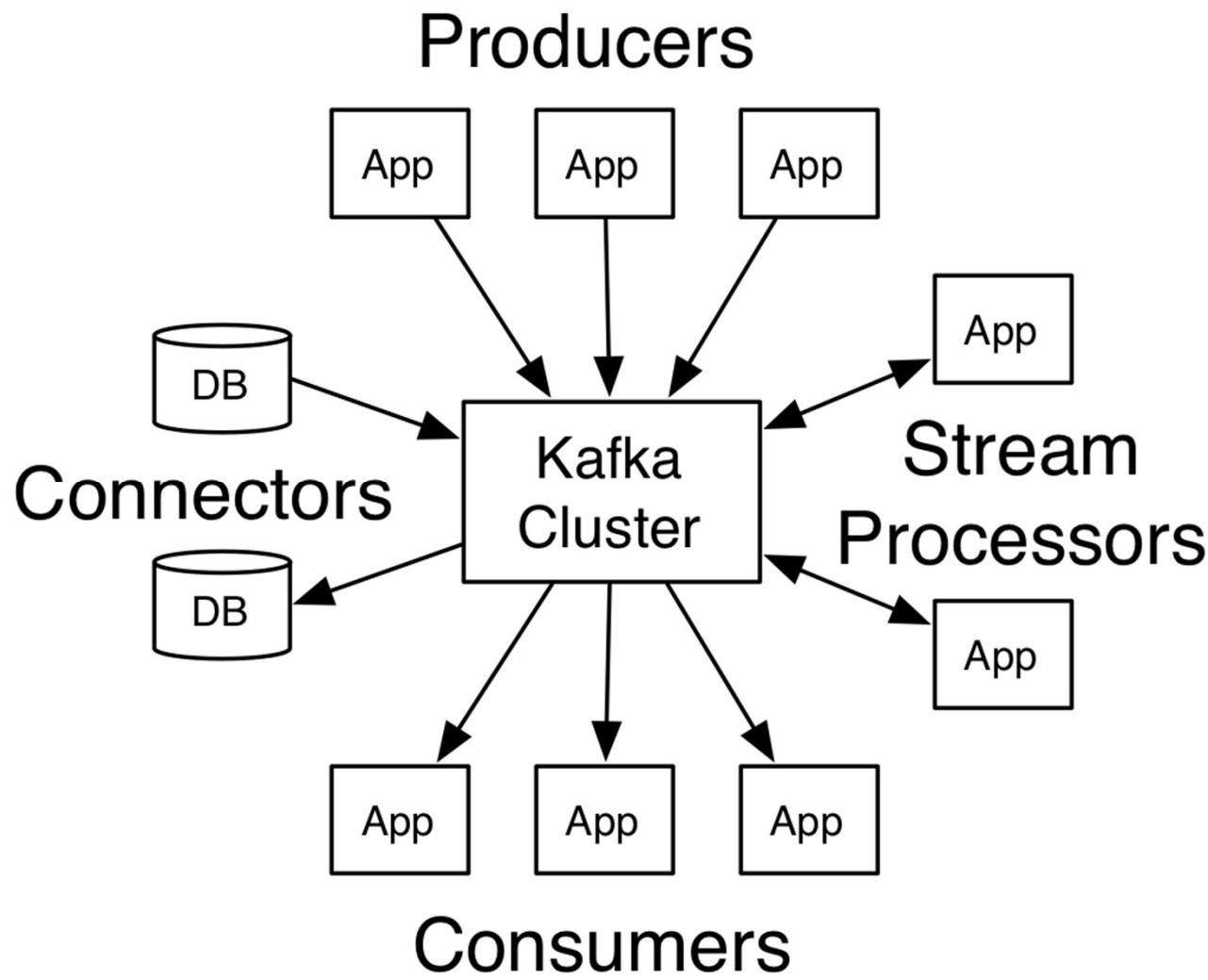


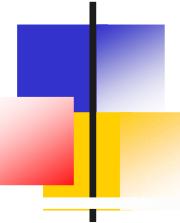
APIs

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





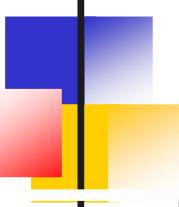
Protocole Client/Serveur

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

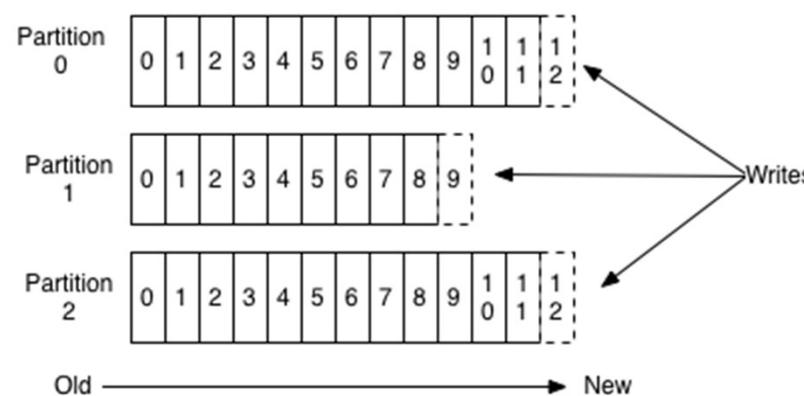
Les *records* sont publiés vers des **topics**.

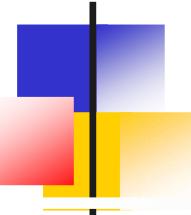
Les *topics* de Kafka peuvent avoir Zéro, Un ou de multiples abonnés

Les topics peuvent être **partitionnés**.

Le cluster Kafka conserve donc un journal partitionné

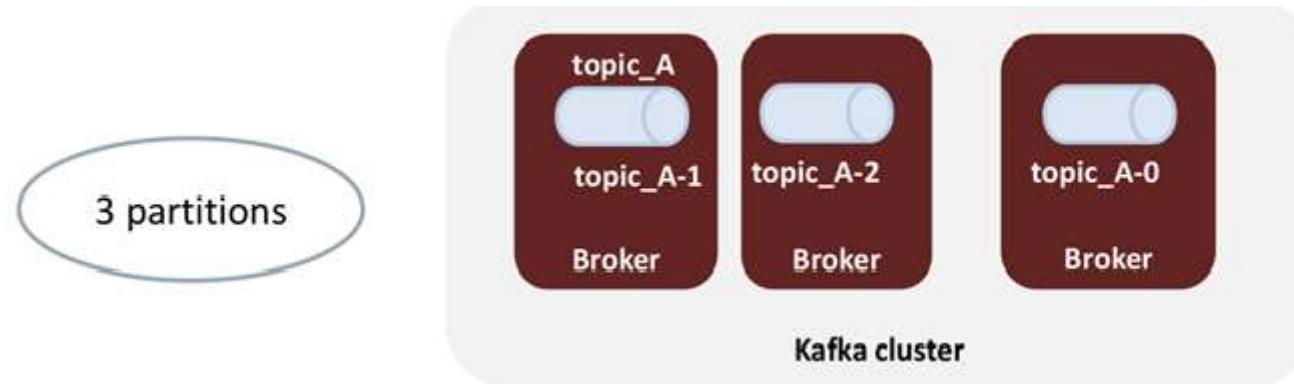
Anatomy of a Topic





Apport des partitions

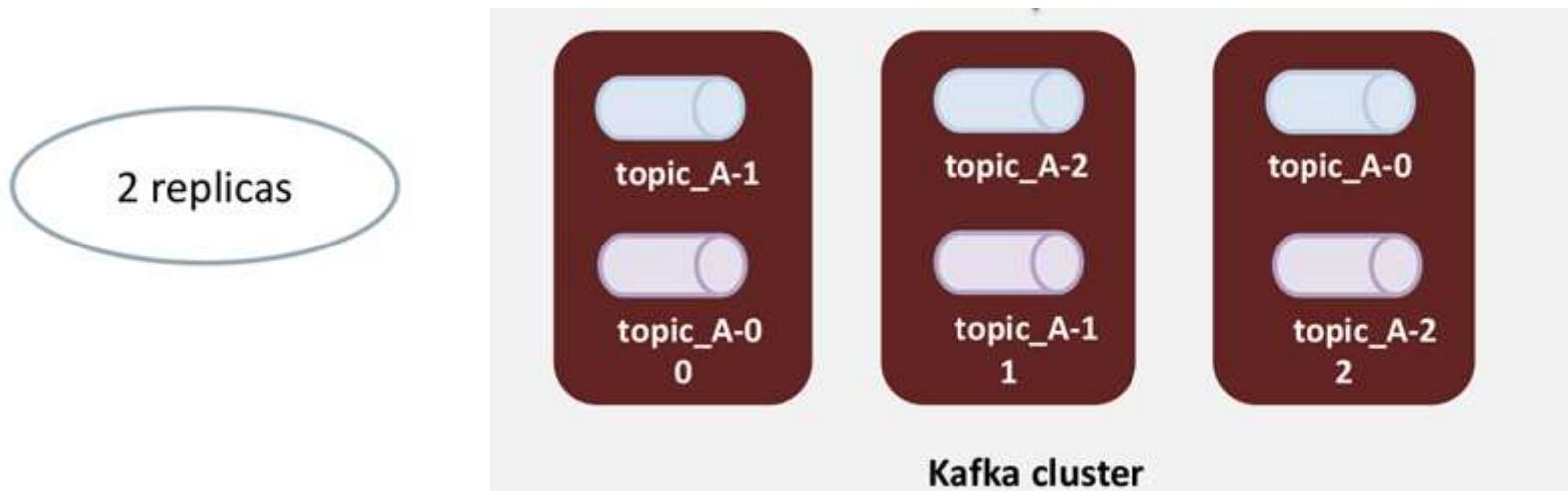
- Les partitions autorisent le parallélisme et augmentent la capacité de stockage en utilisant les capacités disque de chaque nœud.
- L'ordre des messages n'est garanti qu'à l'intérieur d'une partition
- Le nombre de partition est fixé à la création du *topic*

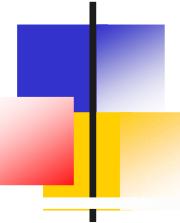


RéPLICATION

Les partitions peuvent être **répliquées**

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





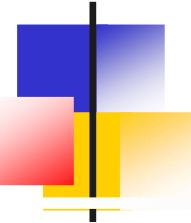
Distribution des partitions

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

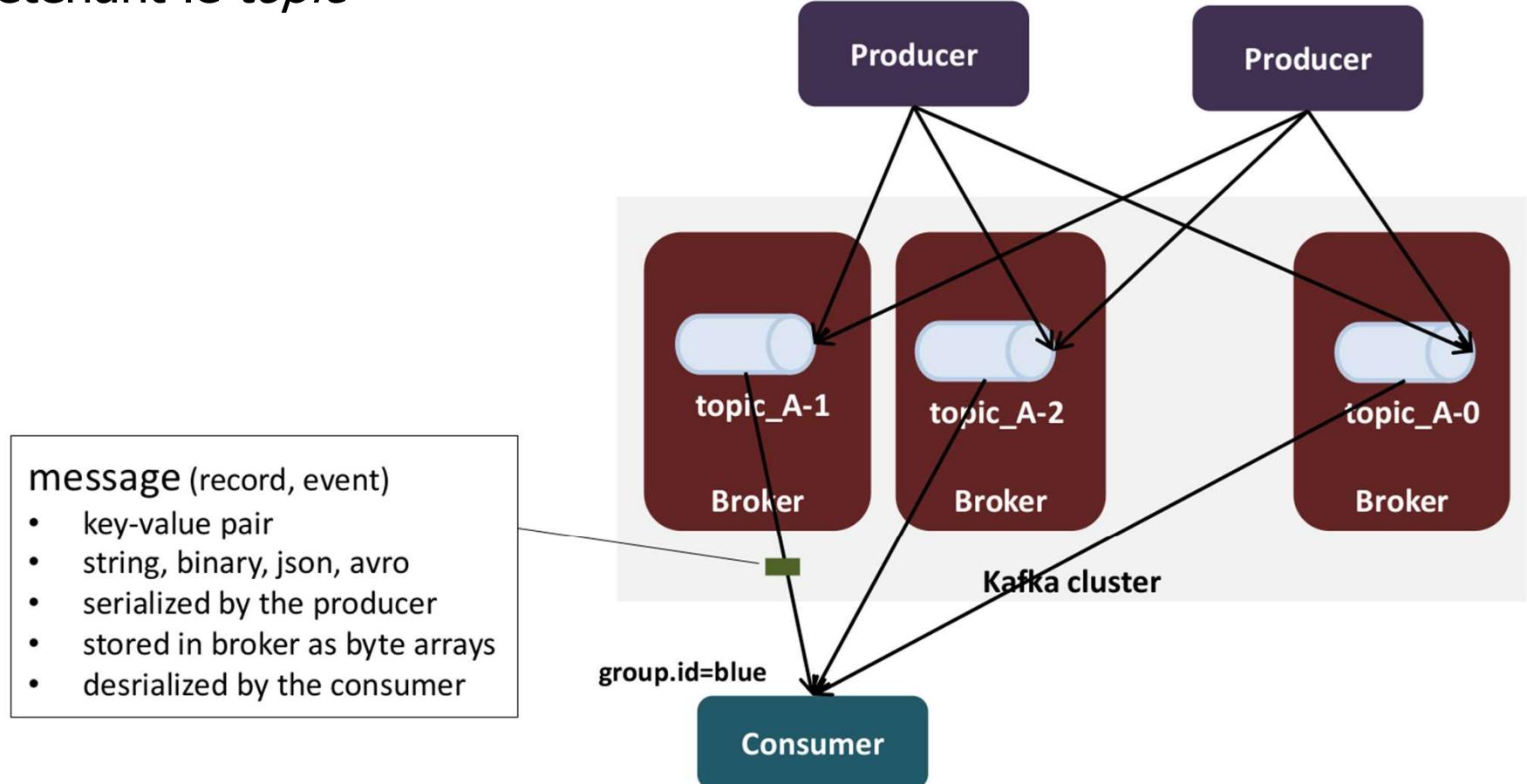
Chaque partition est une séquence **ordonnée et immuable** d'enregistrements.

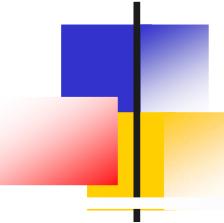
Un numéro d'identification séquentiel nommé **offset** est attribué à chaque enregistrement.

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*



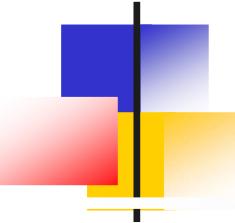


Routing des messages

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

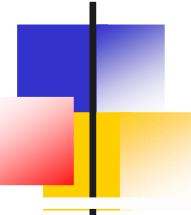


Groupe de consommateurs

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

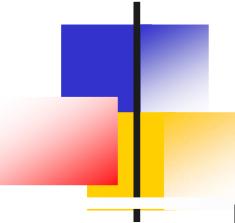


Offset consommateur

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

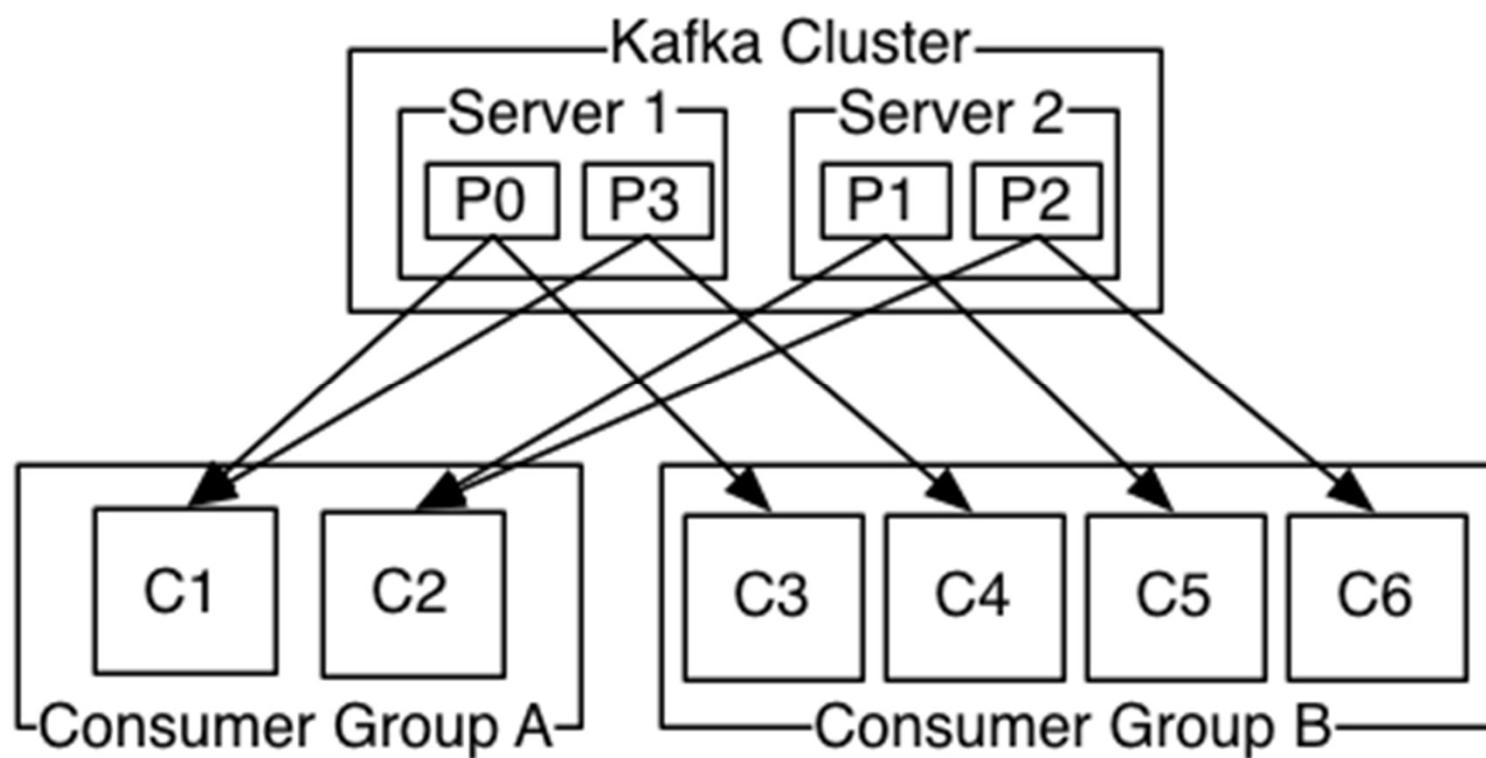
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

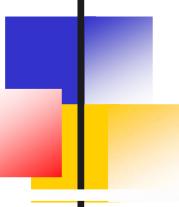
- A tout moment, un consommateur est exclusivement dédié à une partition

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

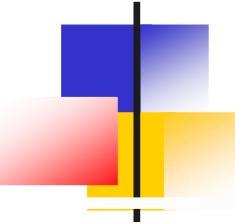




Ordre des enregistrements

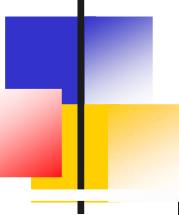
Kafka garantit un ordre total sur les enregistrements d'une partition, mais pas sur les différentes partitions d'un topic.

