





Kafka : Messagerie distribuée avec Apache Kafka

David THIBAU – 2020

david.thibau@gmail.com



Agenda

Introduction à Kafka

- Big-Data / Stream-data
- Kafka, ses cas d'usage
- Architecture et Rationale
- APIs

Installation

Pré-requis et ensemble Zookeeper

Broker Kafka

Cluster Kafka

Kafka APIs

ProducerAPI

Consumer API

Connect API

Kafka Streams

- Introduction et Architecture
- Création d'une topologie
- DSL

Pipeline CI/CD

- Rappels CI/CD
- Tests d'acceptance dans le pipeline
- Techniques d'intégration
- Publication des rapports



Introduction

Tests d'acceptance BDD et agilité

Cucumber
Installation



Origine



Plateforme de streaming distribuée

Kafka a donc trois capacités clés:

- Publier et s'abonner à des flux d'enregistrements, similaires à une file d'attente de messages ou à un système de messagerie d'entreprise.
- Stocker les flux d'enregistrements de manière durable et tolérante aux pannes.
- Traiter les flux d'enregistrements au fur et à mesure qu'ils se produisent.



Concepts de base

Kafka s'exécute en tant que cluster sur un ou plusieurs serveurs pouvant s'étendre sur plusieurs centres de données.

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

Chaque enregistrement se compose d'une clé, d'une valeur et d'un horodatage.

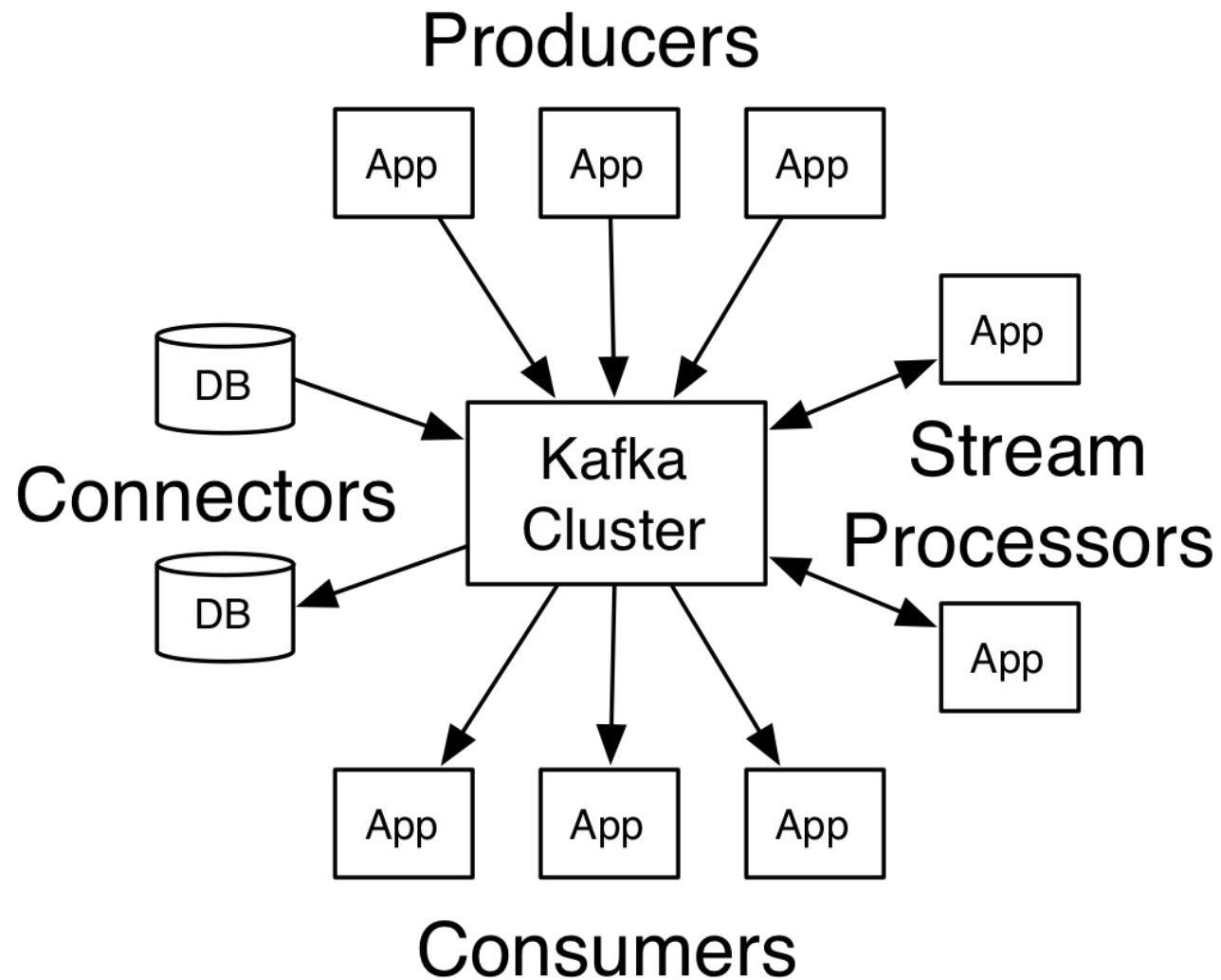


APIs

Kafka propose 4 API :

- 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 réutilisables (BD, STDOUT, ...)

APIs





Protocole Client/Serveur

Dans Kafka, la communication entre les clients et les serveurs se fait avec un protocole TCP simple, performant et indépendant de la langue.

Ce protocole est versionné et maintient une compatibilité ascendante avec une version plus ancienne.

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



Implémentation cliente

Apache ne fournit qu'une implémentation des clients en Java. Les 4 APIs sont donc disponibles sous forme de jar.

Pour les autres technologies, voir :

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

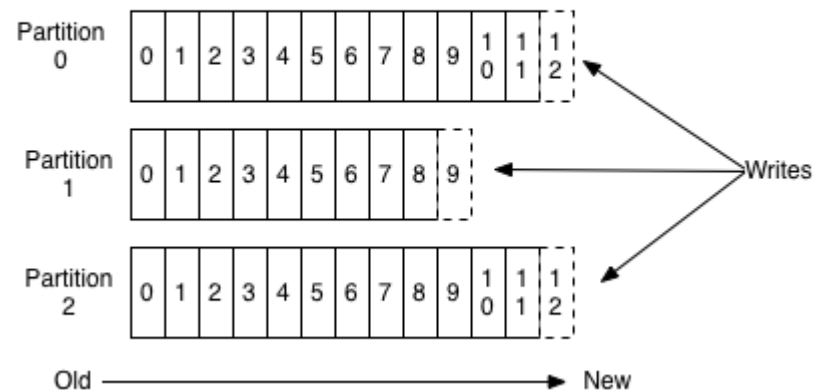
Topic

Les records sont publiés vers des *topics*.

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

Pour chaque *topic*, le cluster Kafka conserve un journal partitionné

Anatomy of a Topic





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.

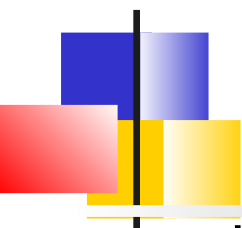


Offset consommateur

Les seules métadonnées conservées par consommateur sont l'offset du consommateur dans le journal.

L'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, retraiter les données les plus anciennes ou ne traiter qu'à partir de la date courante.



Distribution des partitions

Les partitions du journal sont réparties sur les serveurs du cluster.

Chaque serveur gère les données et les requêtes de consommation des partitions.

Chaque partition est répliquée sur un nombre configurable de serveurs pour la tolérance aux pannes.

Pour chaque partition répliquée, un des serveurs agit comme **maître**. Les autres comme **suiveurs**

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



Apports des partitions

Le partitionnement des enregistrements apporte 2 fonctionnalités :

- Les capacités de stockage d'un topic sont scalés. Les ressources de stockage des différents serveurs sont utilisées
- Le parallélisme, les consommateurs



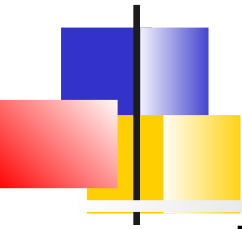
Géo-réplication

Kafka MirrorMaker fournit un support pour la géo-réplication pour vos clusters.

- Les messages sont répliqués sur plusieurs centres de données ou régions

Cela peut être utilisé pour :

- Réplication passive : backup et recovery
- Réplication active : Lecture vers la source de donnée la plus proche



Routing des messages

Les producteurs sont responsable 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 : une 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 consommateur peuvent se trouver dans des processus distincts ou sur des machines distinctes.