- 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



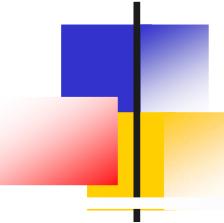
Cluster

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 **contrôleur¹**.
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

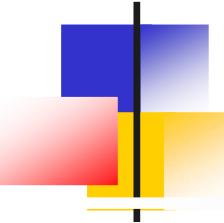
*1. Lors de la présence d'un contrôleur, le cluster s'exécute en mode Kraft
2. Voir annexe Zookeeper*



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.

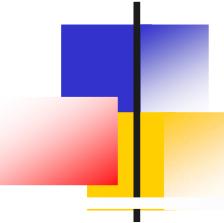


Configuration et démarrage d'un nœud

La distribution Kafka fournit un script de démarrage :
kafka-server-start.sh

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

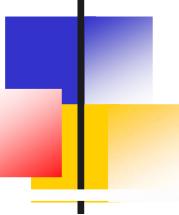
- Dans un fichier ***server.properties***
- Les propriétés peuvent être surchargées par la commande en ligne via l'option
--override



Principales configuration

Les configurations principales sont :

- ***cluster.id*** : Identique pour chaque serveur appartenant au même cluster.
- ***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



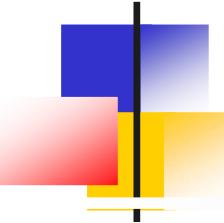
Contrôleurs

Certains processus Kafka sont susceptibles de devenir contrôleur. Ils participent au processus d'élection et de consensus.

- Une majorité des contrôleurs doivent être vivants pour maintenir la disponibilité du cluster
- Il 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***

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



Configuration par défaut des topics

num.partitions : Par défaut 1

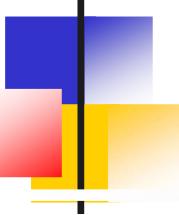
default.replication.factor : Par défaut 1

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



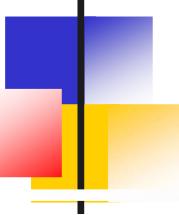
Cluster ID

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 :

bin/kafka-storage.sh random-uuid

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

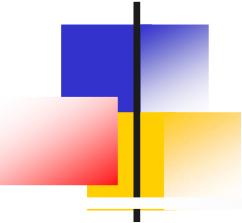


Formatting des répertoires de log

Les répertoires des logs doivent être *formattés*.

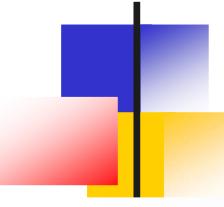
- Le formatting consiste à créer le fichier ***meta.properties*** contenant les identifiants cluster et noeud:

L'outil kafka-storage permet de formatter à partir du fichier de configuration du noeud :
bin/kafka-storage.sh format -t <cluster_id> -c server.properties



Cluster Kafka

Cluster
Distributions / Installation
Utilitaires *Kafka*
Outils graphiques



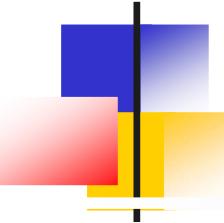
Introduction

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 :

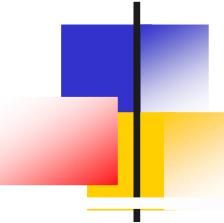
- Binaire chez Apache.
OS recommandé Linux + pré-installation de Java (Support de Java 17 pour 3.1.0)
- Images docker ou packages Helm (Apache depuis peu, Bitnami par exemple)
- Confluent Platform (Téléchargement ou cloud)



Hardware

Afin de sélectionner le matériel :

- Débit de disque : Influence sur les producteurs de messages. SSD si beaucoup de clients
- Capacité de disque : Estimer le volume de message * période de rétention
- Mémoire : Faire en sorte que le système ait assez de mémoire disponible pour utiliser le cache de page.
Pour la JVM, 5Go permet de traiter beaucoup de message
- Réseau : Potentiellement beaucoup de trafic. Favoriser les cartes réseau de 10gb
- CPU : Facteur moins important

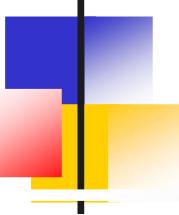


Installation à partir de l'archive

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**



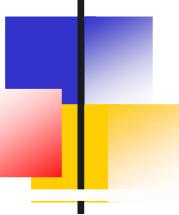
Images bitnami

Images basées sur minideb (minimaliste Debian)

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

Variables d'environnement pour configurer Kafka

- Variables spécifiques bitnami.
Ex : BITNAMI_DEBUG, ALLOW_PLAINTEXT_LISTENER, ...
- Variables d'environnement Kafka préfixée par
KAFKA_CFG
Ex : KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE



Kubernetes

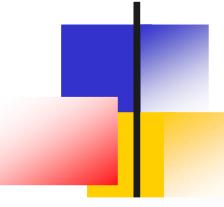
Bitnami propose des packages Helm pour déployer vers Kubernetes ou des clouds

```
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 Kubernetes.



Vérifications de l'installation

Création de topic :

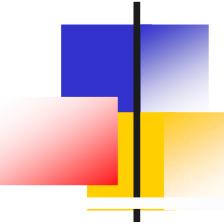
```
bin/kafka-topics.sh --create --bootstrap-server  
localhost:9092 --replication-factor 1 --  
partitions 1 --topic test
```

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 :

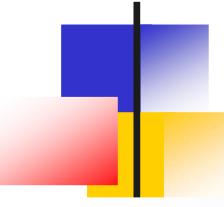
```
bin/kafka-console-consumer.sh --bootstrap-server  
localhost:9092 --topic test --from-beginning  
This is a message  
This is another message
```



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

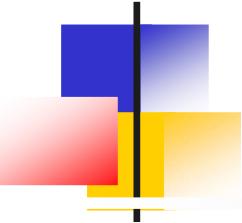


Fichiers de trace

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

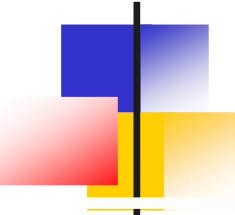
Par défaut sont générés :

- ***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

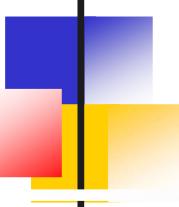


Introduction

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 visualiser un topic

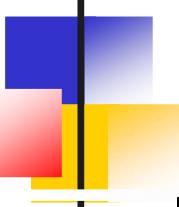
Les valeurs par défaut des topics sont définis dans *server.properties* mais peuvent être surchargées pour chaque topic

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
```

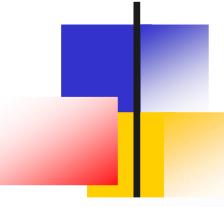


Choix du nombre de partitions

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



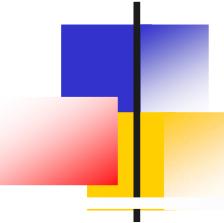
Envoi de message

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

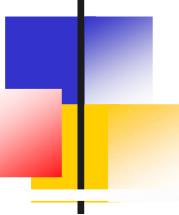


Lecture de message

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

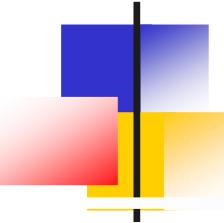
kafka-reassign-partitions.sh permet de gérer les partitions, déplacement sur de nouveau brokers, de nouveaux répertoire, extension de cluster, ...

`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)

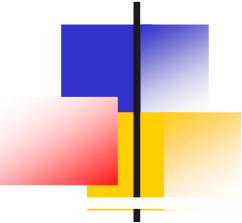
kafka-configs.sh permet des mises à jour de configuration dynamique des brokers

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

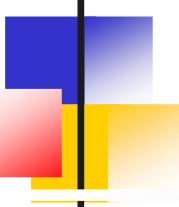
kafka-acls.sh permet de gérer les permissions sur les topics

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 ou à la consultation des topics.

Comme produit gratuit, semi-commerciaux citons :

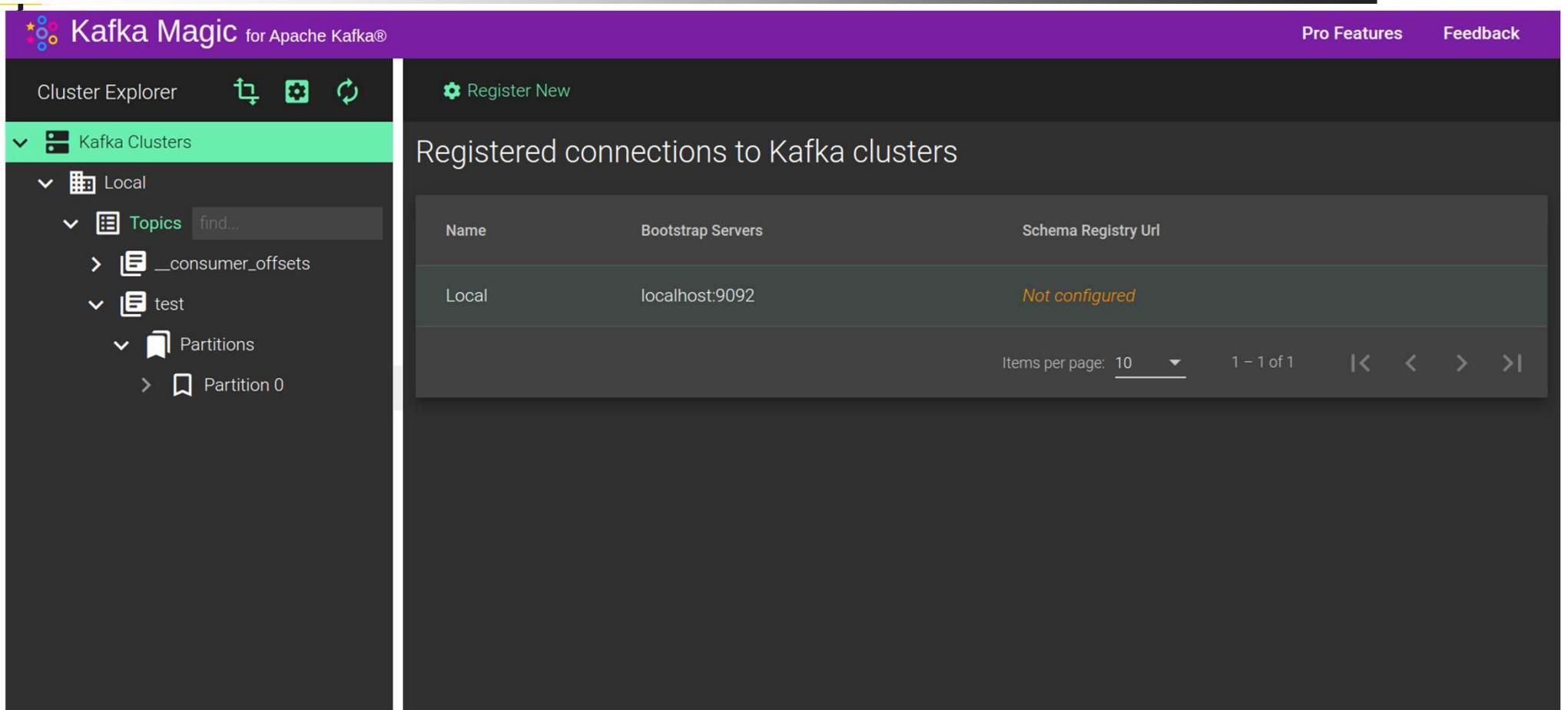
- **Akhq** (<https://akhq.io/>)
- **Kafka Magic** (<https://www.kafkamagic.com/>)
- **Redpanda Console** (<https://docs.redpanda.com/docs/manage/console/>)

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

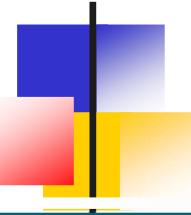
Kafka Magic



The screenshot shows the Kafka Magic interface for Apache Kafka®. The top navigation bar includes "Kafka Magic for Apache Kafka®", "Pro Features", and "Feedback". The left sidebar has a "Cluster Explorer" section with icons for refresh, settings, and a refresh. Below it, the "Kafka Clusters" section is expanded, showing "Local" which is also expanded. Under "Topics", there are entries for "__consumer_offsets" and "test". Under "Partitions", there is an entry for "Partition 0". The main content area displays a table titled "Registered connections to Kafka clusters". The table has columns for "Name", "Bootstrap Servers", and "Schema Registry Url". A single row is present with "Name" set to "Local", "Bootstrap Servers" set to "localhost:9092", and "Schema Registry Url" set to "Not configured". At the bottom of the table, there are pagination controls for "Items per page: 10" and "1 – 1 of 1", along with navigation arrows.

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

Items per page: 10 | 1 – 1 of 1 | < < > >|



akhq

X

0.24.0

my-cluster

Nodes

Topics

Live Tail

Consumer Groups

ACLS

Schema Registry

Settings

Nodes

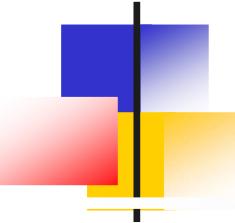
default

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

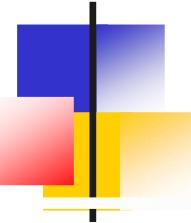
The screenshot shows the Redpanda Console interface. The left sidebar has a dark theme with the Redpanda logo at the top. The 'Topics' option is selected, highlighted with a blue bar. Below it are other options: Overview, Schema Registry, Consumer Groups, Security, Quotas, Connectors, and Reassign Partitions. The main content area is titled 'Cluster > Topics'. It displays two large numbers: '1' for Total Topics and '1' for Total Partitions. Below this, there's a 'Create Topic' button in a red rounded rectangle. A table lists the single topic 'test': Name (test), Partitions (1), Replicas (1), CleanupPolicy (delete), and Size (145 B). To the right of the table is a checkbox labeled 'Show internal topics'. At the bottom, a pagination bar shows 'Total 1 items' and '50 / page'. The footer contains social media icons for GitHub, Docker, Twitter, and LinkedIn, along with the text 'Redpanda Console (Platform Version v23.1) (built May 03, 2023) FC2BF3B'.

Name	Partitions	Replicas	CleanupPolicy	Size
test	1	1	delete	145 B



APIs

Producer API
Consumer API
Schema Registry
Connect API
Admin API

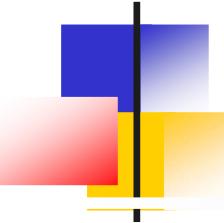


Introduction

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 ?



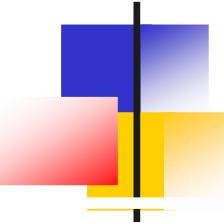
Dépendances

Java

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

.NET

confluent-kafka-dotnet disponible via [NuGet](#) :
S'appuie sur le client C ***librdkafka***

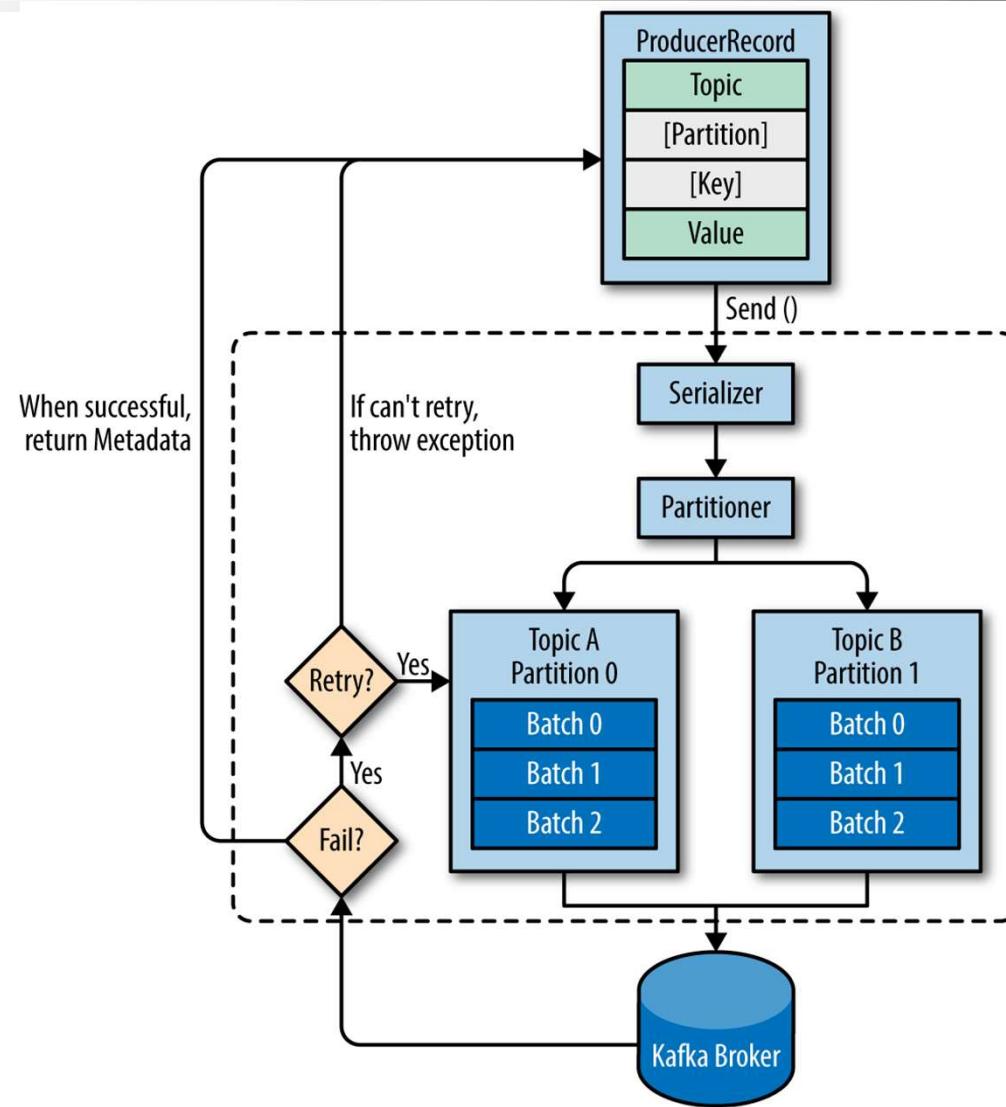


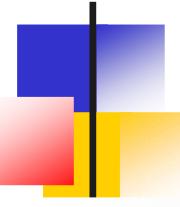
Étapes lors de l'envoi d'un message

L'envoi de message est constitué de plusieurs étapes :

- Création d'un objet **Message** encapsulant le contenu du message et optionnellement une clé, un timestamp 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 **DeliveryReport** 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

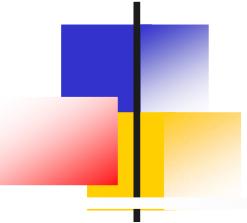
La première étape pour l'envoi consiste à construire un ***IProducer*** à partir d'un objet de configuration ***ProducerConfig*** en utilisant le builder ***ProducerBuilder***

1 propriété de configuration est obligatoire :

- ***BootstrapServers*** : Liste de brokers que le producteur contacte au départ pour découvrir le cluster

3 optionnelles sont généralement positionnées :

- ***key.serializer*** : La classe utilisée pour la sérialisation de la clé indiquée via le builder
- ***value.serializer*** : La classe utilisée pour la sérialisation du message indiquée via le builder
- ***client.id*** : Permet le suivi des messages

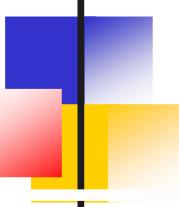


Exemple .NET

```
var config = new ProducerConfig
{
    // Adresse(s) de nœuds du cluster
    BootstrapServers = "localhost:9092" };

    // Crée une instance de Producteur
    // Pour envoyer des messages dont la valeur est de type
    // MyObject et une clé string

var producer = new ProducerBuilder<string, MyObject>(config)
    .SetValueSerializer(mySerializer)
    .Build()
```

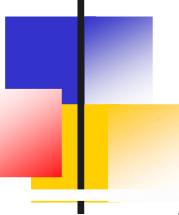


Message

Message représente l'enregistrement à envoyer à Kafka.