=> Scalability et Tolérance aux fautes



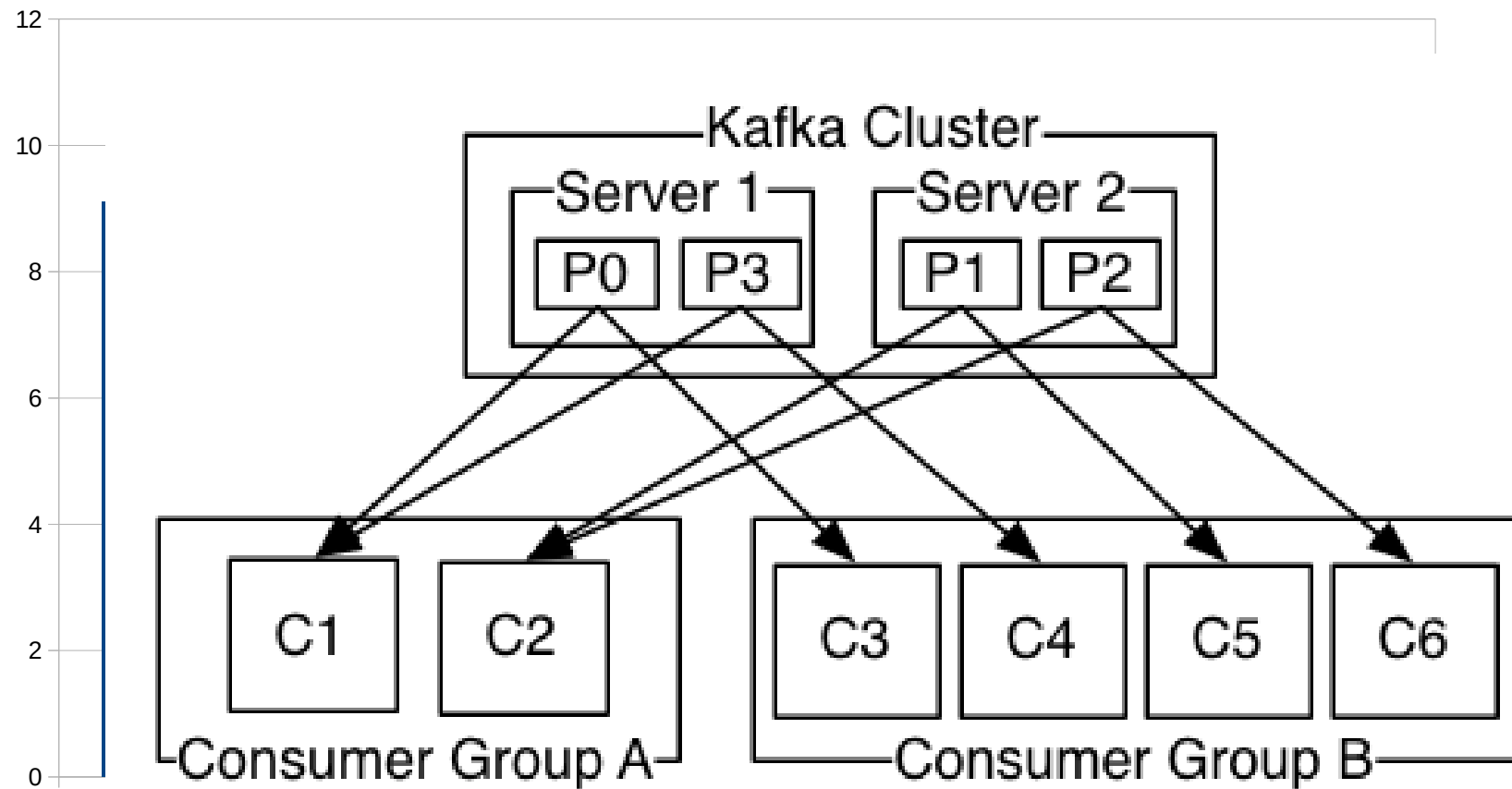
Instance vs Partition

Kafka répartit les partitions sur les instances de consommateur de telle sorte que chaque instance soit le consommateur exclusif d'une seule partition à tout moment.

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.

Example



ne
ne
ne



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 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





Kafka vs Traditional Message broker

Kafka n'a que le modèle PubSub mais grâce au concept de groupe de consommateur, ce modèle est scalable

Kafka offre une garantie plus forte sur l'ordre de livraison des messages



Kafka comme système de stockage

Les enregistrements sont écrits et répliqués sur le disque. Kafka permet aux producteurs d'attendre l'acquittement d'une écriture signifiant que la réplication de la donnée est effectuée.

La structure de stockage de Kafka est très scalable. Kafka fonctionne de la même manière avec 50 Ko ou 50 To de données sur le serveur.

Kafka peut être considéré comme une sorte de système de fichiers distribué dédié au stockage, à la réplication et à la propagation de journaux de validation avec de très hautes performances et une faible latence.

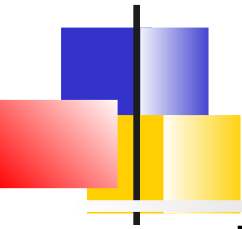


Kafka pour le traitement de flux

Kafka permet de définir des processeurs de flux : composants qui

- consomment des flux continus de données à partir de topics,
- effectuent un certain traitement
- produisent des flux continus de données vers des topics de sortie.

Ces processeurs permettent d'effectuer des transformations complexes comme des calculs d'agrégations ou la fusion de 2 flux distincts



La combinaison des fonctionnalités de messagerie, de stockage et de traitement de flux permet la mise en place d'applications de streaming capable de traiter les **messages passés ou futurs** contenant des données critiques (réplication et tolérance aux défaillances)



Cas d'usage typiques

Système de messagerie simple : Comme ActiveMQ ou RabbitMQ

Suivi des activités d'un site Web : Cas d'usage d'origine.
Enormément de données à traiter

Surveillance de métriques : Centralisé des métriques
provenant d'un système distribué, les agréger

Agrégation de traces : Agréger les lignes d'un fichier de trace

Traitement flux et BigData : Exécuter les différentes étapes de
traitement nécessaire dans une approche BigData

Journal de transaction : Enregistrement des transactions
permettant de rejouer les transactions pour la réplication ou
la recovery



Installation

Pré-requis et ensemble Zookeeper
Broker Kafka
Cluster Kafka



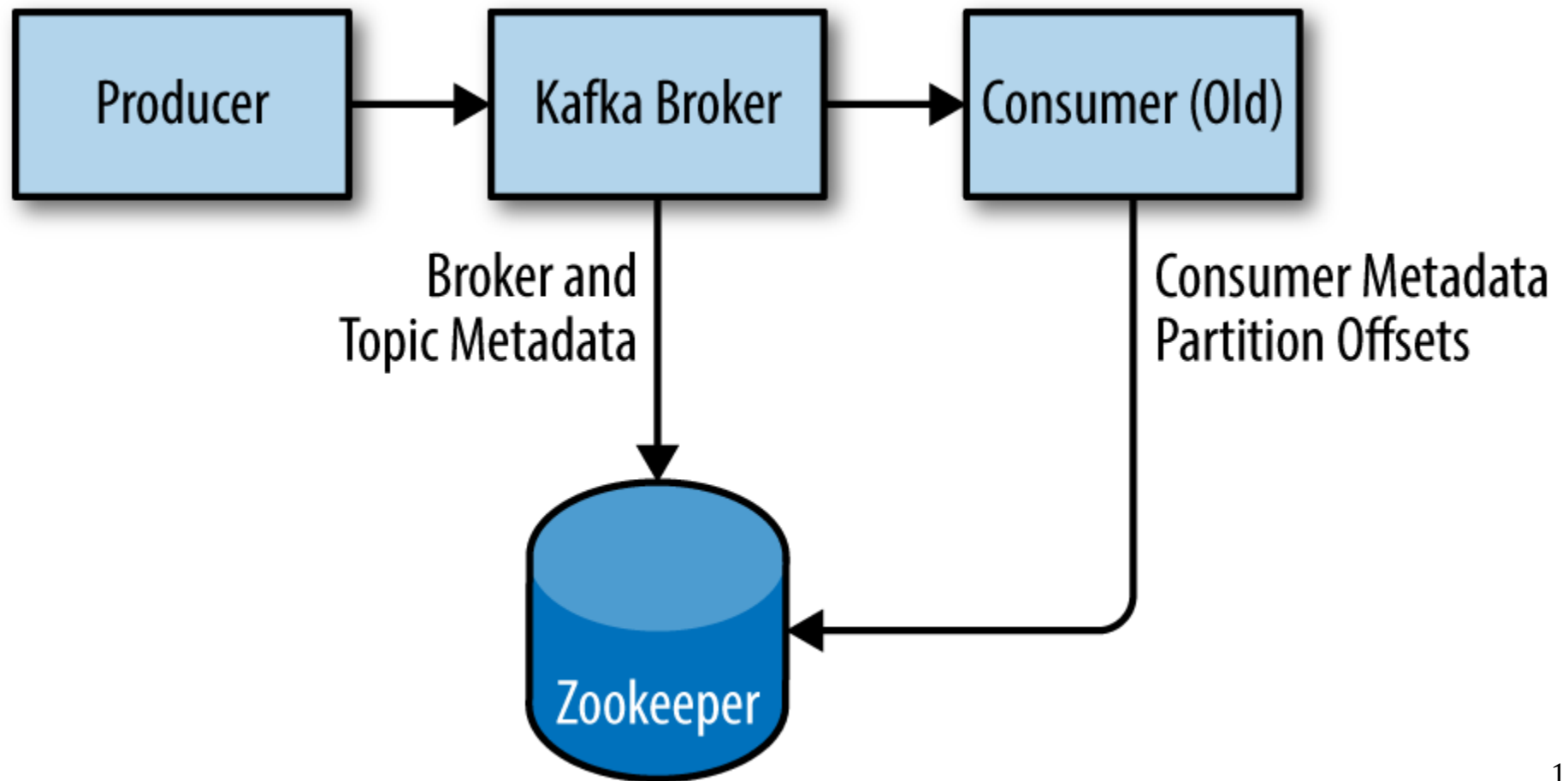
Pré-requis

OS recommandé Linux

JDK8+

Apache Zookeeper

Kafka et Zookeeper





Zookeeper

Kafka contient des scripts permettant de démarrer une instance de Zookeeper mais il est préférable d'installer une version complète à partir de la distribution officielle de Zookeeper (Version courante : 3.5)

Version standalone :

```
# tar -zxf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```




Vérification Zookeeper