Il contient une valeur et éventuellement une clé, un timestamp et des entêtes

```
var message = new Message<string, string> {  
    Key = "key1",  
    Value = "value1",  
    Headers = new Headers { { "header1",  
        Encoding.UTF8.GetBytes("header-value1") } },  
    Timestamp = new Timestamp(DateTime.UtcNow)  
};
```

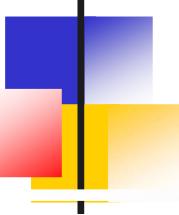


Méthodes d'envoi des messages

L'interface *IProducer* propose 2 méthodes :

- ***ProduceAsync*** : Retourne une *Task* : permet un envoi synchrone ou asynchrone
- ***Produce*** : Permet d'indiquer des gestionnaire (call-back) d'erreur ou de résultat

Il est alors possible d'implémenter du *FireAndForget*, du traitement synchrone ou asynchrone des résultats ou des erreurs

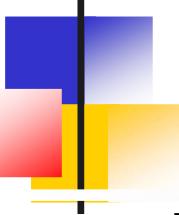


Fire And Forget

```
var config = new ProducerConfig
{
    BootstrapServers = "localhost:9092",
    // Indique au framework de ne pas gérer les méta-data
    // Améliore les performances dans le cas FireAndForget
    EnableDeliveryReports = false
};

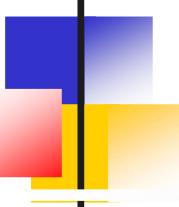
using var producer = new ProducerBuilder<string,
    string>(config).Build();

// Produce sans call-back, i.e Fire and Forget
producer.Produce("topic-name", new Message<string, string> { Key =
    "key", Value = "value" });
```



Envoi synchrone

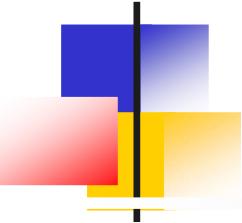
```
var config = new ProducerConfig { BootstrapServers = "localhost:9092"  
};  
  
// Création du producteur  
using (var producer = new ProducerBuilder<Null, string>(config).Build())  
{  
    try {  
        // Envoi synchrone du message  
        var result = producer.ProduceAsync("my-topic",  
            new Message<Null, string> { Value = "Hello Kafka!" })  
            .GetAwaiter().GetResult();  
        Console.WriteLine($"Message sent to {result.Partition} with  
{result.Offset}");  
    } catch (ProduceException<Null, string> e)  
    {  
        Console.WriteLine($"Error producing message: {e.Error.Reason}");  
    }  
}
```



Envoi asynchrone avec call-back

```
using (var producer = new ProducerBuilder<Null, string>(config).Build())
{
    // Envoi asynchrone avec callback
    producer.Produce("my-topic", new Message<Null, string> { Value = "Hello Kafka with
        Callback!" },
        (deliveryReport) =>
    {
        if (deliveryReport.Error != null)
        {
            Console.WriteLine($"Error: {deliveryReport.Error.Reason}");
        }
        else
        {
            Console.WriteLine($"Message delivered to
{deliveryReport.TopicPartitionOffset}");
        }
    });
}

// S'Assurer que le producteur a traité tous les messages avant de fermer
producer.Flush(TimeSpan.FromSeconds(10));
}
```

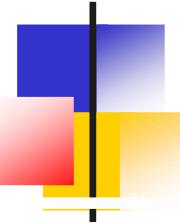


Envoi asynchrone d'un lot de messages

```
public async Task ProduceMessagesAsync(string topic, List<Message<string, string>>
    messages)
{
    var tasks = new List<Task<DeliveryResult<string, string>>();

    foreach (var message in messages)
    {
        // Ajoute chaque tâche ProduceAsync à la liste
        tasks.Add(_producer.ProduceAsync(topic, message));
    }

    try
    {
        // Attend que toutes les tâches soient terminées
        await Task.WhenAll(tasks);
        Console.WriteLine($"Successfully produced {tasks.Count} messages to topic
{topic}");
    }
    catch (ProduceException<string, string> e)
    {
        Console.WriteLine($"Delivery failed: {e.Error.Reason}");
    }
}
```

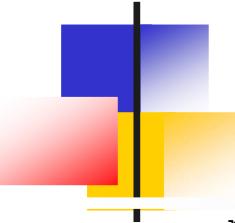


Sérialiseurs

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

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

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

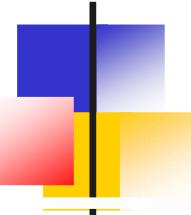


Exemple sérialiseur custom s'appuyant sur Json

```
public class CustomSerializer<T> : ISerializer<T>
{
    public byte[] Serialize(T data, SerializationContext context)
    {
        if (data == null)
            return Array.Empty<byte>();

        // Sérialiser l'objet en JSON
        return JsonSerializer.SerializeToUtf8Bytes(data);
    }
}

---
var config = new ProducerConfig
{
    BootstrapServers = "localhost:9092"
};
var producer = new ProducerBuilder<long,
    Position>(_config).SetValueSerializer(new
    CustomSerializer<CustomType>()).Build()
```



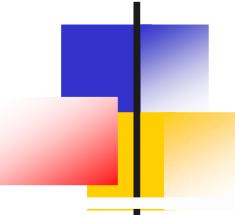
Configuration des producteurs

fiabilité

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

Fiabilité :

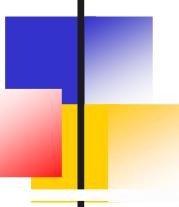
- ***acks*** : contrôle le nombre de réplicas qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie
 - ***retries*** : Si l'erreur renvoyée est de type Retriable, le nombre de tentative de renvoi. Si > 0 possibilité d'envoi en doublon
 - ***max.in.flight.requests.per.connection*** : 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 métadonnées (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



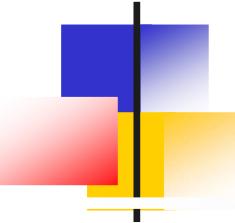
Garantie sur l'ordre

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 ...

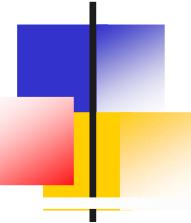
En cas de failure

- Si *retries* > 0 et *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
retries > 0 et *max.in.flights.requests.per.session* = 1
(au détriment du débit global)



APIs

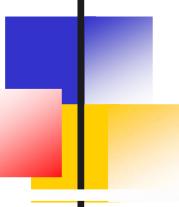
Producer API
Consumer API
Schema Registry
Connect API
Admin API



Introduction

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

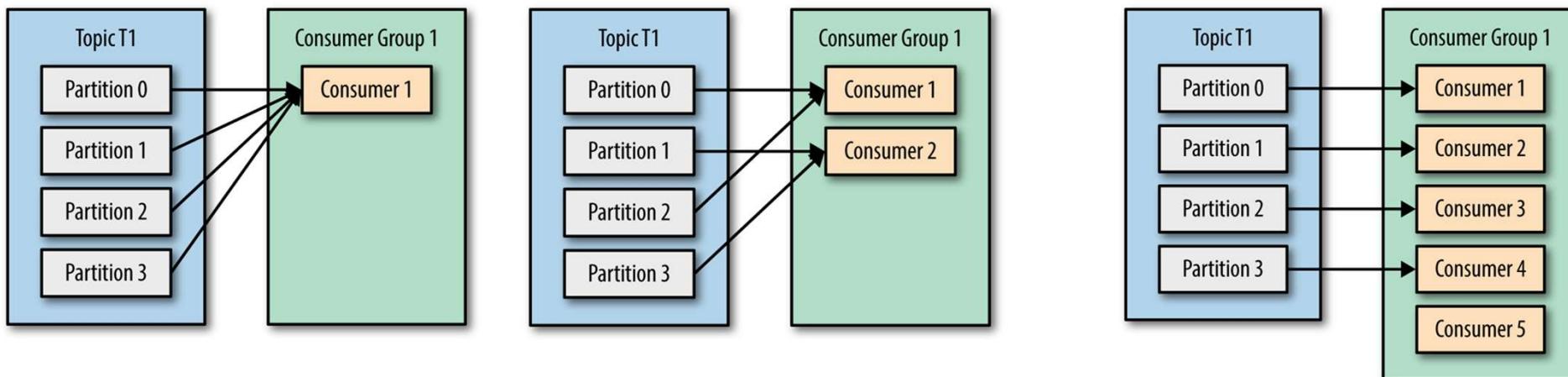
Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relations avec les partitions

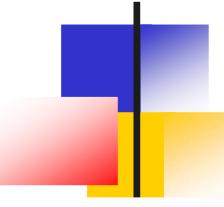


Groupes de consommateurs

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.





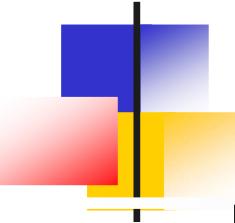
Rééquilibrage dynamique des consommateurs

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

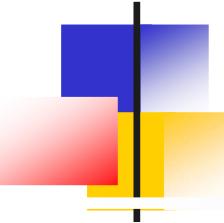


Membership et détection de pannes

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*

La construction d'un ***IConsumer*** est similaire à celle d'une ***IProducer***

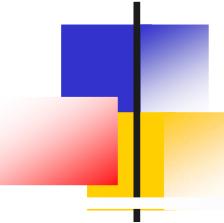
Instancier une classe ***ConsumerConfig*** et la fournir au ***ConsumerBuilder*** :

Plusieurs propriétés doivent être spécifiées :

- *bootstrap.servers*
- *group.id* qui spécifie le groupe de consommateur

```
var config = new ConsumerConfig
{
    BootstrapServers = "host1:9092,host2:9092",
    GroupId = "foo",
};

using (var consumer = new ConsumerBuilder<Ignore, string>(config).Build())
{
    ...
}
```



Abonnement à un *topic*

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

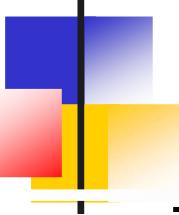
La méthode ***Subscribe()*** prend une liste de *topics* comme paramètre.

Ex :

```
consumer.Subscribe("myTopic");
```

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

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



Boucle de Polling

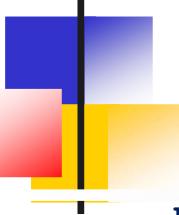
Typiquement, les consommateurs appellent continuellement la méthode ***Consume***.

La méthode *Consume* récupère les enregistrements un par un sous forme de ***ConsumerResult*** mais les messages sont en fait récupérés par lots via des threads en background. (*polling*)

Plusieurs signatures pour *Consume* :

- *Consume(int timeout)* : Bloque jusqu'à ce qu'il récupère un message ou que le timeout expire
- *Consume (Timespan timeout)* : idem
- *Consume(CancellationToken)* : Bloque jusqu'à ce qu'il récupère un message ou qu'un ordre d'annulation servient

ConsumerResult encapsule le message, la partition, l'offset, le timestamp



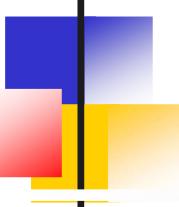
Exemple

```
using (var consumer = new ConsumerBuilder<Ignore, string>(config).Build())
{
    consumer.Subscribe(topics);
    // Créez une source de token d'annulation
    var cancellationTokenSource = new CancellationTokenSource();

    // Utilisez une tâche pour écouter les interruptions (par exemple, Ctrl+C)
    Console.CancelKeyPress += (_, e) => {
        e.Cancel = true; // Empêcher la fermeture immédiate
        cancellationTokenSource.Cancel(); // Demander l'annulation
    };
    // Boucle de consommation
    while (! cancellationTokenSource.Token.IsCancellationRequested)
    {
        var consumeResult = consumer.Consume(cancellationTokenSource.Token);

        // handle consumed message.
        ...
    }

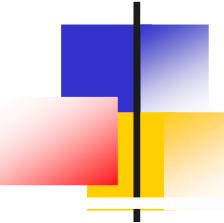
    consumer.Close();
}
```



Configuration des consommateurs

Les propriétés les plus importantes :

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

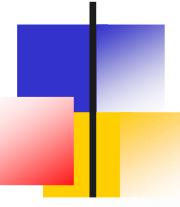


Configuration des consommateurs

(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***

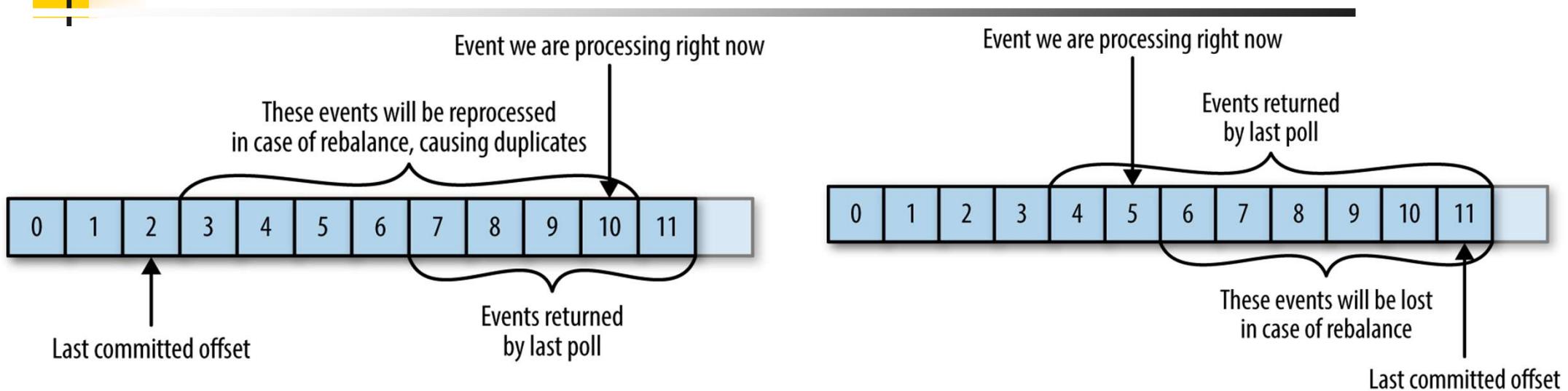
Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka : **__consumer_offsets**

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

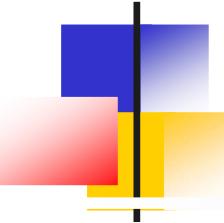
Lors d'une réaffectation de partitions, 2 risques doivent être pris en compte :

- Traiter 2 fois le même message
- Louper des messages

Risques lors d'une réaffectation

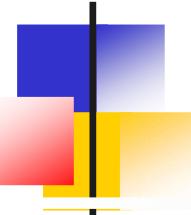


Kafka propose plusieurs façons de gérer les commits



Commit automatique

-
- Si ***EnableAutoCommit=true***,
le consommateur valide toutes les
AutoCommitIntervalMs (par défaut 5000), les
plus grands offset récupérés par les threads de
background ...
 - ... MAIS peut être pas encore consommés par
l'application et la méthode *Consume()*
- => Cette approche (simple) ne protège pas contre les pertes de messages ni les traitements en doublon en cas de failure !

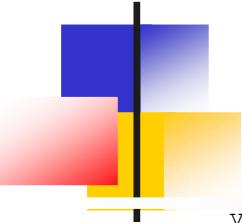


Commit contrôlé

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

Si ***EnableAutoCommit =false***, l'application doit explicitement committer les offsets

- Via un commit synchrone (bloquant) avec
Commit()
- => Ne pas committer à chaque message reçu



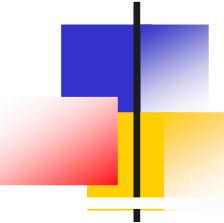
Exemple At Most Once

```
var config = new ConsumerConfig
{
    ...
    // Disable auto-committing of offsets.
    EnableAutoCommit = false
}

...
while (!cancelled)
{
    var consumeResult = consumer.Consume(cancellationToken);

    // process message here.

    if (consumeResult.Offset % commitPeriod == 0)
    {
        try
        {
            consumer.Commit(consumeResult);
        }
        catch (KafkaException e)
        {
            Console.WriteLine($"Commit error: {e.Error.Reason}");
        }
    }
}
```

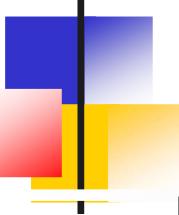


Exemple At Least Once

```
var config = new ConsumerConfig
{
    ...
    // Disable auto-committing of offsets.
    EnableAutoCommit = false
}

...
while (!cancelled)
{
    var consumeResult = consumer.Consume(cancellationToken);

    if (consumeResult.Offset % commitPeriod == 0)
    {
        try
        {
            consumer.Commit(consumeResult);
        }
        catch (KafkaException e)
        {
            Console.WriteLine($"Commit error: {e.Error.Reason}");
        }
    }
    // process message here.
}
```



StoreOffset

La fonctionnalité d'Autocommit est cependant flexible

Il est possible d'empêcher l'autocommit automatique du dernier offset récupéré en background avec :

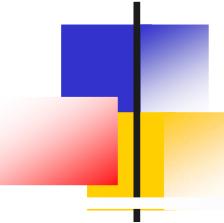
EnableAutoOffsetStore =false.

Il faut alors utiliser la méthode ***StoreOffset*** pour indiquer les offset que l'on désire valider.

- Cette méthode peut être appelée plusieurs fois mais la mise à jour de l'offset ne se fera qu'en fonction de l'intervalle de commit et du plus grand offset spécifié.

Cette approche est préférée à l'approche de validation synchrone précédente.

=> On obtient du *At Least Once* sans appels bloquants



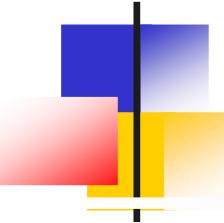
Exemple At Least Once

```
var config = new ConsumerConfig
{
    ...
    EnableAutoCommit = true // (the default)
EnableAutoOffsetStore = false
}

...
while (!cancelled)
{
    var consumeResult = consumer.Consume(cancellationToken);

    // process message here.

    consumer.StoreOffset(consumeResult);
}
```



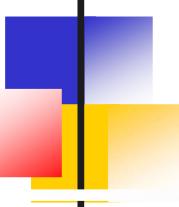
Committer un offset spécifique

L'API permet également de fournir en argument une

IEnumerable<TopicPartitionOffset>

contenant les offsets que l'on veut valider pour chaque partition

=> Cela permet de committer des offsets sans rapport avec la consommation

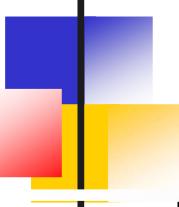


Stocker les offsets hors de Kafka

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 assez 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, ...)