```
# telnet localhost 2181
Trying ::1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 4
Connection closed by foreign host.
#
```



Ensemble Zookeeper

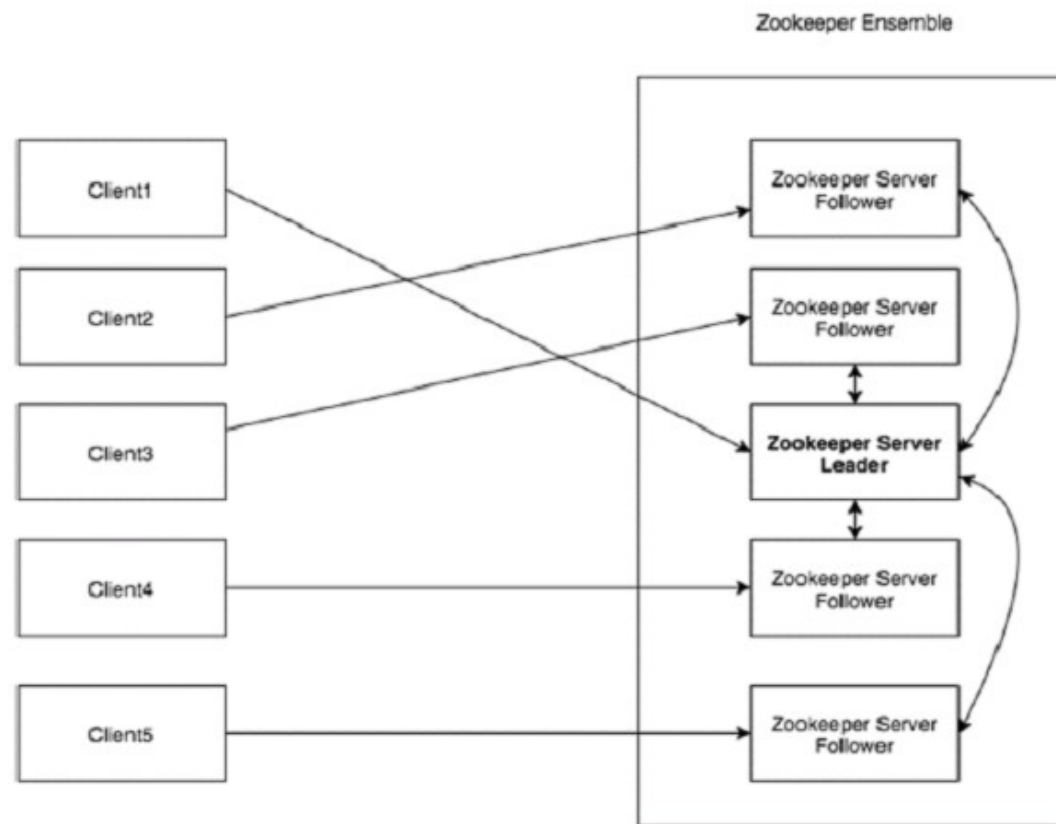
Un cluster Zookeeper est appelé un **ensemble**

Il contient généralement un nombre impair de nœuds (algorithme de consensus basé sur la notion de quorum).

Son nombre dépend du niveau de tolérance aux pannes voulu :

- 3 nœuds => 1 défaillance
- 5 nœuds => 2 défaillances

Architecture ZooKeeper





Configuration d'un ensemble

Les serveurs doivent partager une configuration commune listant les serveurs et chaque serveur doit contenir un fichier *myid* dans son répertoire de données contenant son identifiant

Exemple de config :

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```



Propriétés de config

initLimit nombre de tick autorisé pour la connexion des suiveurs au leader

syncLimit nombre de tick autorisé pour la synchronisation suiveur/leader

tickTime : L'unité de temps en ms

La configuration répertorie également chaque sous la forme :

server.X = nom d'hôte: peerPort: leaderPort

- ***X*** : numéro d'identification du serveur.
- ***nom d'hôte*** : IP
- ***peerPort*** : Port TCP pour communication entre serveurs
- ***leaderPort*** : Port TCP pour l'élection.

Optionnel :

4lw.commands.whitelist : Les commandes d'administration autorisées



Vérifications ensemble Zookeeper

Se connecter à une instance :

```
./zkCli.sh -server 127.0.0.1:2181
```

Voir le mode (leader ou suiveur) d'une instance

si la commande *stat* est autorisée

```
echo stat | nc localhost 2181 | grep Mode
```



Installation

Pré-requis et ensemble Zookeeper
Broker Kafka
Cluster Kafka



Installation broker

Téléchargement, puis

```
# tar -zxf kafka_2.11-0.9.0.1.tgz
# mv kafka_2.11-0.9.0.1 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# 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*



Vérifications

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 --broker-list 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
```



Configuration broker

broker.id : Chaque broker doit avoir un identifiant entier unique à l'intérieur du cluster

port : Par défaut 9092

zookeeper.connect : Listes des serveurs
Zookeeper sous la forme *hostname:port/path*
path chemin optionnel pour l'environnement chroot

log.dirs : Liste de chemin locaux où kafka stocke les messages

auto.create.topics.enable : Création automatique de topic à l'émission ou à la consommation



Configuration topic

Le fichier `.properties` définit également les valeurs de configuration par défaut concernant les topics

- **`num.partitions`** : Nombre de partitions. *Par défaut 1*
- **`default.replication.factor`** : Nombre de répliques par défaut. *Par défaut 1*
- **`log.retention.ms/minutes/hours`** : Le temps de rétention d'un message. *Par défaut 1 semaine*
- **`log.retention.bytes`** : Critère additionnel pour spécifier une taille par partition pour la rétention
- **`log.segment.bytes`** : Taille d'un segment, i.e. fichier où sont stockés les messages. *Par défaut 1Go*
- **`log.segment.ms`** : Le délai de fermeture d'un segment. Exclusif avec `log.segment.bytes`
- **`message.max.bytes`** : Taille maximale d'un message. *Par défaut 1Mo*



Détail sur le mécanisme d'expiration des messages

Les paramètres de rétention s'appliquent sur des segments et non sur les messages.

Au fur et à mesure de leur arrivée, les messages sont ajoutés au segment courant de la partition. Une fois que le segment a atteint la taille spécifiée par `log.segment.bytes` il est fermé et un nouveau s'ouvre.

Une fois fermé, le segment peut être considéré pour expiration.

=> Une taille de segment petite signifie que les fichiers doivent être fermés et alloués plus souvent, ce qui réduit l'efficacité globale des écritures sur disque.

=> Si le débit d'écriture est faible et la taille de segment important, les messages peuvent être retenus plus longtemps que la valeur de configuration



Choix du nombre de partitions

Une fois un topic créé, on ne peut pas diminuer son nombre de partitions

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



Recommandations pour le matériel

Accès disque : Facteur très important influant principalement sur le débit des producteurs. SSD ?

Capacité disque : Dépend du débit et de la période de rétention

Mémoire : Lecture des segments utilise le cache système.

- Taille dépend du décalage entre l'écriture et la lecture des messages.
- JVM typiquement entre 1Go et 5Go

Réseau : Aussi important que le disque, fortement sollicité pour la réplication. Penser à la compression des messages

CPU : Moins important, le CPU est surtout utilisé pour la compression des messages lors de l'écriture sur le disque



Installation

Pré-requis et ensemble Zookeeper
Broker Kafka
Cluster Kafka



Configuration cluster

Seulement 2 pour la configuration d'un cluster :

- Tous les brokers doivent avoir le même paramètre ***zookeeper.connect***.
- Chaque broker doit avoir une valeur unique pour ***broker.id***.

D'autres paramètres de configuration sont utilisés lorsque exécuter un cluster - en particulier, les paramètres qui contrôlent la réplication.



Nombre de brokers

Le premier facteur à considérer est la capacité de disque requise pour conserver les messages et la quantité de stockage disponible sur un seul courtier.

L'autre facteur à considérer est la capacité du cluster à traiter les requêtes.



Cluster et ensemble Zookeeper

Kafka utilise *Zookeeper* pour stocker des informations de métadonnées sur les brokers, les topics et les partitions.

Les écritures sont peu volumineuses et il n'est pas nécessaire de dédier un ensemble Zookeeper à un seul cluster Kafka.

- => Un seul ensemble pour plusieurs clusters Kafka (en utilisant un chemin Zookeeper chroot pour chaque cluster).



APIs

Producer API

Consumer API

Connect API



Introduction

L'API est simple mais les contraintes applicatives influencent la façon de l'utiliser et 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 ?



Dépendance Maven

```
<dependency>  
  <groupId>org.apache.kafka</groupId>  
  <artifactId>kafka-clients</artifactId>  
  <version>2.4.1</version>  
</dependency>
```

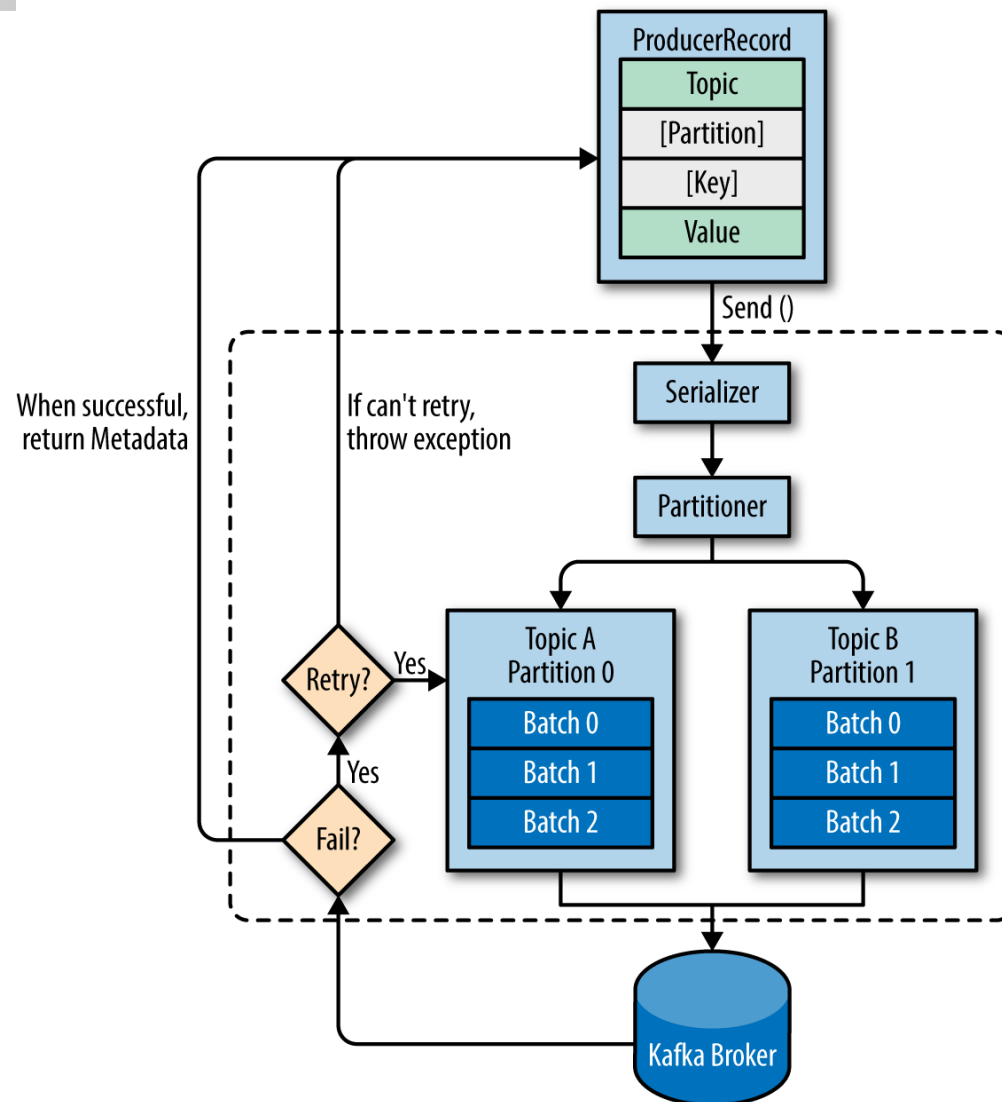


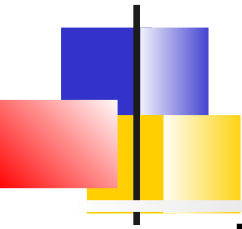
Étapes lors de l'envoi d'un message

L'envoi de message est constitué de plusieurs étapes :

- Création d'un objet *ProductRecord* encapsulant le topic et optionnellement une clé et une partition
- L'objet est sérialisé pour transmission sur le réseau
- Les données sont envoyées à un partitionneur qui détermine la partition de destination, soit à partir de la partition indiquée, soit de la clé du message, soit Round-robin
- Une fois la partition sélectionnée, le message est ajouté à un lot de message destiné à la même partition. Une thread séparé envoi le batch de message
- Lorsque le broker reçoit le message, il renvoie une réponse sous le forme d'un objet *RecordMetadata* encapsulant le topic, la partition, la clé et l'offset
- Si le broker n'arrive pas à écrire le message, il renvoie une erreur et le producteur peut réessayer un certain nombre de fois

Envoi de message





Construire un Producteur

La première étape pour l'envoi consiste à instancier un producteur. Un producteur a 3 propriétés de configurations obligatoires :

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



Example

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

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



ProducerRecord

Représente une paire clé / valeur à envoyer à Kafka.

Il contient le nom du topic, une valeur et éventuellement

- Une clé
- Une partition
- Un timestamp

Quelques constructeurs :

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



Méthodes d'envoi des messages

Il y a 3 façons d'envoyer des messages :

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



Fire And Forget

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

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



Envoi synchrone

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

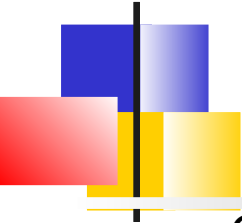
```
try {  
    producer.send(record).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



Envoi asynchrone

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

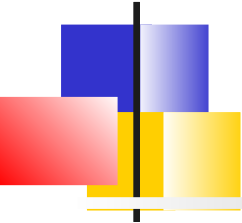
```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Biomedical  
        Materials", "USA");  
producer.send(record, new DemoProducerCallback());
```



Configuration des producteurs

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

- **acks** : contrôle le nombre de réplikas qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie
 - 0 : On attend pas
 - all : On attend toutes les répliques
- **batch.size** : La taille du batch en mémoire pour envoyer les messages
- **linger.ms** : la durée d'attente de messages supplémentaires avant d'envoyer le batch courant.
- **buffer.memory** : Taille buffer pour stocker les messages que l'on ne peut pas envoyé
- **compression.type** : Par défaut, les messages ne sont pas compressés. Valeurs possibles : *snappy* , *gzip* , ou *lz4*
- **Retries** : Si l'erreur renvoyée est de type *Retriable*, le nombre de de tentative de réenvoi. Si > 0 possibilité d'envoi en doublon



Configuration des producteurs (2)

...

- ***client.id*** : Optionnel, pour permettre le suivi des messages
- ***max.in.flight.requests.per.connection*** : Maximum de message en cours de transmission (sans réponse obtenu)
- ***request.timeout.ms***, ***metadata.fetch.timeout.ms*** et ***timeout.ms***: Timeouts pour la réception d'une réponse à un message, pour obtenir des méta-données (leader, etc..) pour obtenir le ack des répliques.
- ***max.block.ms*** : Temps maximum d'attente pour la méthode `send()`. Dans le cas où le buffer est rempli
- ***max.request.size*** : Taille max d'un message
- ***receive.buffer.bytes*** et ***send.buffer.bytes***: Taille des buffers TCP



Garanti sur l'ordre

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

Attention :

- Si *Retries* > 0 et *max.in.flights.requests.per.session* > 1 .
Il se peut que lors d'un renvoi l'ordre initial soit inversé.

=> Si l'ordre et la fiabilité sont importantes, on configure généralement *Retries* > 0 et *max.in.flights.requests.per.session* =1 au détriment du débit global



Propriété Idempotent et transactions

A partir de Kafka 0.11, le `KafkaProducer` prend en charge deux modes supplémentaires: idempotence et transactionnel.

- Un producteur idempotent garantit la livraison unique de chaque message (pas de doublon)
- Un producteur transactionnel permet à une application d'envoyer des messages à plusieurs partitions (et topics!) de façon atomique



Idempotence

Pour autoriser l'idempotence :
`enable.idempotence=true`

Dans ce cas,

- *retries* a par défaut *Integer.MAX_VALUE*
- et *acks* a par défaut *all*.

L'idempotence n'a pas d'incidence sur le code applicatif.

Par contre, l'application ne doit pas essayer de renvoyer elle-même les messages en cas d'erreur



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 eux être configurés pour faire du *read committed*
- L'API *send()* devient bloquante



Exemple

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new
    StringSerializer());
```

```
producer.initTransactions();
```

```
try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", ""+i, ""+i));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // Impossible de rattraper, la seule solution est de fermer le producer and exit.
    producer.close();
} catch (KafkaException e) {
    // Pour les autres exceptions, abort et essayer à nouveau.
    producer.abortTransaction();
}
producer.close();
```



Sérialiseurs

Kafka inclut les classes *ByteArraySerializer* et *StringSerializer* utile pour types basiques.

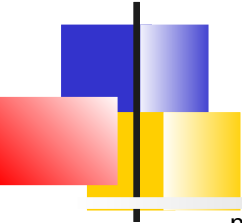
Pour des objets du domaine, il implémenter ses propres sérialiseurs/désérialiseurs en s'appuyant sur des librairies comme Avro, Thrift, Protobuf ou Jackson



Workflow

Le workflow de développement consiste à :

- Implémenter l'interface :
org.apache.kafka.common.serialization.Serializer
- Implémenter l'interface :
org.apache.kafka.common.serialization.Deserializer
- Implémenter l'interface
org.apache.kafka.common.serialization.Serde
manuellement ou en utilisant des méthode
utilitaire de *Serdes* comme :
Serdes.serdeFrom(Serializer<T>, Deserializer<T>)



Exemple sérialiseur s'appuyant sur Jackson

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




Exemple désérialiseur basé sur Jackson

```
public class JsonPOJODeserializer<T> implements Deserializer<T> {
    private ObjectMapper objectMapper = new ObjectMapper();

    private Class<T> tClass;

    /**
     * Default constructor requis par Kafka
     */
    public JsonPOJODeserializer() { }

    @SuppressWarnings("unchecked")
    @Override
    public void configure(Map<String, ?> props, boolean isKey) {
        tClass = (Class<T>) props.get("JsonPOJOClass");
    }

    @Override
    public T deserialize(String topic, byte[] bytes) {
        if (bytes == null)
            return null;

        T data;
        try {
            data = objectMapper.readValue(bytes, tClass);
        } catch (Exception e) {
            throw new SerializationException(e);
        }

        return data;
    }

    @Override
    public void close() { }
}
```



Envoi de message

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

```
String topic = "customerContacts";  
int wait = 500;  
Producer<String, Customer> producer = new  
    KafkaProducer<String, Customer>(props);
```