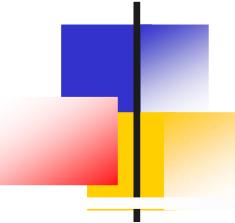
Lors de la construction du Consumer, il est possible de fournir un ***PartitionRevokedHandler*** et un ***PartitionAssignedHandler***

```
using (var consumer = new ConsumerBuilder<Ignore, string>(config

    .SetPartitionsRevokedHandler((c, partitions) =>

    {

        log.Info("Lost partitions in rebalance.           Committing current offsets:" +
currentOffsets);
        consumer.commit(currentOffsets);
    }).Build()) { ... }
```

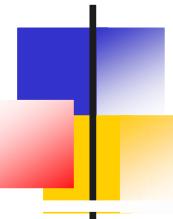


Consommation de messages avec des offsets spécifiques

L'API permet de se placer à un offset spécifique :

- ***Seek(TopicPartitionOffset)*** :

=> Lorsque l'on gère soit même les rebalance, cela permet de se positionner à l'offset stocké hors Kafka

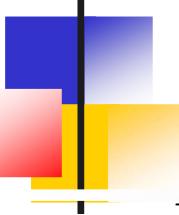


Affectation statique des partitions

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`

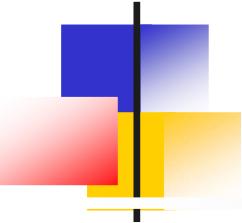


Exemple (Java)

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

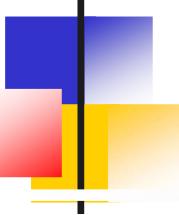
if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new
TopicPartition(partition.topic(),partition.partition()));

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 API

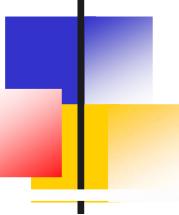


Introduction

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 le code correspondant au schéma.
Ex : Package ***Apache.Avro.Tools***, ***Chr.Avro*** en .NET

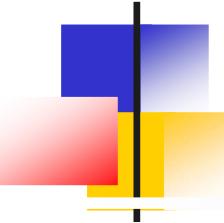
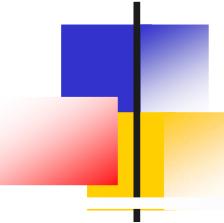


Schéma Avro

```
{  
  "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" } ] } ] } }
```



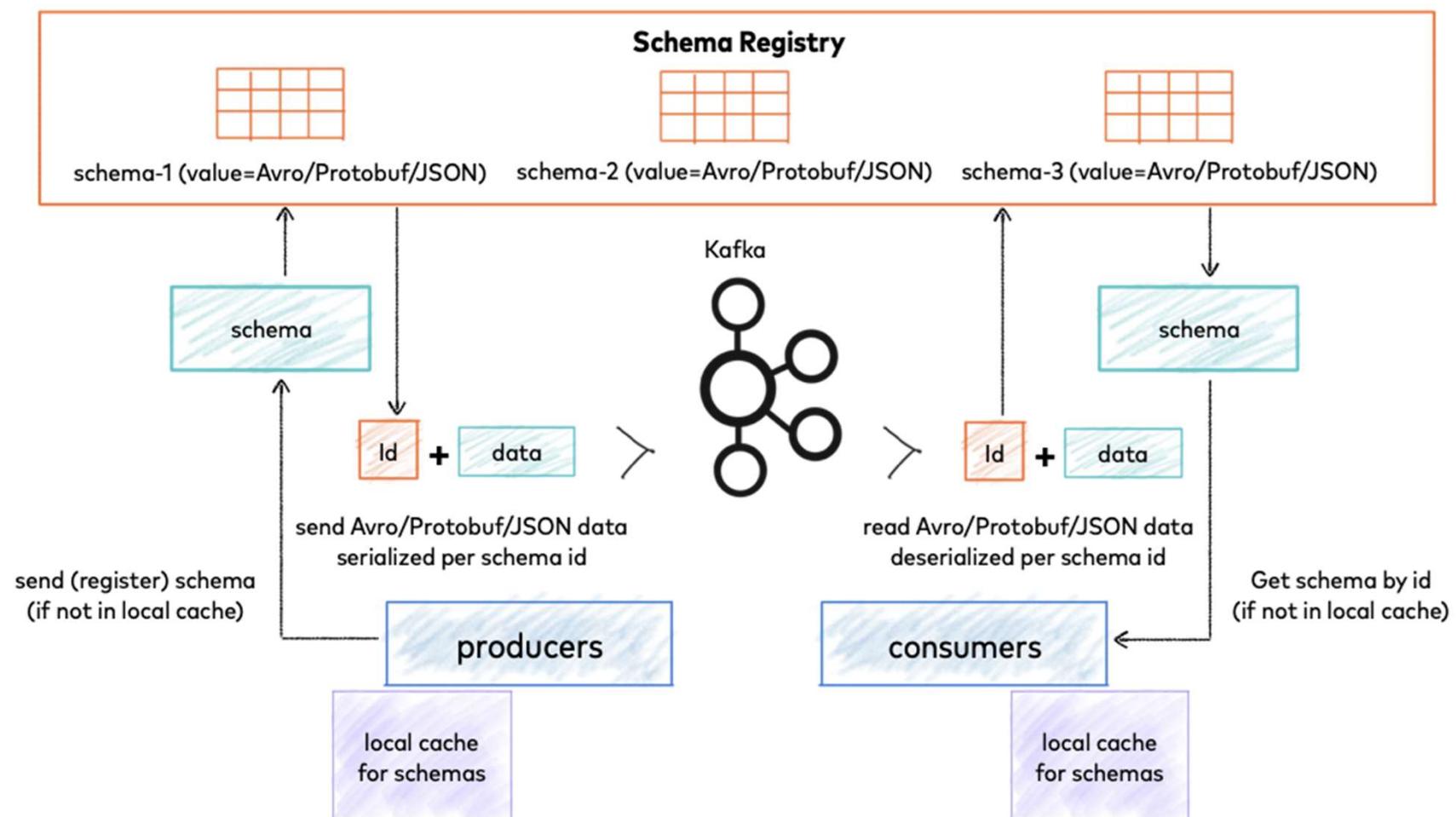
Utilisation du schéma

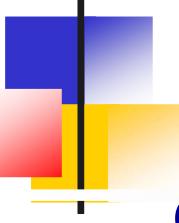
Confluent Schema Registry offre une API Rest

- Permettant de 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

Schema Registry





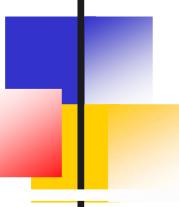
Packages Confluent

Confluent.SchemaRegistry.Serdes.Avro : Fournit un sérialiseur et désérialiseur *Avro* et intégrer Confluent Schema Registry.

Confluent.SchemaRegistry.Serdes.Protobuf - Fournit un sérialiseur et désérialiseur *Protobuf* et intégrer Confluent Schema Registry.

Confluent.SchemaRegistry.Serdes.Json - Fournit un sérialiseur et désérialiseur pour travailler avec Json et intégrer Confluent Schema Registry.

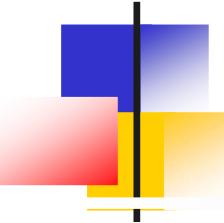
Confluent.SchemaRegistry – Le client du Schema Registry
(dépendances des autres packages)



Producteur (1)

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

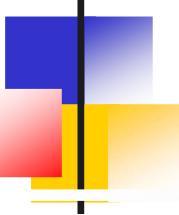
```
var schemaRegistry = new CachedSchemaRegistryClient(new SchemaRegistryConfig {  
    Url = schemaRegistryUrl });  
  
string avroSchemaString = File.ReadAllText("myschema.avsc");  
  
schemaRegistry.RegisterSchemaAsync($"{topic}-value", avroSchemaString).Wait();
```



Producteur (2)

A la création du Producer, il faut positionner le sérialiseur *AvroSerializer*:

```
var schemaRegistry = new CachedSchemaRegistryClient(new  
    SchemaRegistryConfig { Url = schemaRegistryUrl } );  
  
new ProducerBuilder<long, Coursier>(_config)  
    .SetValueSerializer(new AvroSerializer<Coursier>(schemaRegistry))  
    .Build();
```



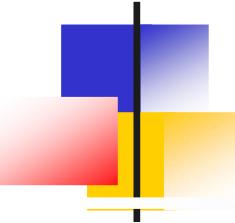
Consommateur

Le consommateur doit utiliser AvroDeserializer dont le constructeur prend le client du SchemaRegistry

```
SetValueDeserializer(new  
AvroDeserializer<GenericRecord>(schemaRegistry).AsSyncOverAsyn  
c())
```

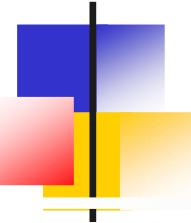
Il peut récupérer les messages sous la forme de **GenericRecord** plutôt que des classes spécialisés.

```
var consumeResult = consumer.Consume(cancellationToken);  
GenericRecord record = consumeResult.Message.Value;  
  
Console.WriteLine($"Position : {record["position"]}");
```



APIs

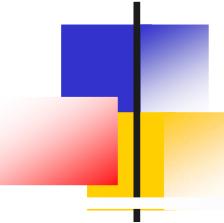
Producer API
Consumer API
Schema Registry
Connect API
Admin API



Introduction

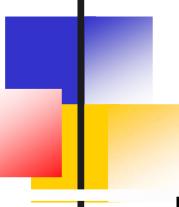
Kafka Connect permet d'intégrer Apache Kafka avec d'autres systèmes en utilisant des connecteurs.

- => 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
- Intégration vers les systèmes de streaming ou batch



Mode standalone

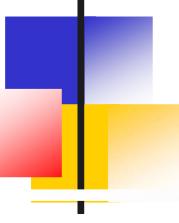
Pour démarrer *Kafka Connect* en mode standalone

```
> 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

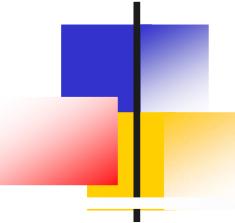
Le mode distribué gère le scaling dynamique et offre la tolérance aux pannes

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



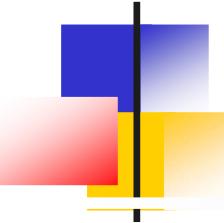
Configurations supplémentaires en mode distribué

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

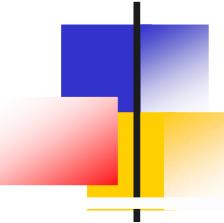
Configuration :

- En mode standalone, via un fichier *.properties*
- 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

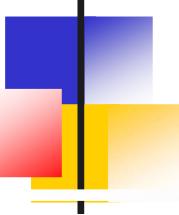


Connecteurs

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, ...)



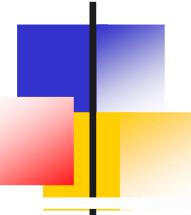
Exemple Connecteur JDBC

```
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

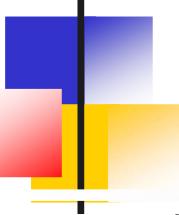
topic.prefix=mysql-
```



Transformations

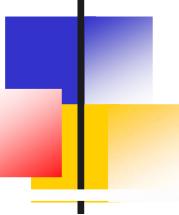
Une chaîne de transformation, s'appuyant sur des **transformers** prédéfinis, est spécifiée dans la configuration d'un connecteur.

- **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 définit un topic de destination
topic=connect-test
# 2 transformes 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
```



Transformers disponibles

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

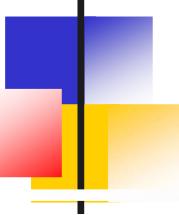
ValueToKey : Échange clé/valeur

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

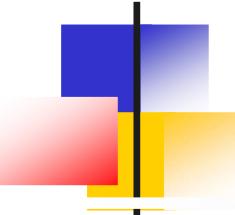
L'adresse d'écoute de l'API REST peut être configuré via la propriété listeners

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

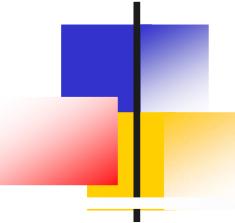
GET /connectors/{name}/tasks/{taskid}/status : Statut d'une tâche

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

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

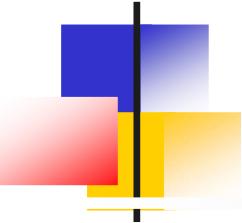
POST /connectors/{name}/restart : Redémarrage après un plantage

DELETE /connectors/{name} : Supprimer un connecteur



APIs

Producer API
Consumer API
Schema Registry
Connect API
Admin API



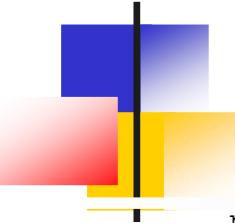
Introduction

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

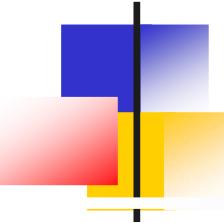
2. Équivalent .NET : Streamiz



Exemple Admin (Java) : Lister les configurations

```
public class ListingConfigs {

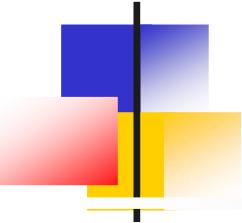
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        Properties config = new Properties();
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        AdminClient admin = AdminClient.create(config);
        for (Node node : 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 (Java)

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);  
    }  
}
```



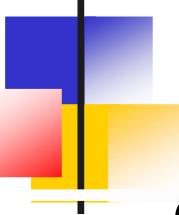
Garanties Kafka

Mécanismes de réPLICATION

At Most One, At Least One

Exactly Once

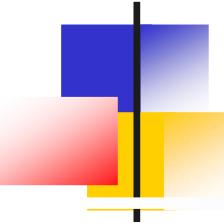
Débit, latence, durabilité



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 $max_in_flight_request > 1$ et $retries > 0$
- **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 certain de défaillance des brokers ou consommateurs .



Rôles des brokers

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

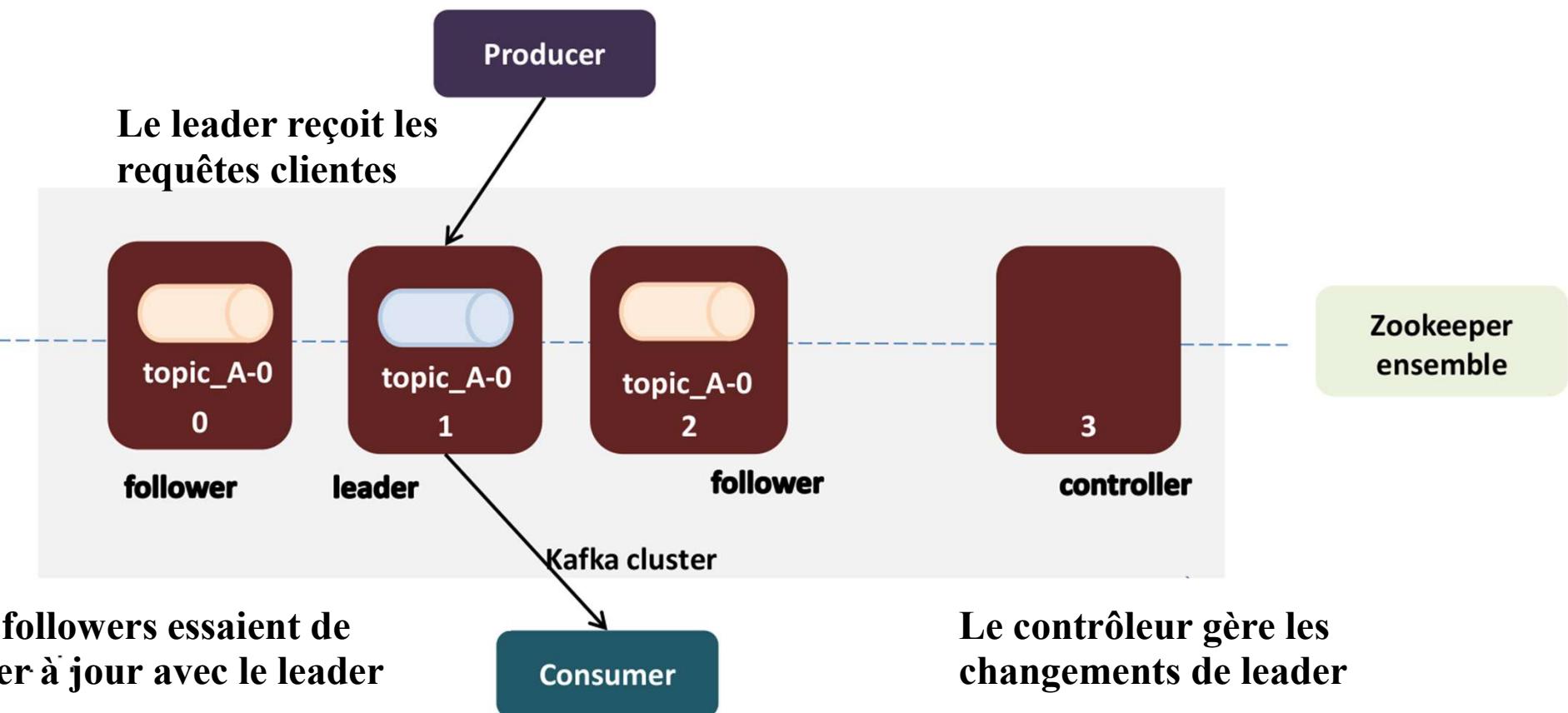
Pour chaque partition répliquée :

- **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 :

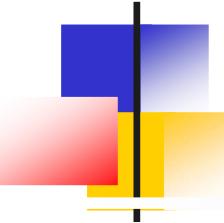
- Un **contrôleur** est responsable des élections de leader

Rôle des brokers gérant un topic



Les followers essaient de rester à jour avec le leader

Le contrôleur gère les changements de leader



Synchronisation des répliques

Contrôlé par la propriété :

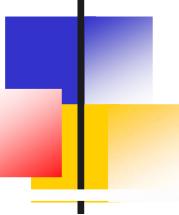
replica.lag.time.max.ms (défaut 30 000)

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

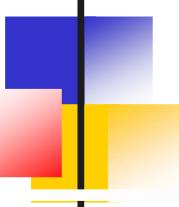
- Il est supprimé de la liste des **ISR (In Sync Replica)**
- Il peut ensuite rattraper son retard et être réintégré aux ISRs



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)



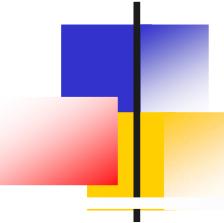
min.insync.replica

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*

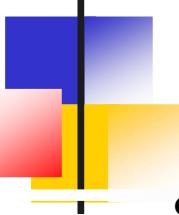
Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



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.