```
Customer customer = CustomerGenerator.getNext();
```

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



APIs

Producer API
Consumer API
Connect API



Introduction

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

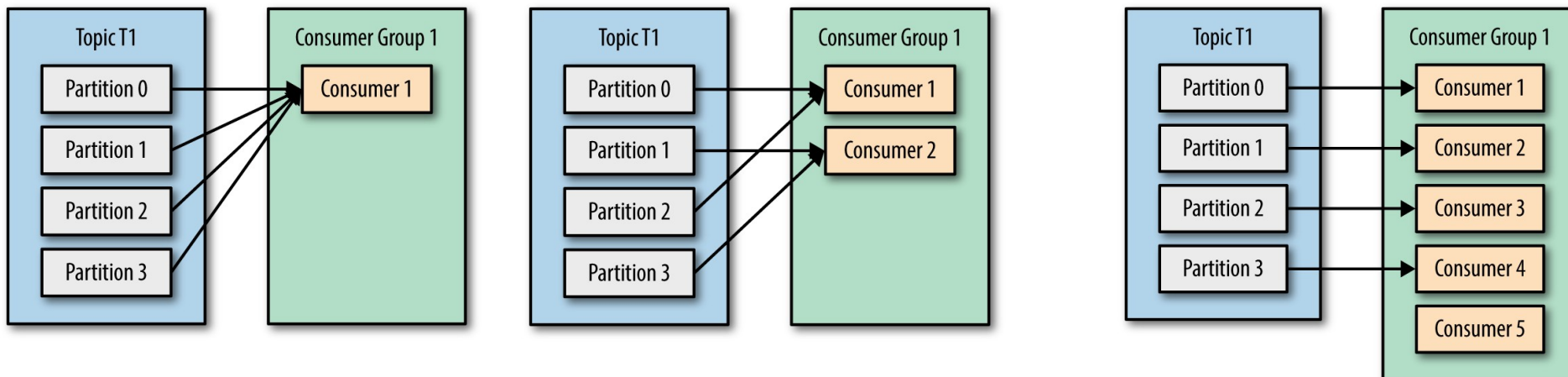
Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relation avec les partitions



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.





Répartition dynamique des consommateurs

Lors de l'ajout d'un nouveau consommateur, celui-ci commence par consommer les messages d'une partition consommés précédemment par un autre consommateur du groupe.

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

Cette répartition dynamique offre la scalabilité et la tolérance aux défaillances mais elle n'est pas spécialement désirable car durant la réaffectation les messages ne sont pas consommés de plus si les consommateurs utilisent des caches ils sont obligés de les rafraîchir



Membership

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).

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



Création de KafkaConsumer

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

3 propriétés doivent être spécifiées dans une classe Properties :

- bootstrap.servers , key.deserializer , et value.deserializer

La propriété group.id n'est pas requise mais généralement utilisée pour spécifier le groupe de consommateur



Abonnement à un topic

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

La méthode ***subscribe()*** prend une liste de topic comme paramètre. Ex :

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

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

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



Boucle de Polling

Typiquement, les applications consommatrices poll continuellement les topics auxquels ils sont abonnés pour récupérer les données

Les objets retournés par poll sont une collection de ConsumerRecord contenant en plus du message :

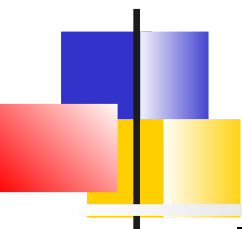
- La partition
- L'offset
- Le timestamp



Exemple

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

        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount)
        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString(4))
    }
}
} finally {
    consumer.close();
}
```



Thread et consommateur

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

=> 1 consommateur = 1 thread

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



Configuration des consommateurs

Les plus importantes propriétés de configuration :

- ***fetch.min.bytes*** : Minimum volume 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 3s
- ***heartbeat.interval.ms*** : L'intervalle d'envoi des heartbeat durant la méthode poll()



Configuration des consommateurs (2)

...

- ***auto.offset.reset*** : ***latest*** (défaut) ou ***earliest***.
Contrôle le comportement du consommateur si il ne détient pas d'offset valide
- ***enable.auto.commit*** : Le consommateur commit les offsets automatiquement. Par défaut *true*
- ***partition.assignment.strategy*** : Stratégie d'affectation des partitions *Range* (défaut), *RoundRobin* ou *Custom*
- ***client.id*** : Une chaîne de caractère utilisé pour les métriques. Par défaut 3s
- ***receive.buffer.bytes*** et ***send.buffer.bytes*** : Taille des buffers TCP



Offsets et Commits

Les consommateurs peuvent suivre leur offset d'une partition en s'adressant à Kafka.

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

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

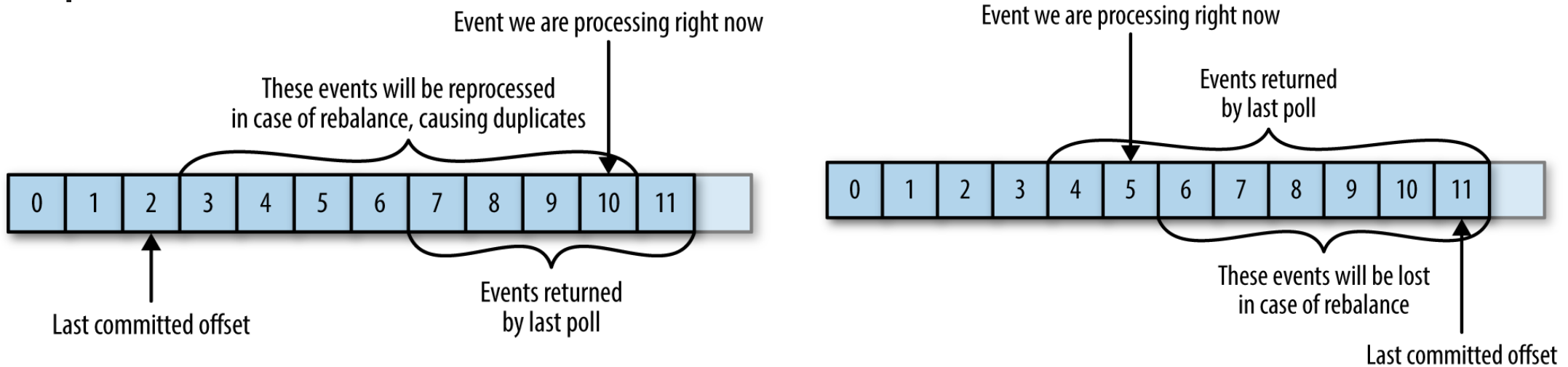
__consumer_offsets

- Le message 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



Kafla propose plusieurs façons de gérer les commits



Commit automatique

Si ***enable.auto.commit=true*** ,
le consommateur valide toutes les
auto.commit.interval.ms secondes (par
défaut 5), les plus grand offset reçus par poll()

Lors d'un appel à poll(), le consommateur vérifie
si il est temps de commit, si c'est le cas il
committe les offsets du précédent appel à
poll()

=> Cette approche (simple) ne protège pas
contre les duplications en cas de ré-affectation



Commit contrôlé

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

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

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



Commit Synchrone

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

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

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



Exemple

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



Commit asynchrone

La méthode `commitAsync()` est non bloquante

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



Example

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



Committer un offset spécifique

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

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



Example

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




Réagir aux réaffectations

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

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

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



Example

```
private class HandleRebalance implements
    ConsumerRebalanceListener {

    public void onPartitionsAssigned(
        Collection<TopicPartition> partitions) { }

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



Consommation de messages avec des offsets spécifiques

L'API permet différentes façon pour indiquer un offset spécifique :

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



Sortie de boucle

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

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

On peut utiliser

Runtime.addShutdownHook(Thread hook)



Exemple

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



Example (2)

```
try {
// looping until ctrl-c, the shutdown hook will cleanup on exit
while (true) {
    ConsumerRecords<String, String> records = movingAvg.consumer.poll(1000);
    log.info(System.currentTimeMillis() + "-- waiting for data...");
    for (ConsumerRecord<String, String> record : records) {
        log.info("offset = %d, key = %s,value = %s\n",
            record.offset(), record.key(),record.value());
    }
    for (TopicPartition tp: consumer.assignment())
        log.info("Committing offset atposition:"+consumer.position(tp));
    movingAvg.consumer.commitSync();
}
} catch (WakeupException e) {
} finally {
    consumer.close();
    log.info("Closed consumer and we are done");
}
```



Consommateur standalone

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()*



Example

```
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
Connect API



Introduction

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

Avec Kafka Connect, on peut ingérer des bases de données volumineuses, des données de surveillance avec des latences minimales



Fonctionnalités

Un cadre commun pour les connecteurs qui standardise l'intégration

Mode distribué ou standalone

Une interface REST permettant de gérer facilement les connecteurs

Gestion automatique des offset

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 (single process)

```
> 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

Les autres paramètres définissent les configuration des différents connecteurs



Mode distribué

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

Pour démarre 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 partition 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 (état de connexion par défaut) : topic utilisé pour stocker les états; Peut avoir plusieurs partitions et doit être répliquée et configurée pour le compactage



Configuration des connecteurs

En mode standalone, la configuration est défini dans un fichier `.properties`

En mode distribué, c'est via une API Rest et un format JSON que la configuration est mise à jour.

Les valeurs sont très dépendantes du type de connecteur.

Comme valeurs communes, on peut citer :

- ***name*** : Nom unique.
- ***connector.class*** : Classe Java du connecteur
- ***tasks.max*** : Maximum de tâches créées pour le connecteur. (degré de parallélisme)
- ***key.converter***, ***value.converter***

Les connecteurs de type *Sink* ont en plus :

- ***topics*** : Liste des topics servant d'entrée
- ***topics.regex*** : RegExp pour les topics d'entrée



Connecteurs

Confluent qui package une distribution de Kafka offre de base les connecteurs suivants :

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

De nombreux éditeurs fournissent 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

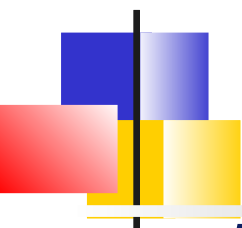
Une chaîne de transformation, s'appuyant sur des *transformer* prédéfinis, peut être spécifiée dans la configuration d'un connecteur.

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



Exemple de configuration

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```



Transformers disponibles

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

ReplaceField : Filtrer ou renommer des champs

MaskField : Remplace le champ avec une valeur nulle

ValueToKey : Échange clé/valeur

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

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 de destination via une regexp



API Rest

Le serveur API REST peut être configuré via la propriété listeners

`listeners=http://localhost:8080,https://localhost:8443`

Ces listener sont également utilisé par la communication intra-cluster

- Pour utiliser d'autres Ips pour cle 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



Kafka Streams

Introduction et Architecture
Création de topologie
Kafka DSL



Introduction

Depuis la version 0.10.0, Kafka fait plus que fournir une source fiable de flux de données.

Il inclut une dans son API une librairie de traitement de flux permettant aux développeurs de consommer, traiter et produire des événements dans leurs propres applications, sans s'appuyer sur un framework externe : La **stream API**



Apports de KafkaStream

Librairie simple et légère, pouvant être facilement intégrée dans n'importe quelle application Java .

Pas de dépendances externes sur des systèmes autres qu'Apache Kafka. Utilise notamment le modèle de partition pour le scaling et les garanties de livraison .

Permet un état local tolérant aux pannes permettant des opérations d'agrégation et de jointures très rapides.

Garantie que chaque évènement soit traité une seule fois, même en cas de défaillance.

Temps de latence des traitement en millisecondes,

Supporte des opérations de fenêtrage temporel avec l'arrivée des événements dans le désordre.

Offre les primitives de traitement de flux nécessaires sous forme de DSL Streams de haut niveau ou d'une API bas niveau.

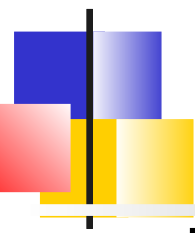


Définitions

Un ***data stream*** 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



Paradigme de programmation

Le traitement de flux est un paradigme de programmation, tout comme le traitement de requêtes ou le traitement par lot.

Il comble l'écart entre le monde de la requête-réponse où on attend des événements qui prennent quelques millisecondes à traiter et le monde du traitement par lots où les données sont traitées une fois par jour et prennent huit heures pour se terminer



Concepts Kafka

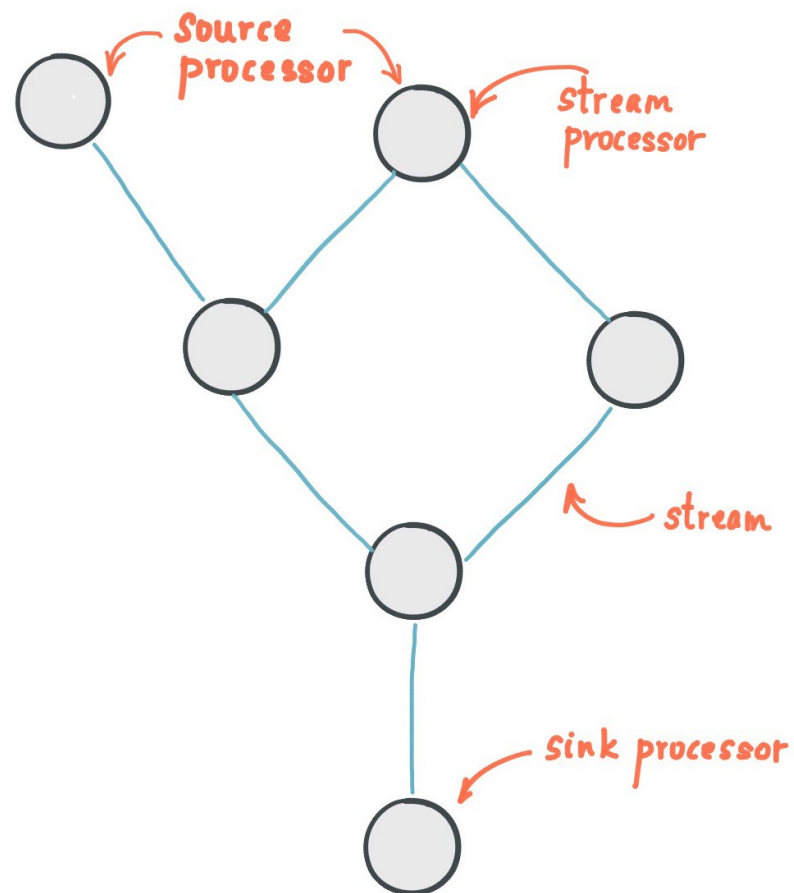
Une application de traitement de flux est un programme qui définit sa logique de calcul via une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

Certains processeurs :

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

Un processeur effectue une transformation sur ces données d'entrée et alimente soit un topic Kafka soit un système externe

Topologie processeurs



PROCESSOR TOPOLOGY



Temps

Différentes notions de date existent dans un Stream

- Date de l'événement : Le moment où l'événement s'est produit
- Date d'ingestion : Le moment où un événement est stocké dans une partition d'un topic.
- Date de traitement : Le moment où l'événement est traité par l'application de traitement de flux



Horodatage

Kafka Streams attribue un horodatage à chaque événement (via l'interface `TimestampExtractor`).

Ces horodatages décrivent la progression d'un flux par rapport au temps et sont exploités par des opérations dépendantes du temps telles que les extractions de fenêtre temporelle.

Ce temps n'avance que lorsqu'un nouvel événement arriver : ***stream time***

Chaque fois qu'une application Kafka Streams écrit des enregistrements dans Kafka, elle attribue également des horodatages à ces nouveaux enregistrements.



Flux et Table

En plus de l'abstraction *Stream*,
KafkaStream fournit l'abstraction
Table permettant les agrégations.

Il existe en fait une relation étroite entre
les flux et les tables : une dualité

=> Un stream peut être vu comme une
table, et une table peut être vue
comme un stream.



Agrégation, Jointure, fenêtrage

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'agréations ou des jointures.

- Possibilité d'indiquer une grace period : période pendant laquelle les événements arrivant en retard sont pris en compte



State store

Certaines applications de traitement de flux ne nécessitent pas d'état, i.e, le traitement d'un message est indépendant du traitement de tous les autres messages

D'autres nécessitent de conserver un état pour grouper, joindre, agréger

Kafka Streams fournit des **state stores** qui permettent aux applications de stocker et requêter les données

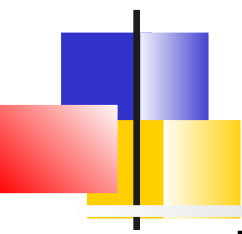
- KStream offre des fonctionnalités de tolérance aux pannes et une récupération automatique.
- Les state stores expose des interactive queries permettant un accès en lecture aux données ;



Garantie de traitement

2 principales garanties de traitement accessibles par simple configuration de KafkaStream

- ***At-least-one*** (défaut) : Tout événement est traité au moins 1 fois. Tolérance vis à vis des doublons de traitement
- ***Exactly-once*** : S'appuie sur les capacités de transactions et d'idempotence de Kafka



Événement non ordonnés

Kafka Stream



Partition de flux

Kafka Streams utilise les concepts de **partitions de flux** basé sur les partitions des topics Kafka.

- Chaque partition de flux est une séquence totalement ordonnée d'événement et correspond à une partition d'un topic Kafka.
- Un événement dans le flux correspond à un message Kafka du topic.
- Les clés des événements déterminent le partitionnement des données dans les flux et comment les données sont acheminées vers des partitions spécifiques dans les topics.



Tâches

La scalabilité d'une topologie de traitement de flux est assuré par les **tâches**

- *Kafka Streams* crée un nombre fixe de tâches en fonction des partitions de flux d'entrée pour l'application, chaque tâche se voit attribuer une liste de partitions à partir des flux d'entrée.
- L'affectation de partitions aux tâches ne change jamais, chaque tâche est une unité fixe de parallélisme de l'application
- Les tâches conservent un buffer un tampon pour chacune des partitions qui leur sont affectées et traitent les messages un par un à partir de ces buffers



Modèle concurrentiel

Kafka Streams permet de configurer le nombre de threads utilisé pour le parallélisme.

- Chaque thread peut exécuter 1 ou plusieurs tâches.
- Démarrer de nouvelles threads ou de nouvelles instances de l'application revient simplement à répliquer la topologie de traitement et à la faire traiter un sous-ensemble différent de partitions Kafka augmentant le parallélisme global
- L'affectation des partitions de topics Kafka parmi les différents threads est gérée de manière transparente par Kafka Streams.

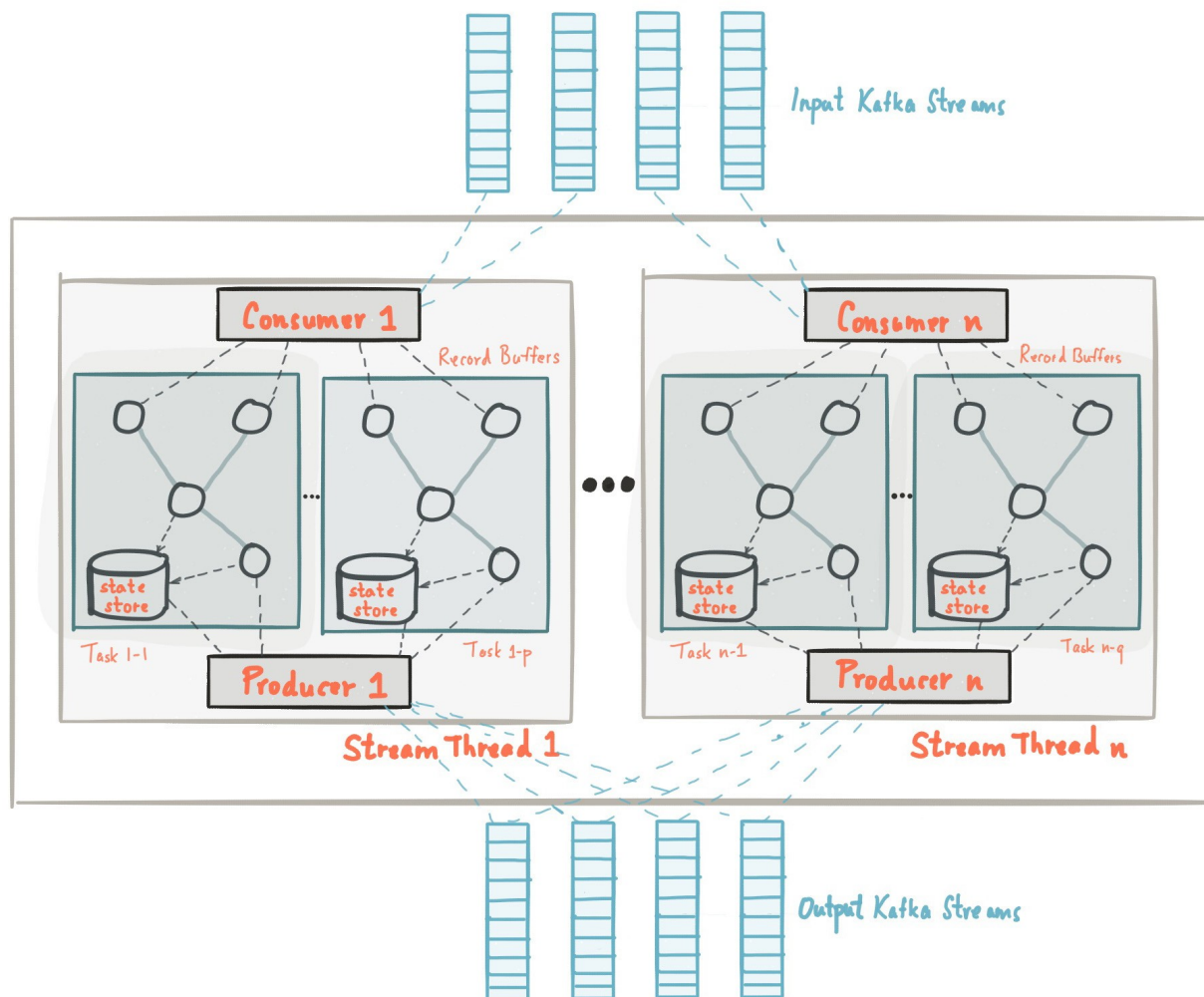


Local state store

Chaque tâche de flux peut gérer 1 ou plusieurs ***local state store***

- Ces stores sont accessibles via des API pour stocker et interroger les données nécessaires au traitement.
- Kafka Streams offre une tolérance aux pannes et une récupération automatique de ces local state store.

Architecture





Kafka Streams

Introduction et Architecture
Création de topologie
Kafka DSL



Introduction

La logique de traitement d'une application Kafka Streams est définie par une topologie de processeur, i.e un graphe de processeurs de flux

La topologie peut être définie via

- **Kafka DSL** : Une API de haut niveau qui fournit les opérations de transformation de données les plus courantes telles que *map*, *filter*, *join* prêtes à l'emploi.

Le DSL couvre de nombreux cas d'usage et est généralement suffisant

- **L'API Processor** qui permet d'ajouter ces processeurs et d'interagir avec les *state store*



Dépendances Maven

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>2.4.1</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.4.1</version>
</dependency>
<!-- Optionnellement inclure Kafka Streams DSL pour Scala 2.12 -->
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams-scala_2.12</artifactId>
  <version>2.4.1</version>
</dependency>
```