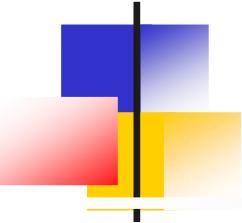
Si le **nombre de répliques disponible < *min.insync.replicas***

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

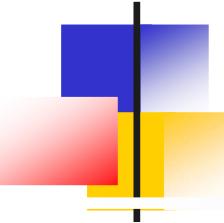
n répliques
=> tolère $n-1$ failures pour que la partition soit disponible

n répliques, $\text{min.insync.replicas} = m$
=> Tolère $n-m$ failures pour accepter les envois



Garanties Kafka

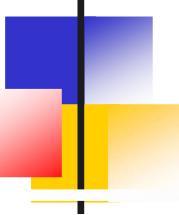
Mécanismes de réPLICATION
At Most Once, At Least Once
Exactly Once
Débit, latence, durabilité



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 ***MessageSendMaxRetries*** qui permet de tolérer les défaillances temporaires (un broker qui redémarre par exemple)



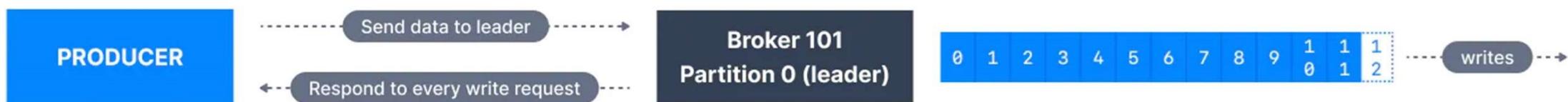
Acks=0

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

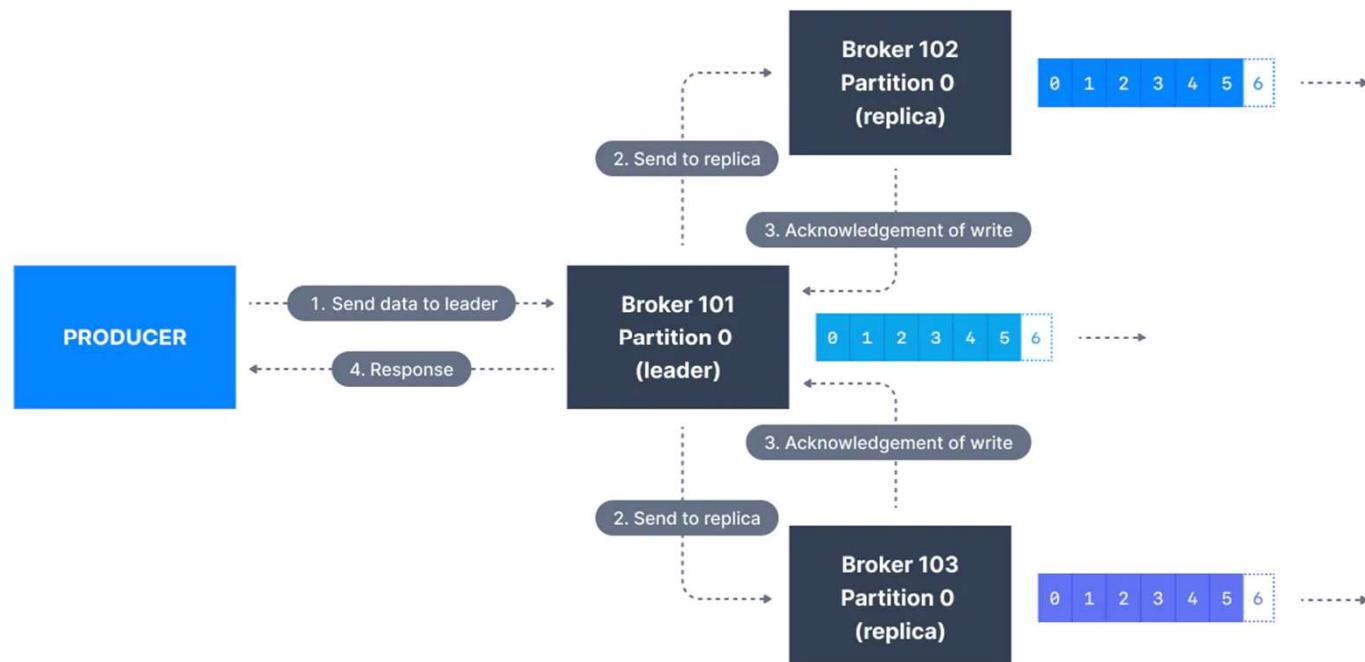
Acks=1 : Le producteur considère le message écrit lorsque le leader a acquitté l'écriture
=> Si le leader s'arrête et que les répliques n'ont pas eu le temps d'écrire le message, perte de données



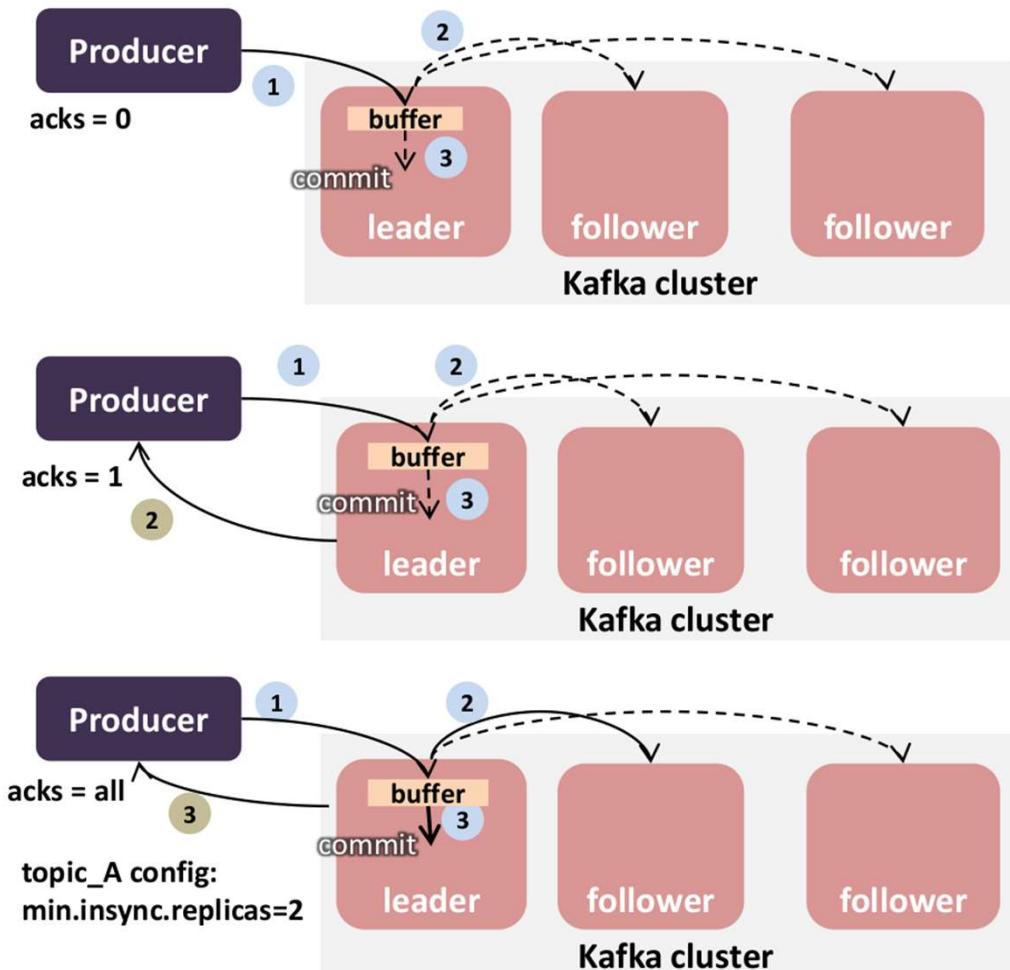
Acks=all

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



Latency

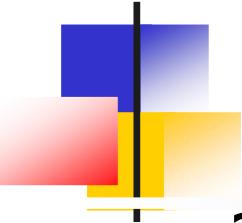


Data loss risk



Data loss risk





Gestion des erreurs

2 types d'erreur pour le producteur:

- les erreurs que les producteurs gèrent automatiquement .

Ce sont les erreurs ré-essayable (ex :

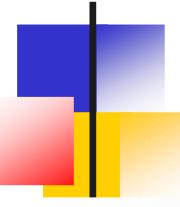
LEADER_NOT_AVAILABLE, NOT_ENOUGH_REPLICA)

Nombre d'essai configurable via

MessageSendMaxRetries.

=> Attention, peut générer des doublons

- les erreurs qui doivent être traitées par le code.
(ex : *INVALID_CONFIG,*
SERIALIZATION_EXCEPTION)

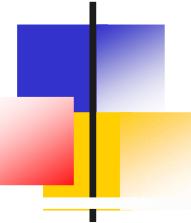


Côté consommateur

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

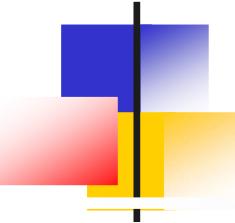


Gestion des commits

Les commits peuvent être automatiquement géré par Kafka

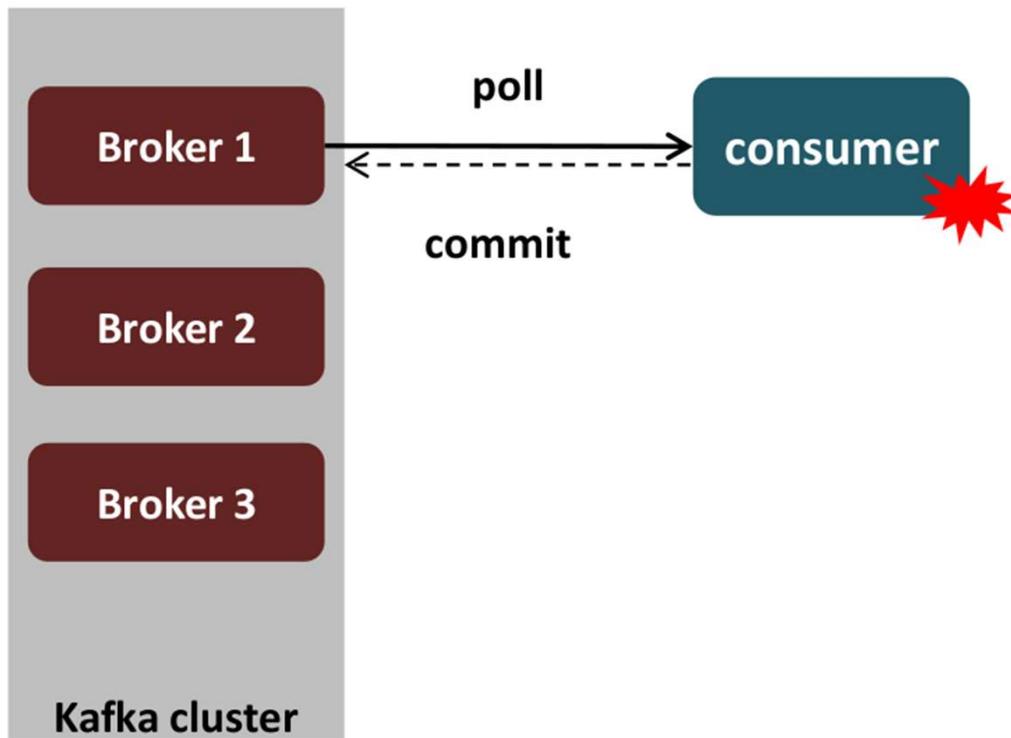
Si ***EnableAutoCommit=true***,

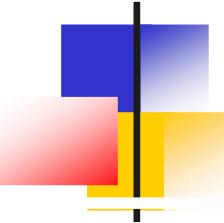
Lorsque la thread de background récupère un lot de message (appel à `poll()`) si ***AutoCommitIntervalMs*** s'est écoulée, il valide les plus grands offset reçus par le `poll()` précédent



Consommation: At Most Once

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





Configuration *At Most Once*

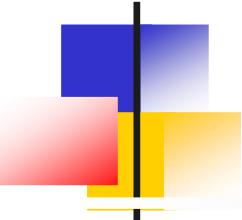
Configuration par défaut

OU

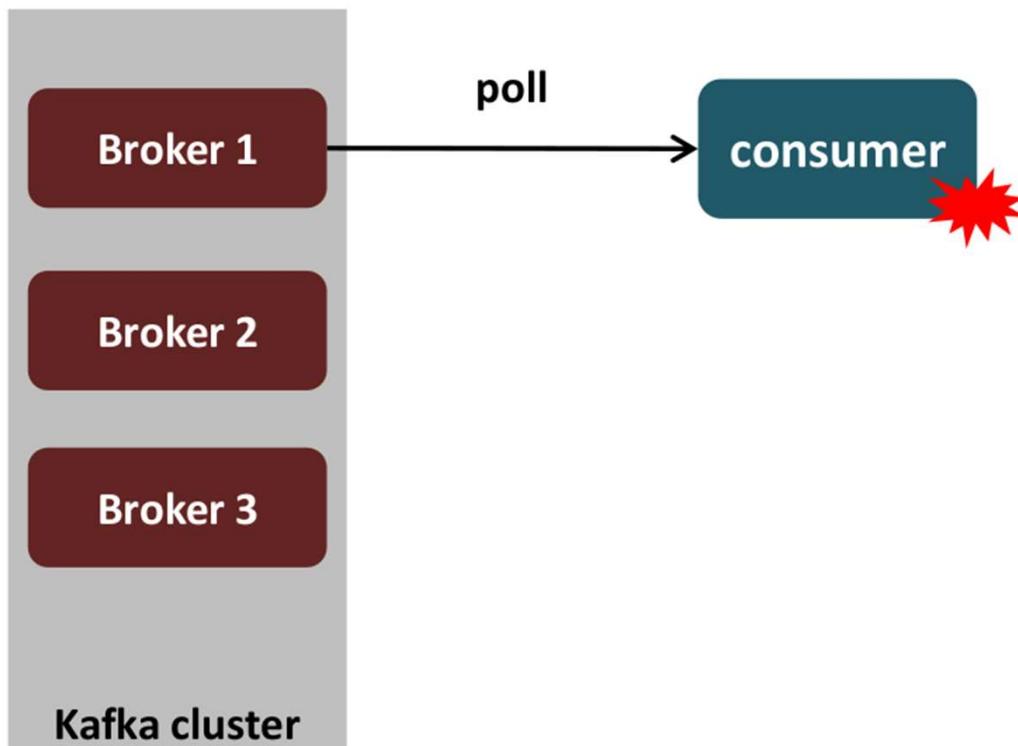
Commit manuel avant le traitement des messages ou traitement de messages asynchrone

Exemple : Commit manuel et traitement asynchrone

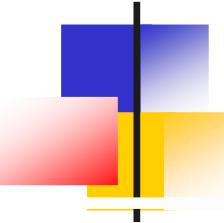
```
while (!cancelled) {
    var consumeResult = consumer.Consume(cancellationToken);
    // Traitement asynchrone
    asyncProcess(consumeResult);
    if (consumeResult.Offset % commitPeriod == 0)  {
        try  {
            consumer.Commit(consumeResult);
        } catch (KafkaException e) { Console.WriteLine($"Commit error: {e.Error.Reason}"); }
    }
}
```



Réception : *At Least Once*



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



Configuration *At Least Once*

AutoCommit avec *StoreOffset* (méthode recommandée)

Ou

Commit manuel après traitement via *consumer.Commit();*

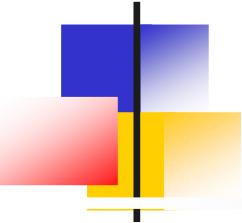
Exemple : AutoCommit et StoreOffset

```
while (!cancelled)
{
    var consumeResult = consumer.Consume(cancellationToken);

    // Traitement synchrone du message.
    syncProcess(consumeResult) ;

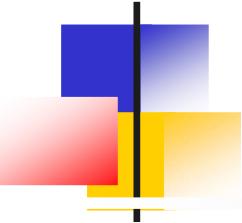
    consumer.StoreOffset(consumeResult);

}
```



Garanties Kafka

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

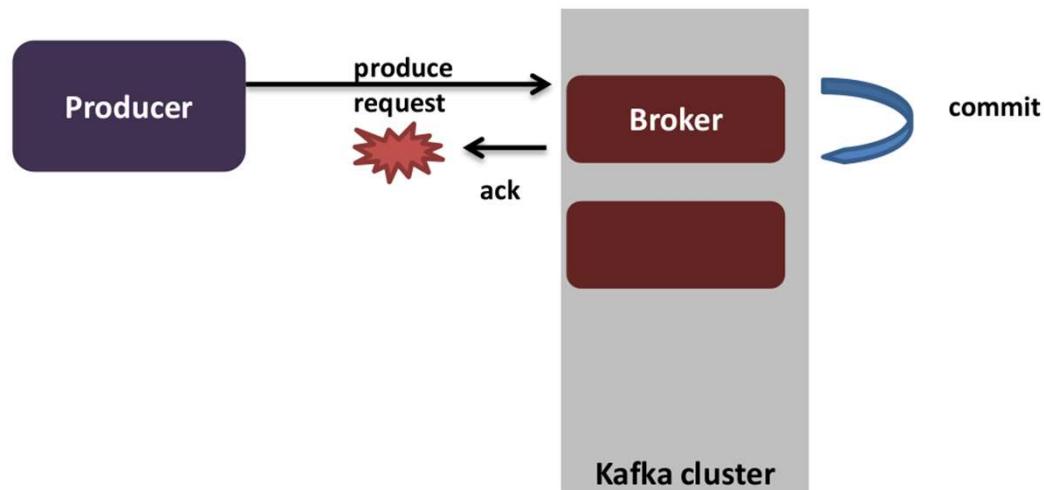


Introduction

La sémantique *Exactly Once* est basée sur *At least Once* et empêche la production ou la consommation de doublon

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

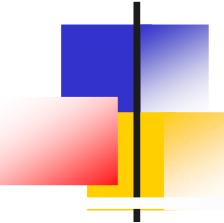
Producteur idempotent



Le producteur ajoute une nombre séquentiel et un ID de producteur
Dans le message.

Le broker détecte le doublon

=> Il envoie un *ack* mais sans stocker le message



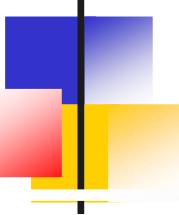
Configuration producteur

EnableIdempotence

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

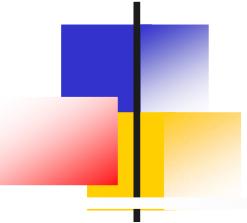


Consommateur

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

- Gérer manuellement les offsets committés dans un support de persistance transactionnel, 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.

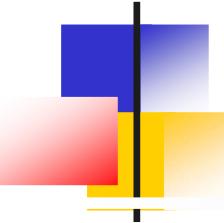
1. C'est le cas de KafkaStream



Exemple Gestion des offsets (1)

EnableAutoCommit=false

```
while (!cancelled)
{
    var consumeResult = consumer.Consume(cancellationToken);
    // Démarrage transaction
    // Traitement du message
    // Sauvegarder l'offset traité .
    offsetManager.saveOffsetInExternalStore(record.topic(),
        record.partition(), record.offset());
    // Commit transaction
}
}
```



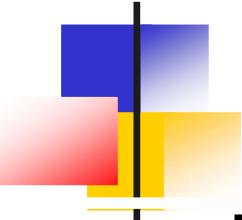
Exemple Gestion des offsets (2)

```
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));
        }
    }

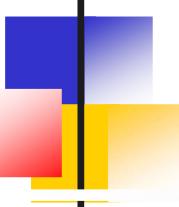
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            consumer.seek(partition,
                offsetManager.readOffsetFromExternalStore(partition.topic(),
                    partition.partition()));
        }
   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



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

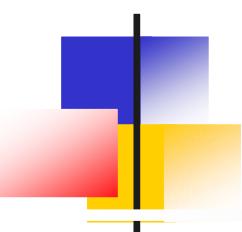
- Le mode transactionnel inclut le mode idempotent
- Les topics impliqués doivent être configurés avec *replication.factor >= 3* et *min.insync.replicas = 2*
- Les consommateurs doivent avoir la propriété *isolation.level* à *read committed*
- L'API *send()* devient bloquante



Transaction

Côté producteur, les transactions permettent la production atomique d'un lot de messages, i.e. Les producteurs produisent l'ensemble des messages ou aucun

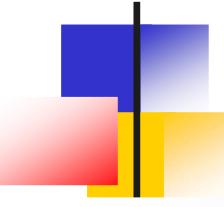
```
// initTransaction démarre une transaction avec l'id.  
// Si une transaction existe avec le même id, les messages sont  
// roll-backés  
producer.InitTransactions(DefaultTimeout);  
try {  
    producer.BeginTransaction();  
    for (int i = 0; i < 100; ++i) {  
        var message = new Message...  
        await producer.ProduceAsync(message);  
    }  
    producer.commitTransaction();  
} catch (Exception e) { producer.abortTransaction(); }
```



Configuration du consommateur

Afin de lire les messages transactionnels validés :

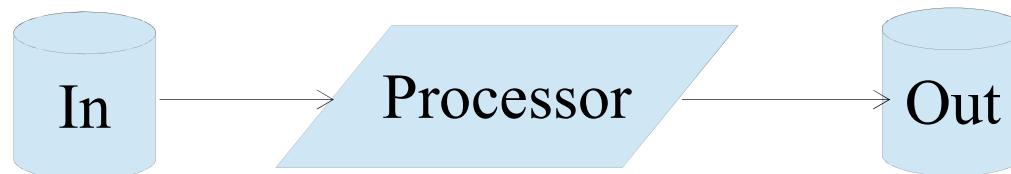
- ***IsolationLevel=read_committed***
(Défaut: *read_committed*)
 - *read_committed*: Messages (transactionnels ou non) validés
 - *read_uncommitted*: Tous les messages (même les messages transactionnels non validés)

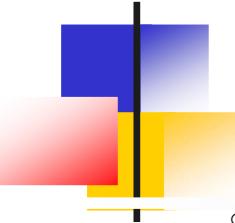


Exactly Once pour le transfert de messages entre topics

Lorsque un consommateur produit vers un autre topic, on peut utiliser les transactions afin d'écrire l'offset dans la même transaction que les messages 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 / Producteur



Streamprocessor

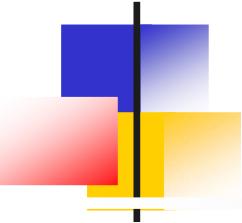
```
consumer.Subscribe(Topic_InputLines);

producer.InitTransactions(DefaultTimeout);
producer.BeginTransaction();
while (!cancellationToken) {
    // Ne pas bloquer indéfiniment pour ne pas provoquer de timeout de la transaction
    var cr = consumer.Consume(TimeSpan.FromSeconds(1));
    // Process incoming message and produce outgoing messages
    Map<String, Integer> wordCountMap =...
    wordCountMap.forEach((key, value) ->
        producer.Produce(OUTPUT_TOPIC, new Message<string, Null> { Key = w }));
}

if (DateTime.Now > lastTxnCommit + txnCommitPeriod) {
    producer.SendOffsetsToTransaction(
        consumer.Assignment.Select(a => new TopicPartitionOffset(a, consumer.Position(a))),
        consumer.ConsumerGroupMetadata,
        DefaultTimeout);
    producer.CommitTransaction();
    producer.BeginTransaction();
    lastTxnCommit = DateTime.Now;
}
}
```

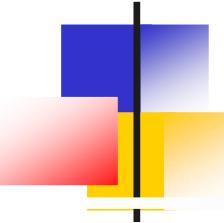
Voir Exemple :

<https://github.com/confluentinc/confluent-kafka-dotnet/tree/master/examples/ExactlyOnce>



Garanties Kafka

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



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)

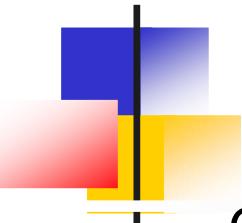
Puis

- *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

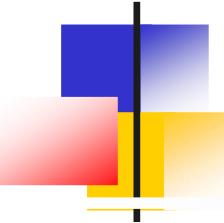
- *num.replica.fetchers* : (défaut 1)

Côté producteur

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

Côté consommateur

- *fetch.min.bytes: 1*



Configuration pour la durabilité

Cluster

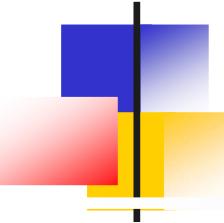
- *replication.factor*: 3
- *min.insync.replicas*: 2 (défaut 1)
- *unclean.leader.election.enable* : false (défaut false)
- *broker.rack*: rack du broker (défaut null)

Producteur

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

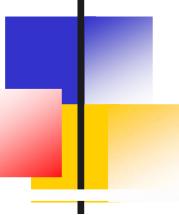
Consommateur

- *isolation.level: read_committed*



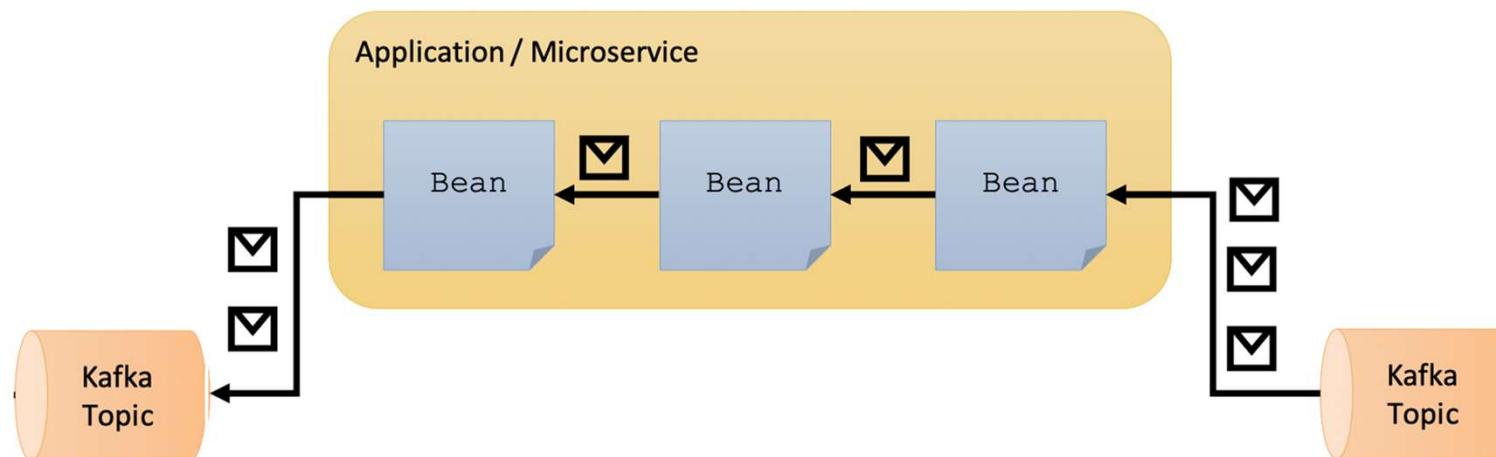
KafkaStream

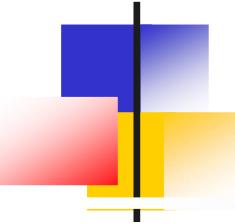
Concepts
Streamiz
ksqldb



Kafka Streams

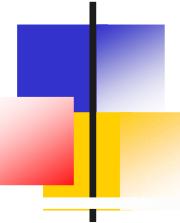
Kafka Streams API est une bibliothèque cliente pour la création d'applications et de micro-services, où les données d'entrée et de sortie sont stockées dans des topics Kafka





Apports de *KafkaStream*

- Abstractions *KStream* et *KTable* permettant la transformation de flux d'évènements infinis, des jointures et des agrégations
- Peut garantir que chaque évènement soit traité une et une seule fois, même en cas de défaillance. (Par défaut 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.

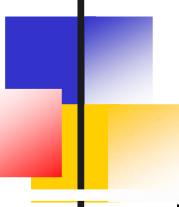


Définitions

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

D'autres attributs caractérisent ce flux d'événements :

- Les événements sont ordonnés
- Les événements sont immuables
- On peut rejouer un flux d'événements



Application KafkaStream

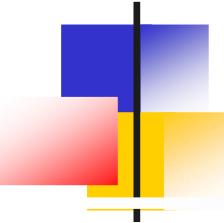
Une application KafkaStream définit sa logique de traitement à travers une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

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

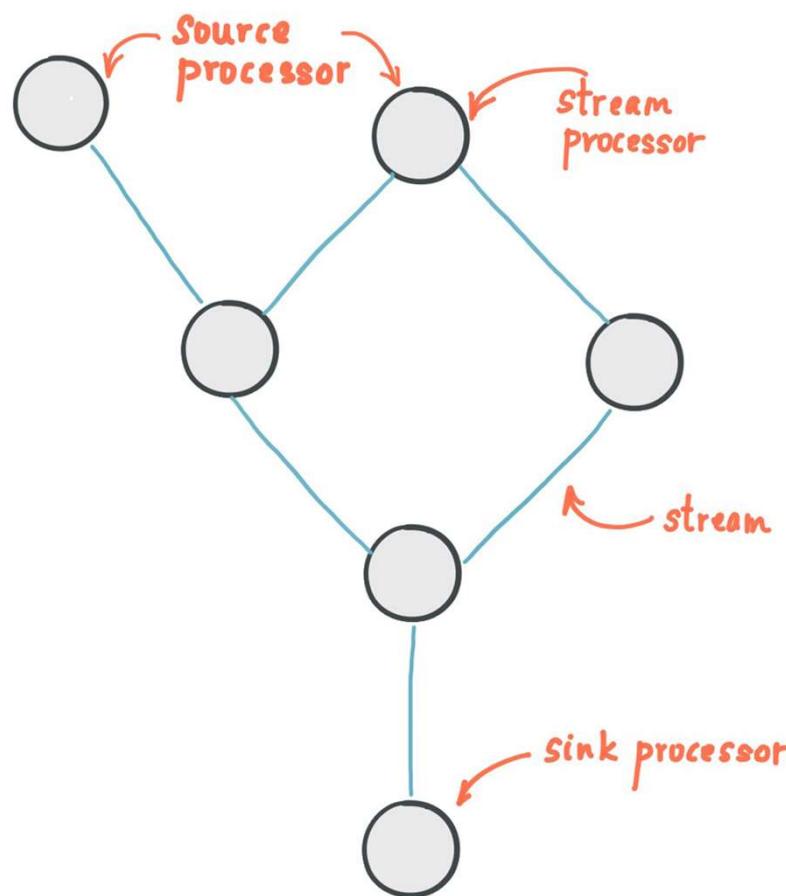
Certains processeurs :

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

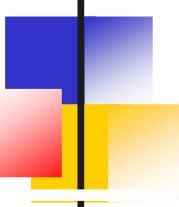
La topologie peut être spécifiée programmatiquement ou par un DSL offrant les opérateurs classiques (filter, map, join, ...)



Topologie processeurs



PROCESSOR TOPOLOGY



KStream et KTable

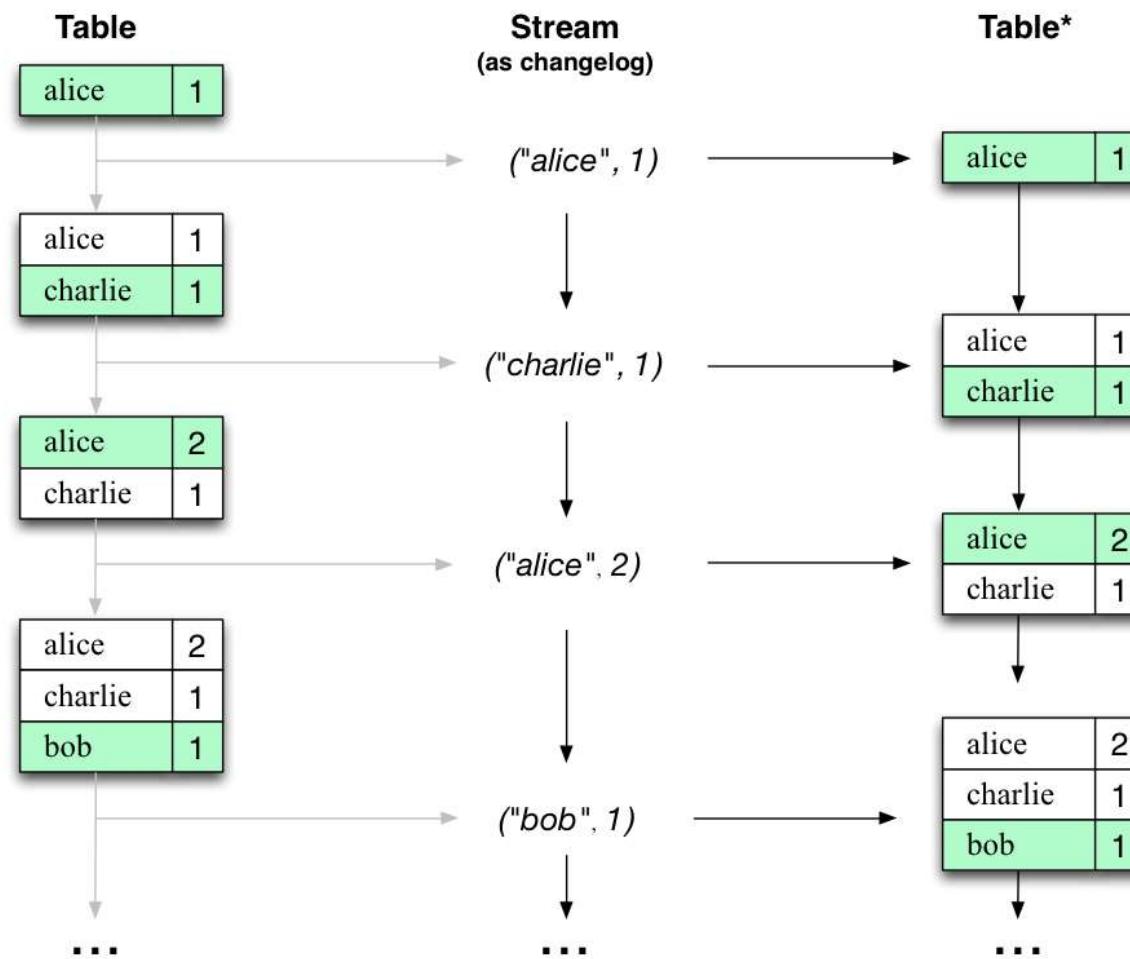
KafkaStream fournit également l'abstraction ***KTable*** qui représente un ensemble de fait qui évoluent.

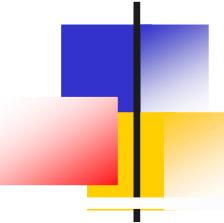
Cela peut être vu comme une table d'une base de données ou il n'existe qu'une valeur pour une clé donné.

A table

key1	value1
key2	value2
key3	value3

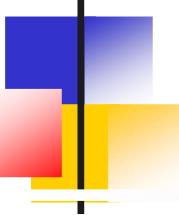
Dualité KStream / KTable





Corollaires

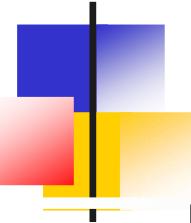
- Une *KTable* peut être transformé en *KStream* contenant les événements d'update
- Une *KTable* peut être construit à partir d'un *KStream*.
Seule la dernière valeur ou une agrégation d'une clé donnée est conservée
- Un *KStream* peut effectuer des jointures sur une *KTable* et produit alors un *KStream* enrichi



KGlobalTable

KGlobalTable représente une table distribuée globale.

Contrairement à une KTable, qui est partitionnée et locale à chaque instance d'une application Kafka Streams, une KGlobalTable contient une copie complète des données sur chaque instance de l'application.



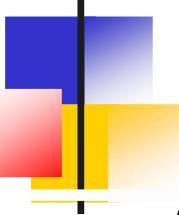
Agrégation

Une opération **d'agrégation** prend un flux d'entrée ou une table et génère une nouvelle table en combinant plusieurs enregistrements d'entrée en un seul enregistrement de sortie.

- Exemples : Somme, moyenne, ...

Une opération **de jointure** prend plusieurs flux d'entrée et fournit un flux de sortie

Le **fenêtrage** permet de contrôler comment regrouper des enregistrements qui ont une même clé pour des opérations d'agrégations ou des jointures.



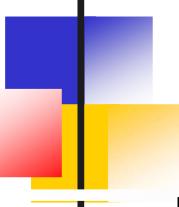
State store

Certaines applications (stateful) nécessitent de conserver un état pour grouper, joindre, agréger

Kafka Streams fournit des ***State stores*** qui permettent aux applications de stocker et interroger des états

On peut y définir des *interactive queries* permettant un accès en lecture à ces données.

Les state store sont implémentés sous forme de topic Kafka avec la stratégie de compactage des logs



Tâches et partitions

Kafka Streams utilise les concepts de partitions de flux et de tâches pour implémenter le parallélisme.

Ces concepts sont basés sur les partitions Kafka.

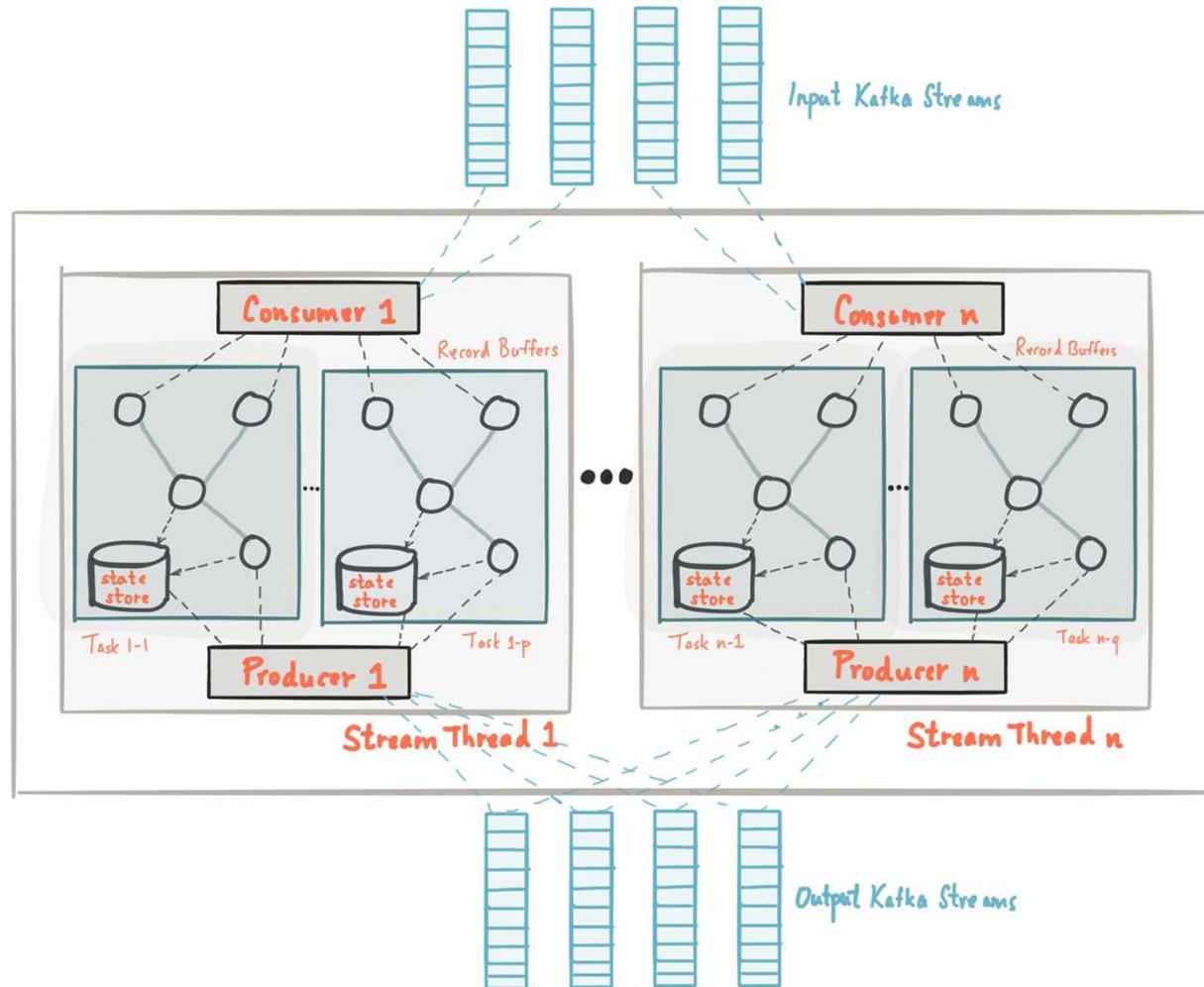
Partition de flux :

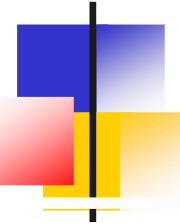
- Chaque partition de flux est une séquence totalement ordonnée d'enregistrements de données et correspond à une partition d'un topic Kafka.
- Un enregistrement dans le flux correspond à un message Kafka du topic.
- Les clés des enregistrements de données déterminent le partitionnement.

Les tâches permettent la scalabilité.

- Kafka Streams 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 jamais
- Les tâches instantient leur propre topologie de processeurs
- Elles ont leur propre StateStore

Architecture et scalabilité





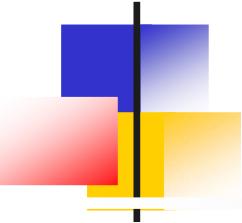
Modèle de threads

Kafka Streams permet de configurer le nombre de threads que la bibliothèque peut utiliser 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.

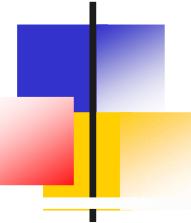
Le scaling consiste à démarrer des instances supplémentaires de l'application. Kafka Streams se charge de distribuer les partitions entre les tâches qui s'exécutent dans les instances de l'application.

En démarrant autant de threads de l'application qu'il y a de partitions dans le topic Kafka en entrée, on s'assure que chaque thread (ou plutôt, les tâches qu'il exécute) ait au moins une partition d'entrée à traiter.



KafkaStream

Concepts
Streamiz
ksqlDB

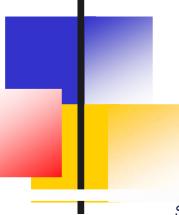


Introduction

Streamiz.Kafka.Net a comme objectif de fournir les mêmes fonctionnalités que Kafka Streams.

Typiquement, une application Streamiz :

- Définit une configuration via StreamConfig (bootstrap-servers, Serdes, ApplicationId)
- Construit une topologie via StreamBuilder
- Démarré l'application



Exemple (1)