Application *KafkaStream*

Généralement, les appels à Kafka Streams sont appelés dans la méthode *main ()*.

Il faut d'abord définir la topologie de traitement, en créant une instance de *KafkaStreams*. Le constructeur prend 2 arguments :

- Une topologie
(*StreamsBuilder#build()* pour le DSL, *Topology* pour l'API Processor API)
- Une instance de *java.util.Properties*, définissant la configuration.

Ensuite, il faut démarrer les threads associées à cette topologie via la méthode *start()*

Si d'autres applications sont démarrées, Kafka réassigne automatiquement les tâches existantes aux nouvelles tâches démarrées



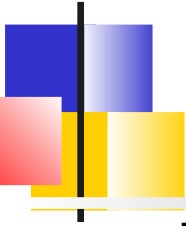
Exemple

```
StreamsBuilder builder = ...; // lors de l'utilisation de DSL
Topology topology = builder.build();
// OU
Topology topology = ...; // en utilisant l'API Processor

// Configuration : Emplacement du cluster
// sérialiseurs/désérialiseurs
// sécurité ...
Properties props = ...;

KafkaStreams streams = new KafkaStreams(topology, props);

// Démarrage des threads
streams.start();
```



Traitement des exceptions

Pour traiter les exceptions, il faut positionner un ***java.lang.Thread.UncaughtExceptionHandler*** avant de démarrer l'application.

Ex :

```
streams.setUncaughtExceptionHandler((Thread thread, Throwable throwable) -> {  
    // Traitement de l'exception  
});
```



Arrêt et ShutdownHook

Pour arrêter une instance et stopper les threads :

```
streams.close();
```

Généralement, on positionne un ShutdownHook qui arrête le stream :

```
Runtime.getRuntime().addShutdownHook(  
    new Thread(streams::close));
```




Configuration

2 propriétés sont requises pour la configuration :

- ***application.id*** : Chaque application doit avoir un identifiant unique pour le cluster Kafka.
Toutes les instances d'une même application ont le même identifiant
Il peut contenir la version de l'application
L'identifiant est utilisé comme préfixe pour les propriétés :
 - *client.id* des producteurs et consommateurs Kafka
 - *group.id* des consommateurs
 - Nom du sous-répertoire pour les local store
 - Les noms des topics internes Kafka
- ***bootstrap.servers*** : Liste de brokers pour déterminer le cluster



Autres propriétés de configuration

replication.factor (1) : Facteur de réplication des topics internes de *KafkaStream*. Valeur recommandée égale à celle des topics source

state.dir (/tmp/kafka-streams): Emplacement des state store

key.serde* / *value.serde (*Serdes.ByteArray()*):
Sérialiseur/Désérialiseur par défaut des clés/valeurs

num.stream.threads (1) : Nombre de threads exécutant les traitements

num.standby.replicas (0) : Nombre de copies des local store



Autres propriétés de configuration (2)

processing.guarantee (at-least-one) : Garanti de traitement

retries (0) : Nombre de réessai lors d'une exception
Retryable

timestamp.extractor (*FailOnInvalidTimestamp*): Classe qui extrait le timestamp du message

default.production.exception.handler : Gestionnaire d'exception lors de l'envoi de message

default.deserialization.exception.handler :
Gestionnaire d'exception lors de la désérialisation

...



Kafka Streams

Introduction et Architecture
Création de topologie
Kafka DSL



Introduction

DSL propose :

- Des abstractions pour les flux et les tables sous la forme de KStream, KTable et GlobalKTable
- Un modèle de programmation déclaratif et fonctionnel avec des transformations stateless (map, filter) ou stateful (agregation, join, ...)



DSL et topologies

La définition d'une topologie avec DSL s'effectue en :

- Spécifiant un ou plusieurs flux d'entrée à partir de topics Kafka
- Composant des transformations sur ces flux
- Écrivant les flux de sortie résultants dans les rubriques Kafka ou en exposant les résultats via des requêtes interactives (par exemple, via une API REST).



KStream vs KTable

Un ***KStream*** est une abstraction d'un flux d'enregistrement, où chaque enregistrement représente une donnée autonome dans l'ensemble illimité.

Un enregistrement est interprété comme un "INSERT" ou "APPEND"

`("alice", 1) --> ("alice", 3) => alice=4`

Une ***KTable*** est une abstraction d'un flux de journal des modifications, où chaque enregistrement de données représente une mise à jour.

Un enregistrement est interprété comme un "UPDATE"

`("alice", 1) --> ("alice", 3) => alice=3`



GlobalKTable vs KTable

Une *KTable* est renseignée avec les données d'une partition Kafka, une ***GlobalKtabke*** est renseignée avec les données de tout un topic

- *GlobalKTable* permet de rechercher les valeurs courantes des enregistrements par clés facilitant les jointures
- *GlobalKTable* n'a aucune notion de temps contrairement à un *KTable*.



Spécification des flux d'entrée

Les topics Kafka d'entrée peuvent être transformés en *KStream* ou *Ktable*.

Par exemple pour *KStream* :

```
StreamsBuilder builder = new StreamsBuilder();
```

```
KStream<String, Long> wordCounts = builder.stream(  
    "word-counts-input-topic", /* topic d'entrée */  
    Consumed.with(  
        Serdes.String(), /* key serde */  
        Serdes.Long()    /* value serde */  
    );
```



Spécification des flux d'entrée - GlobalKTable

Une *KTable* doit être nommée afin de permettre des requêtes interactives. Les données d'une table sont stockées dans un *store*

Par exemple pour une *GlobalKTable* :

```
StreamsBuilder builder = new StreamsBuilder();

GlobalKTable<String, Long> wordCounts = builder.globalTable(
    "word-counts-input-topic",
    Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as(
        "word-counts-global-store" /* nom de la table/store */
        .withKeySerde(Serdes.String()) /* key serde */
        .withValueSerde(Serdes.Long()) /* value serde */
    );
```



Transformations

Les interfaces KStream et KTable supportent de nombreuses opérations de transformations génériques

- Certaines transformations génère 1 ou plusieurs Kstream. (Ex : map, branch)
- D'autres génèrent des Ktable (Ex : agrégation)
- Les transformations sur les tables ne peuvent générer que des KTable
- Kafka permet de convertir un Ktable en Kstream

Les transformations peuvent être classifiées en *stateless* et *stateful*



Transformations stateless

map ($KStream \rightarrow KStream$) : Prend un enregistrement et produit un autre enregistrement. La clé et la valeur peuvent être modifiées

filter ($KStream \rightarrow KStream, KTable \rightarrow KTable$) : Retient certains enregistrements à partir d'une condition booléenne

FlatMap ($KStream \rightarrow KStream$) : Prend un enregistrement et produit zéro, un ou plusieurs enregistrements

forEach ($KStream \rightarrow void, KTable \rightarrow void$) : Opération terminale sur chaque enregistrement

peek ($KStream \rightarrow KStream$) : Effectue une opération stateless sur chaque enregistrement



Transformations stateless (2)

selectKey (KStream → KStream) : Assigne une nouvelle clé

groupBy (KStream → KGroupedStream, KTable → KGroupedTable) :
Pré-requis à une agrégation, groupe les enregistrements par une nouvelle clé

branch (KStream → KStream[]) : Divise le flux d'entrée en plusieurs flux selon un prédicat

Merge (KStream → KStream) : Fusionne un flux avec un flux passé en paramètre. Aucune garantie d'ordre

TableToStream (KTable → KStream) : Transforme une *KTable* en *KStream*

print (KStream → void) : Opération terminale, Affiche chaque enregistrement



Examples

```
// map
KStream<String, Integer> transformed = stream.map(
    (key, value) -> KeyValue.pair(value.toLowerCase(), value.length()));
// Filter
KStream<String, Long> onlyPositives = stream.filter((key, value) -> value > 0);
// branch
KStream<String, Long> stream = ...;
KStream<String, Long>[] branches = stream.branch(
    (key, value) -> key.startsWith("A"), /* first predicate */
    (key, value) -> key.startsWith("B"), /* second predicate */
    (key, value) -> true                 /* third predicate */
);
// merge
KStream<byte[], String> merged = stream1.merge(stream2);
// foreach
stream.foreach((key, value) -> System.out.println(key + " => " + value));
// peek
KStream<byte[], String> unmodifiedStream = stream.peek(
    (key, value) -> System.out.println("key=" + key + ", value=" + value));
```