```
static async Task Main(string[] args)
{
    // Stream configuration
    var config = new StreamConfig<StringSerDes, StringSerDes>();
    config.ApplicationId = "test-app";
    config.BootstrapServers = "localhost:9092";

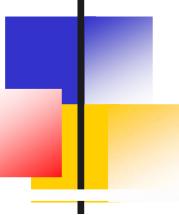
    StreamBuilder builder = new StreamBuilder();

    // Stream "test" topic with filterNot condition and persist in "test-output" topic.
    builder.Stream<string, string>("test")
        .FilterNot((k, v) => v.Contains("test"))
        .To("test-output");

    // Create a table with "test-ktable" topic, and materialize this with in memory store named "test-store"
    builder.Table("test-ktable", InMemory.As<string, string>("test-store"));

    // Build topology
    Topology t = builder.Build();

}
```

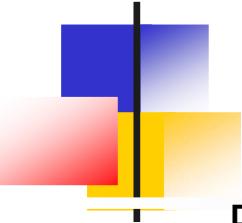


Exemple (2)

```
// Create a stream instance with topology and configuration
KafkaStream stream = new KafkaStream(t, config);

// Subscribe CTRL + C to quit stream application
Console.CancelKeyPress += (o, e) =>
{
    stream.Dispose();
};

// Start stream instance with cancellable token
await stream.StartAsync();
}
```



Configuration

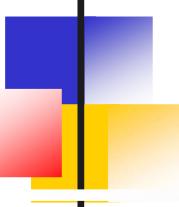
Paramètres requis :

- **ApplicationId** : Identifiant unique de l'application (groupe de consommateurs)
- **BootstrapServers** :

Paramètres optionnels :

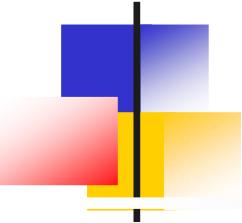
- **Guarantee** : ProcessingGuarantee.AT_LEAST_ONCE (défaut) ou ProcessingGuarantee.EXACTLY_ONCE
- **DefaultTimestampExtractor** : Extrait le timestamp à partir de ConsumeResult. Par défaut le timestamp du message Kafka
- **DefaultKeySerDes**, **DefaultValueSerDes** : Les sérialiseur/désérialiseur par défaut pour les clés et les valeurs
- **NumStreamThreads** : Nombre de threads à utiliser
- **ClientId** : Un préfixe utilisé pour nommer les producteurs et consommateurs internes
- **TransactionalId**, **TransactionTimeout** : Si exactly Once
- **CommitIntervalMs**, **PollMs**, **MaxPollRecords**, **MaxPollInterval**, ...
- **SchemaRegistryUrl**, **AutoRegisterSchemas**

Les configurations de Producteur et des consommateurs peuvent également être spécifiées



Exemple Configuration

```
var config = new StreamConfig<StringSerDes, StringSerDes>();
config.ApplicationId = "test-app";
config.BootstrapServers = "192.168.56.1:9092";
config.SaslMechanism = SaslMechanism.Plain;
config.SaslUsername = "admin";
config.SaslPassword = "admin";
config.SecurityProtocol = SecurityProtocol.SaslPlaintext;
config.AutoOffsetReset = AutoOffsetReset.Earliest;
config.NumStreamThreads = 1;
config.SchemaRegistryUrl = "http://localhost:8081";
config.BasicAuthUserInfo = "user:password";
config.BasicAuthCredentialsSource = 0;
```

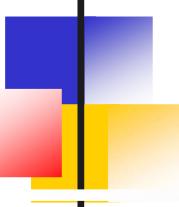


Processseurs stateless

Les processseurs stateless ne nécessitent pas de StateStore.

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

Les processseurs prennent donc un argument optionnel *queryableStoreName*



Quelques processeurs stateless

Filter, NotFilter : 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

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

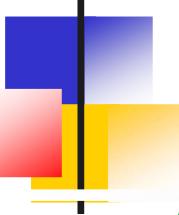
GroupsBy : Regroupe les enregistrements par une nouvelle clé. Pré-requis pour faire de l'agrégation

GroupsByKey : Regroupe en utilisant la clé

SelectKey : Permet de définir une nouvelle clé

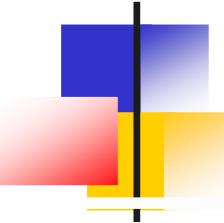
MapAsync, MapValuesAsync, FlatMapAsync, FlatMapValuesAsync, ForEachAsync : Opérations asynchrones sur le modèle requête réponse. Cas d'usage : Enrichir les données avec un système externe

Process, Transform, TransformValues : Hors DSL. Permet de définir ses propres processeurs ou transformateurs



Exemples

```
// Branch
IKStream<string, string> stream = ....;
var branches = stream.Branch(
    (k,v) => k.StartsWith("A"),
    (k,v) => k.StartsWith("B"),
    (k,v) => true); // DLQ pattern
// branches[0] contains all records whose keys start with "A"
// branches[1] contains all records whose keys start with "B"
// branches[3] contains other records
// Filtre
IKStream<string, string> stream = ....;
IKTable<string, string> table = ...;
stream.Filter((k, v) => v.Contains("test"))
table.Filter((k, v) => v.Contains("test"))
// FlatMap
stream
    .FlatMap((k, v) =>
{
    List<KeyValuePair<string, long>> results = new List<KeyValuePair<string, long>>();
    results.Add(KeyValuePair.Create(v.ToUpper(), 100L));
    results.Add(KeyValuePair.Create(v.ToUpper(), 900L));
    return results;
})
// Map
stream.Map((k,v) => KeyValuePair.Create(v.ToUpper(), k.ToUpper()))
```



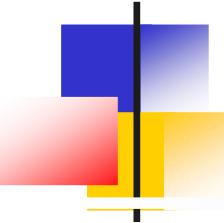
Processeurs stateful

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

Les stores sont fournis :

- soit en argument du processeur
- Soit le processeur s'applique sur une *WindowStore*

Pour appliquer un processeur stateful, il est généralement nécessaire de grouper les enregistrement avec les opérateurs stateless ***GroupBy****



Types de store

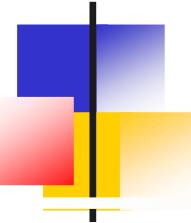
Les stores sont associés ou non à une fenêtre temporel et à un Serde permettant de sérialiser et désérialiser les données

2 types de stores sont fournis par Streamiz :

- inMemory
- RockDB persitant

Les stores fournissent également des fonctions de cache et de métriques

Ils permettent également d'effectuer des requêtes

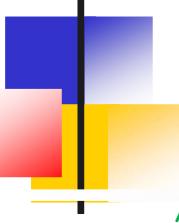


Quelques processeurs stateful

Count : A partir d'un *GroupedStream*, compte les enregistrements

Aggregate : A partir d'un *GroupedStream*, agrège les enregistrements. Généralisation de Reduce qui permet à la valeur agrégée d'avoir un type différent de celui des valeurs d'entrée

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



Exemples

```
// Rolling count
var groupedStream = builder
    .Stream<string, string>("test")
    .GroupBy((k, v) => k.ToUpper());

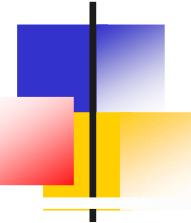
var table = groupedStream.Count(InMemory<string, long>.As("count-store"));

// Windowed count
var groupedStream = builder
    .Stream<string, string>("topic")
    .GroupByKey();

var countStream = groupedStream
    .WindowedBy(TumblingWindowOptions.Of(2000))
    .Count(m);

// Windowed Aggregate
var groupedStream = builder
    .Stream<string, string>("topic")
    .GroupByKey();

var aggStream = groupedStream
    .WindowedBy(TumblingWindowOptions.Of(2000))
    .Aggregate(
        () => 0,
        (k, v, agg) => Math.Max(v.Length, agg),
        m);
```



Jointures

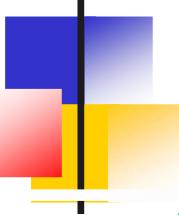
Les jointures sont également des opérations stateful (Conserver les clés pas encore jointes)

On retrouve les mêmes concepts qu'en SQL :
inner, Left, outer

Les jointures s'effectuent sur les clés des enregistrements des 2 flux

Les jointures possibles sont :

- Kstream / Kstream -> Kstream (Windowed store)
- Kstream / Ktable -> Kstream (Non Windowed store)
- Kstream / GlobalTable -> Kstream (Non Windowed store)
- Ktable / Ktable -> Ktable (Non windowed)



Exemples

```
// Inner join Kstream, KStream
var stream1 = builder.Stream<string, string>("topic1");
var stream2 = builder.Stream<string, string>("topic2");
// La jointure s'effectue sur la clé v1.key == v2.key
stream1.Join(stream2,
             (v1, v2) => $"{v1}-{v2}",
             JoinWindowOptions.Of(TimeSpan.FromMinutes(1))).To("output-join");

// Left join Kstream KTable
var stream1 = builder.Stream<string, string>("topic1");
var table = builder.Table<string, string>("topic2", InMemory<string, string>.As("table-store"));

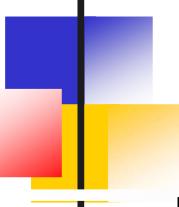
stream1.LeftJoin(table, (v1, v2) => $"{v1}-{v2}").To("output-join");

// Inner join Kstream KGlobalTable
var stream1 = builder.Stream<string, string>("topic1");
var table = builder.GlobalTable<string, string>("topic2", InMemory<string, string>.As("table-store"));

stream1.Join(table,
            (k,v) => k.ToUpper(),
            (v1, v2) => $"{v1}-{v2}").To("output-join");

// Outer join Ktable Ktable
var table1 = builder.Table<string, string>("topic2", InMemory<string, string>.As("table1-store"));
var table2 = builder.Table<string, string>("topic2", InMemory<string, string>.As("table2-store"));

var tableJoin = table1.OuterJoin(table2, (v1, v2) => $"{v1}-{v2}");
```



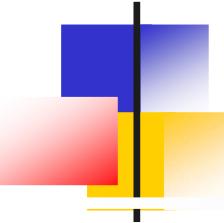
KTable et requêtes

Même si Streamiz n'implémente pas complètement les Requêtes interactives de KafkaStream, on peut facilement reproduire le comportement :

- Construire une KTable à partir d'un topic
- Périodiquement, interroger les données à partir de la clé

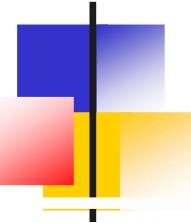
Exemple :

```
// Création d'un KTable basé sur le topic 'coursier-disponibles'  
var coursiersDisponiblesTable = builder.Table<string,  
RegionCoursier>("coursier-disponibles",  
    new StringSerDes(), new JsonSerDes<RegionCoursier>());  
  
while (true) {  
    // Récupérer le Store local associé à la table  
    var store = stream.Store(coursiersDisponiblesTable.QueryableStoreName,  
QueryableStoreTypes.KeyValueStore<string, RegionCoursier>());  
    // Requête  
    var coursiersDisponibles = store.Get(region);  
}
```



KafkaStream

Concepts
Streamiz
ksqldb

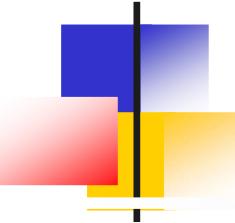


ksqldb

ksqldb fournit donc une couche d'abstraction SQL permettant de manipuler les *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.

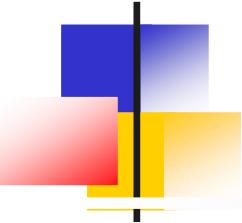


ksqldb

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*

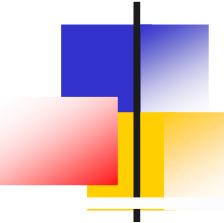


ksqldb

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 :

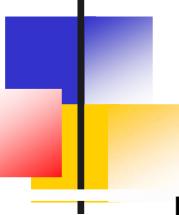
```
CREATE STREAM pageviews (viewtime BIGINT, userid VARCHAR, pageid  
VARCHAR)  
WITH (KAFKA_TOPIC='pageviews', VALUE_FORMAT='JSON');
```

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

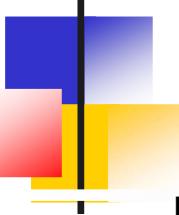
```
CREATE STREAM important_events AS SELECT * FROM source_stream WHERE
event_type = 'important';
```

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

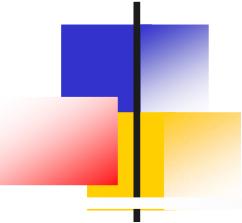
```
SELECT user_id, COUNT(*) AS event_count FROM event_stream GROUP BY user_id;
```

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

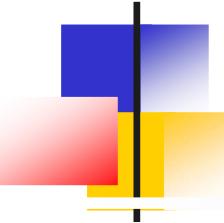
Configuration des listeners

SSL/TLS

Authentification via SASL

ACLs

Quotas

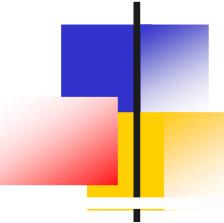


Introduction

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

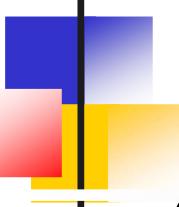
Naturellement, dégradation des performances avec SSL



Listeners

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

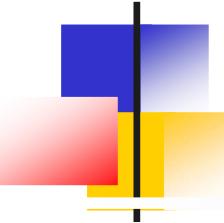
Chaque listener peut être configuré pour authentifier les clients et pour garantir que le trafic entre le serveur et le client est crypté.



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 :

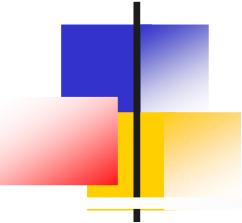
```
listeners=CLIENT://:9092,BROKER://:9095
```

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

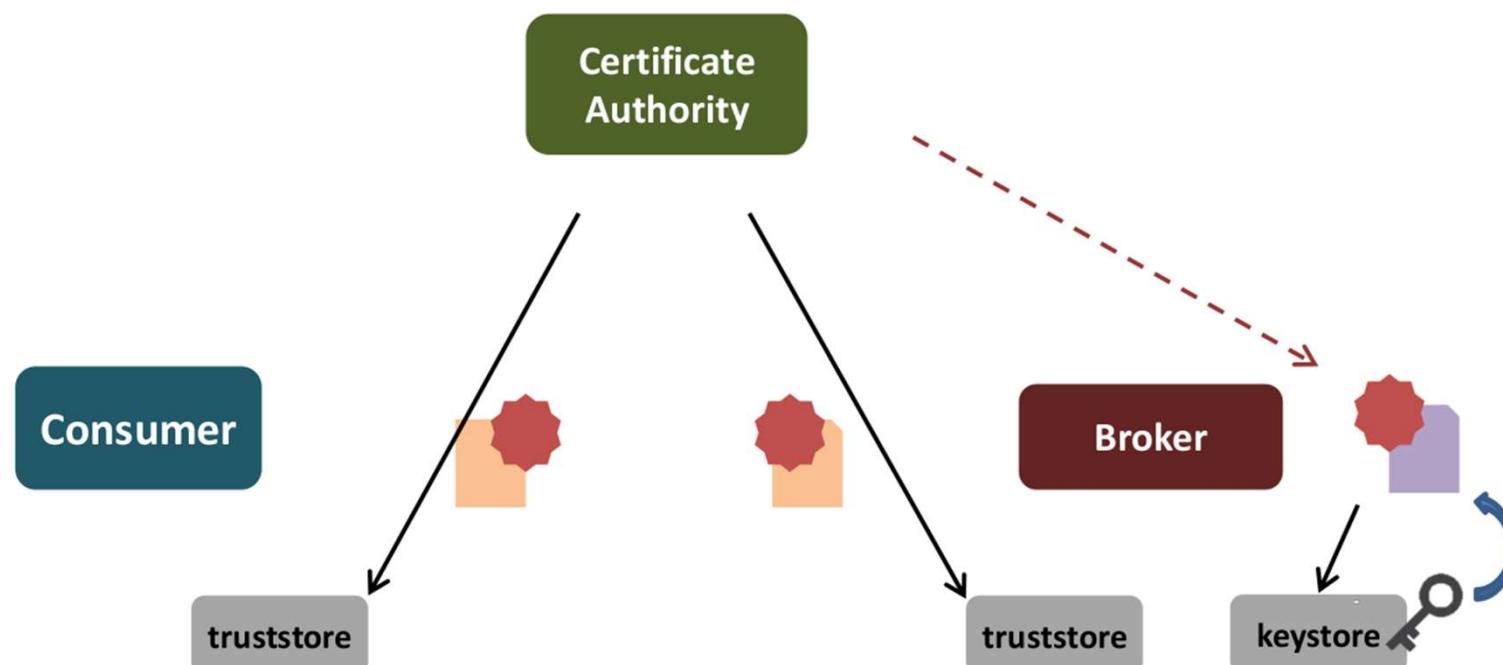
Authentification via SASL

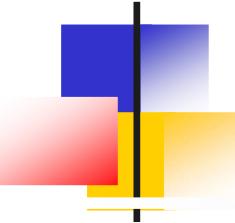
ACLs

Quotas

Certificats

SSL pour le cryptage et l'authentification

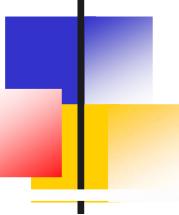




Configuration TLS et Vérification d'hôte

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

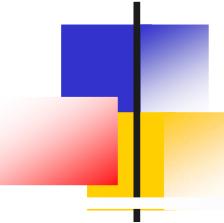


Configuration du broker

server.properties :

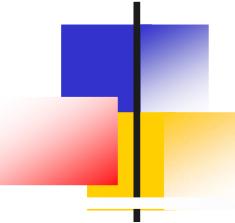
```
listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

```
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

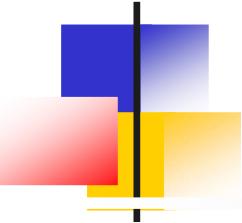
```
security.protocol=SSL  
ssl.truststore.location=/var/private/ssl/clien  
t.truststore.jks  
ssl.truststore.password=clipass
```



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
```

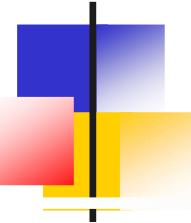


Sécurité

Configuration des listeners
SSL/TLS

Authentification via SASL

ACLs
Quotas



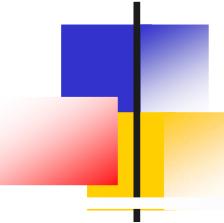
SASL

Simple Authentication and Security Layer pour l'authentification

Kafka utilise JAAS pour la configuration SASL.

Mécanismes:

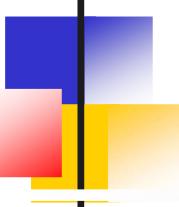
- GSSAPI: Kerberos
- SCRAM-SHA-256, SCRAM-SHA-512: hashed passwords
- PLAIN: username/password en clair
- OAUTHBEARER (Depuis Kafka 3.x)



Configuration JAAS

La configuration JAAS s'effectue :

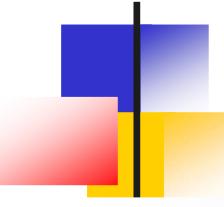
- Soit via un **fichier jaas**
- La propriété JVM
java.security.auth.login.config référence l'emplacement du fichier:
- Le fichier contient une section **KafkaServer** pour l'authentification auprès d'un broker
- Soit par la propriété :
listener.name.<listener-name>.<mechanism>.sasl.jaas.config



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";  
};
```

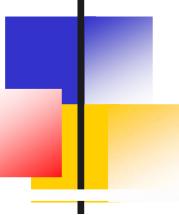


Propriété *sasl.jaas.config*

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";
```



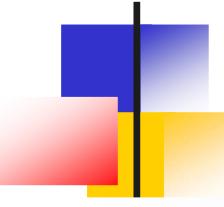
Mécanismes

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



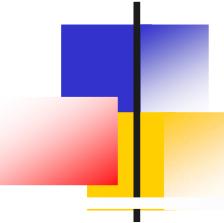
SASL PLAIN

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

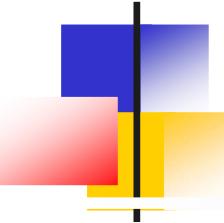
```
KafkaClient {  
    org.apache.kafka.common.security.plain.PlainLoginModule  
    required  
    username="alice"  
    password="alice-secret"  
}; ;
```

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 :

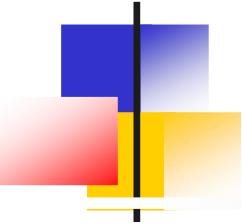
sasl.server.callback.handler.class

et

sasl.client.callback.handler.class.

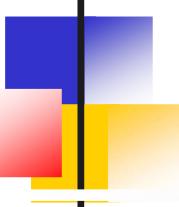
Possibilité d'intégrer des *callback handler* utilisant des serveurs d'authentification externes pour la vérification du mot de passe via

sasl.server.callback.handler.class



Sécurité

Configuration des listeners
SSL/TLS
Authentification via SASL
ACLs
Quotas



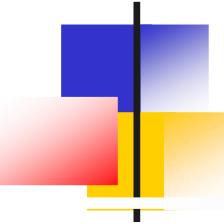
Introduction

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

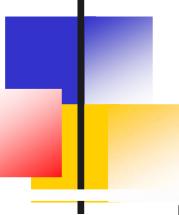
Dès lors que les ACLs sont activées, les contrôleurs doivent s'identifier pour communiquer.

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";
};
```



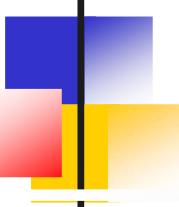
Autorisation

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 :

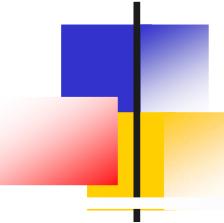
Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP



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
```



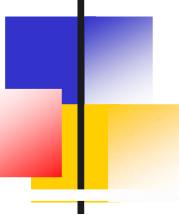
Ressources et opérations

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 :

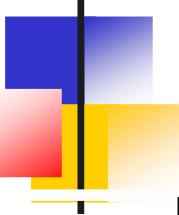
User:Alice has Allow permission for Write to Prefixed Topic:customer from 192.168.0.1



Règles

AclAuthorizer autorise une action s'il n'y a pas d'ACL de DENY qui correspond 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

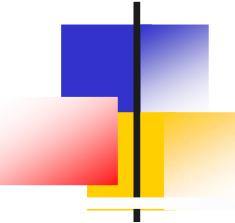
Brokers : ***Cluster:ClusterAction*** pour autoriser les requêtes de contrôleur et les requêtes de fetch des répliques.

Producteurs simples : ***Topic:Write***

- idempotents sans transactions : ***Cluster:IdempotentWrite***.
- Transactionnels : ***TransactionalId: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.

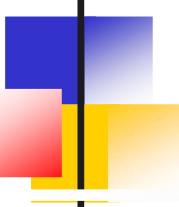
Clients admin : ***Create, Delete, Describe, Alter,***
DescribeConfigs, AlterConfigs .



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 :

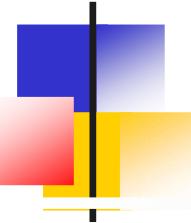
- ***super.users*** : Permet de définir les utilisateurs ayant droit à tout
- ***allow.everyone.if.no.acl.found=true*** : Tous les utilisateurs ont accès aux ressources sans ACL.



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
```

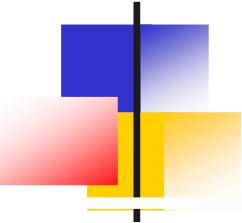


Audit

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

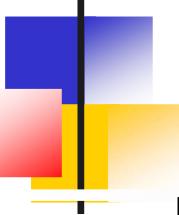
La configuration s'effectue dans
conf/log4j.properties

- 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

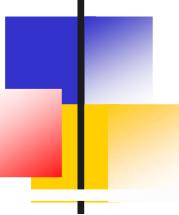
Kafka permet d'appliquer des quotas sur les requêtes pour contrôler les ressources du broker utilisées par les clients.

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

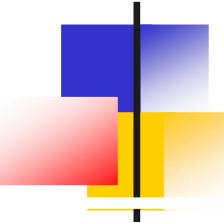


Quota de bande passante

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.

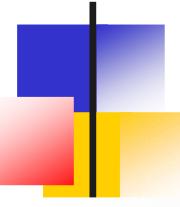
```
kafka-configs.sh --bootstrap-server $BOOT --alter  
--add-config  
'producer_byte_rate=1024,consumer_byte_rate=1024'  
' --entity-type users --entity-name  
<authenticated-principal> --command-config  
/tmp/client.properties
```



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
```



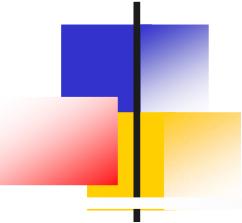
Application des quotas

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

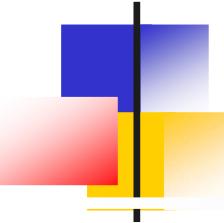
Selon le client (récent ou non), le client peut soit respecter ce délai, soit l'ignorer.



Annexes

Stockage et rétention des partitions

Monitoring et JMX

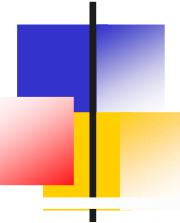


Introduction

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

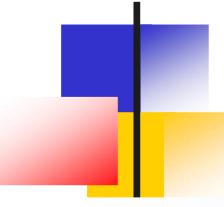


Allocation des partitions

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

L'administrateur Kafka configure une période de rétention pour chaque topic

- Soit une durée
- Soit un volume

Pour accélérer la purge, Kafka utilise les ***segments***

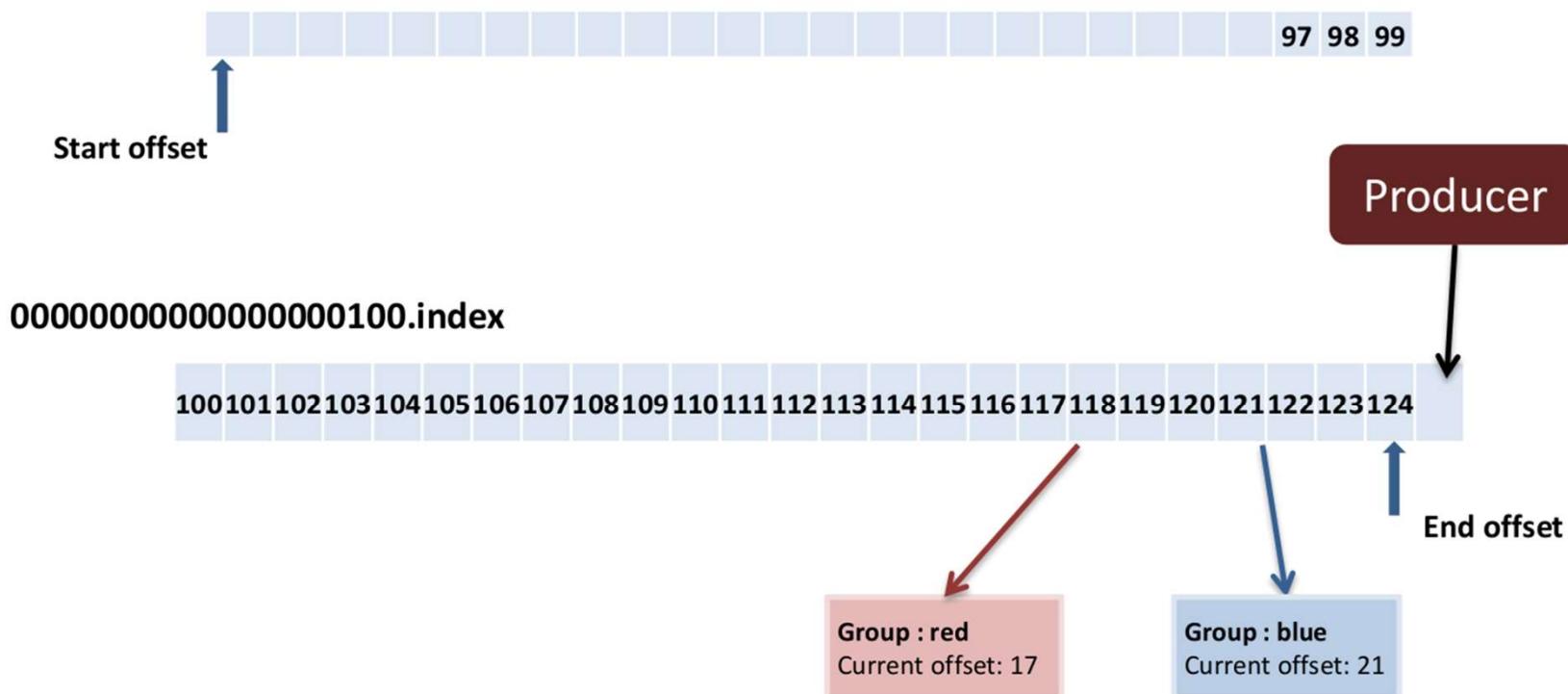
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas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

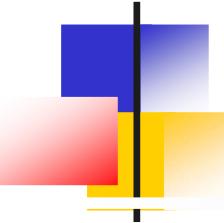
Segments

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)



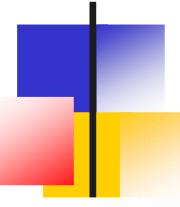


Indexation

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



Principales configurations

log.retention.hours (défaut 168 : 7 jours),

log.retention.minutes (défaut null),

log.retention.ms (défaut null, si -1 infini)

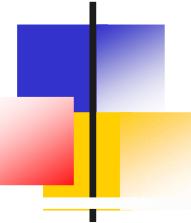
Période de rétention des vieux segment avant de les supprimer

log.retention.bytes (défaut -1)

La taille maximale du log

offsets.retention.minutes (défaut 10080 : 7 jours)

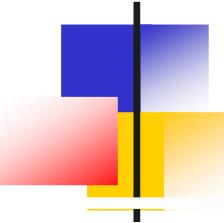
Le nombre de minutes pour conserver l'index d'offset d'un groupe n'ayant plus de consommateurs



Compactage

Kafka autorise également la stratégie de rétention ***compact***, qui ne stocke que la valeur la plus récente pour chaque clé du *topic*.

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

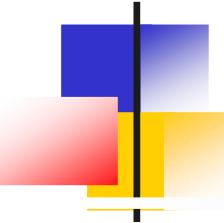


Nettoyage des logs

Propriété log.cleanup.policy, 2 stratégies disponibles :

- ***delete*** (défaut) :
 - Suppression des vieux segments en fonction de l'âge et la taille du log (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 and compact)

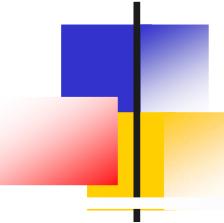


Stratégie *delete*

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)

=> Meilleur contrôle de l'usage disque



Stratégie compact

2 propriétés de configuration importante 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.

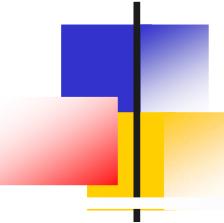
Le pool est configurable :

- *log.cleaner.enable* : doit être activé si stratégie compact
- *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

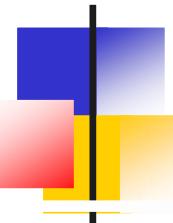


Exemple *compact*

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*

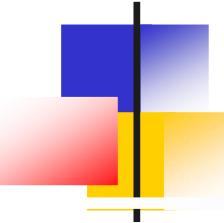
Consommateurs «à jour» (offset courant dans le segment actif), récupèrent tous les messages

- 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

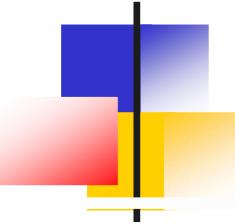
segment.ms (7 jours)

segment.byte (1G)

min.compaction.lag.ms (défaut 0)

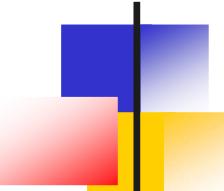
delete.retention.ms (défaut 1 jour)

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



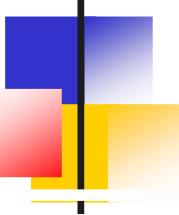
Annexes

Stockage et rétention des partitions
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 $\neq 1$
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	269



Métriques clients

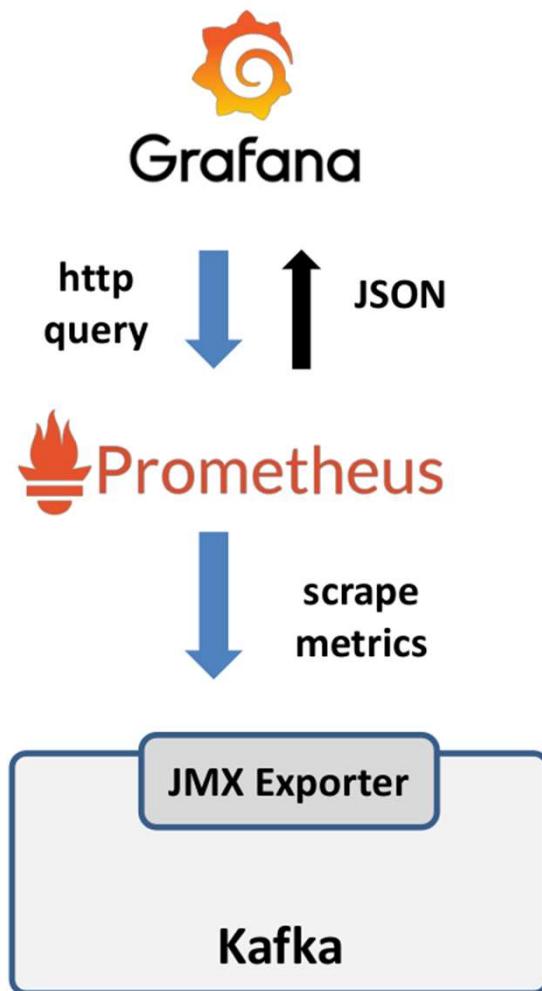
Producteur

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

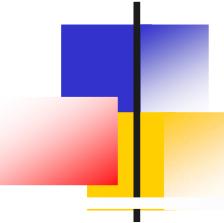


Tdb dispo :

<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)

...