Transformations stateful

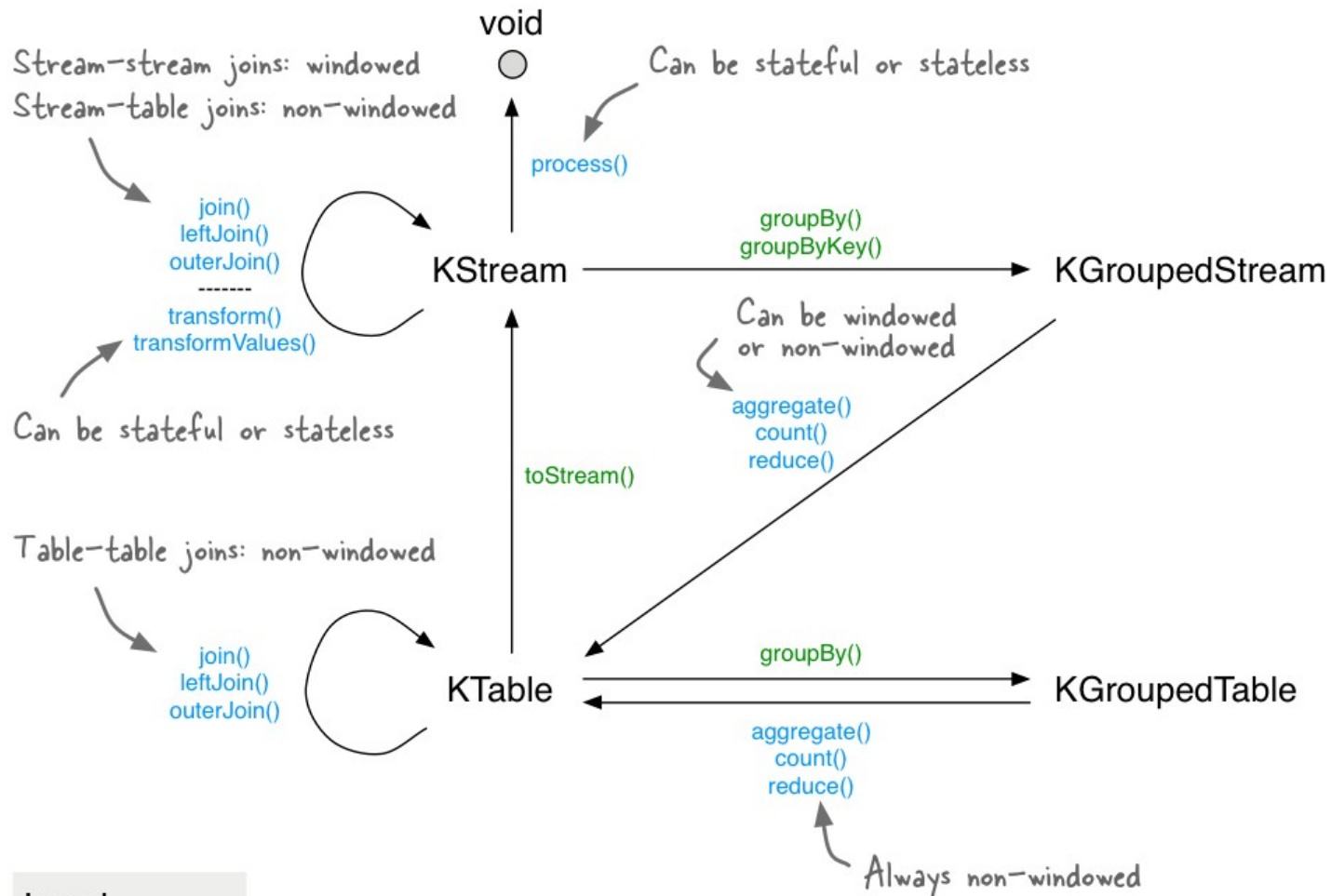
Les transformations stateful dépendent d'un état stocké dans un store (fault-tolerant)

Par exemple, pour une agrégation, un store est utilisé pour collecter les derniers résultats d'agrégation pour une fenêtre temporelle donnée.

Les transformations disponibles avec DSL sont :

- Les agrégations
- Les jointures
- Le fenêtrage
- Une intégration de processeurs spécialisés définis avec la Processor API

Résumé



Legend

Stateful operations
Stateless operations

GlobalKTable
no direct operations



Exemple

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

```
KStream<String, Long> wordCounts = textLines
    // Split chaque ligne de texte en mots.
    // La ligne étant stockée dans la valeur
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // Grouper le flux par mot. La clé est le mot
    .groupBy((key, word) -> word)
    // Compter les occurrences de chaque mot .
    // `KGroupedStream<String, String>`
    // => `KTable<String, Long>` (word -> count).
    .count()
    // Convertir la `KTable<String, Long>` en un `KStream<String, Long>`.
    .toStream();
```



Agrégations

Quand les enregistrements ont été regroupés par clé via `groupByKey` ou `groupByKey` - et donc représentés comme `KGroupedStream` ou `KGroupedTable`, ils peuvent être agrégés via une opération telle que *reduce*

Les agrégations sont des opérations basées sur les clés, i.e. elles s'appliquent sur les enregistrements ayant la même valeur de clé fin d'un job.

- En mode mouvant, elles ont comme signature :
KGroupedStream → *KTable*, *KGroupedTable* → *Ktable*
- En mode fenêtrée, elles ne sont disponible que pour :
KGroupedStream → *KTable*



Transformations d'agrégations

count : Comptage par clé

reduce : La valeur d'enregistrement courant est combinée avec la dernière valeur réduite et une nouvelle valeur réduite est renvoyée.

- Pour un *KGroupedStream*, on doit fournir un **adder**
- Pour un *KGroupedTable*, il faut fournir un **subtractor**, méthode appelée pour les valeurs nulles

aggregate : Généralisation de *reduce* permet d'agréger des valeurs de différents types que le résultat.

- Pour un *KGroupedStream*, on doit fournir un **initializer** et un **adder**
- Pour un *KGroupedTable*, il faut fournir un **subtractor**
- Lors d'un fenêtrage basée sur les sessions, il faut fournir un **sessionMerger**, méthode appelée lorsque 2 sessions sont fusionnées



Exemples

// Compter un KGroupedStream

```
KTable<String, Long> aggregatedStream = groupedStream.count();
```

// Compter un KGroupedStream avec une fenêtre de temps de 5mn

```
KTable<Windowed<String>, Long> aggregatedStream =  
    groupedStream.windowedBy(  
        TimeWindows.of(Duration.ofMinutes(5))) /* time-based window */  
        .count();
```

// Reduire a KGroupedStream

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

// Aggregation avec une fenêtre basé sur la session (timeout 5 minutes)

```
KTable<Windowed<String>, Long> sessionizedAggregatedStream =  
    groupedStream.windowedBy(  
        SessionWindows.with(Duration.ofMinutes(5))) /* session window */  
        .reduce(  
            (aggValue, newValue) -> aggValue + newValue /* adder */  
        );
```



Exemples (2)

```
// Aggregation pour un KGroupedStream (Passage de String à Long)
KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
    () -> 0L, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /*
    adder */
    Materialized.as("aggregated-stream-store") /* nom du state store */
    .withValueSerde(Serdes.Long()); /* serde pour valeur agrégée */
```

```
// Aggregating a KGroupedTable (note how the value type changes from
    String to Long)
KTable<byte[], Long> aggregatedTable = groupedTable.aggregate(
    () -> 0L, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */
    (aggKey, oldValue, aggValue) -> aggValue - oldValue.length(), /* subtractor */
    Materialized.as("aggregated-table-store") /* nom du state store */
    .withValueSerde(Serdes.Long()) /* serde pour valeur agrégée */
```



Exemples (3)

// Agrégation sur une fenêtre de temps de 5 mn

```
KTable<Windowed<String>, Long> timeWindowedAggregatedStream =  
  groupedStream.windowedBy(Duration.ofMinutes(5))  
    .aggregate(  
      () -> 0L, /* initializer */  
      (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */  
      Materialized.<String, Long, WindowStore<Bytes, byte[]>>as("time-windowed-  
aggregated-stream-store")  
      .withValueSerde(Serdes.Long()));
```

// Agrégation basée sur une session de 5 minutes d'inactivité

```
KTable<Windowed<String>, Long> sessionizedAggregatedStream =  
  groupedStream.windowedBy(SessionWindows.with(Duration.ofMinutes(5)).  
    aggregate(  
      () -> 0L, /* initializer */  
      (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */  
      (aggKey, leftAggV, rightAggV) -> leftAggV + rightAggV, /* session merger */  
      Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("sessionized-  
aggregated-stream-store")  
      .withValueSerde(Serdes.Long()));
```



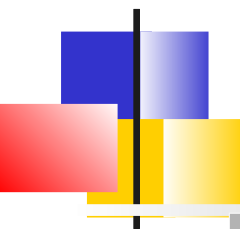
Jointures, introduction

Les KStreams et les KTables peuvent également être joints.

Dans la pratique, de nombreuses applications traitant un événement doivent effectuer des recherches supplémentaires pour compléter les données sur l'événement.

Un pattern classique consiste à rendre les informations des BD relationnelles disponibles dans Kafka, puis exploiter l'API Streams pour effectuer des jointures locales extrêmement rapides et efficaces sur ces tables.

=> Les KTable permettent de suivre le dernier état d'une table BD dans un local store, réduisant ainsi considérablement la latence de traitement et la charge BD.



Opérandes de jointure	Type	Inner	LEFT	OUTER
KStream vers KStream	Fenêtré	Oui	Oui	Oui
KTable vers KTable	Non-fenêtré	Oui	Oui	Oui
KTable vers KTable Clé étrangère	Non-fenêtré	Oui	Oui	Non
KStream vers KTable	Non-fenêtré	Oui	Oui	Non
KStream vers GlobalTable	Non-fenêtré	Oui	Oui	Non
KTable vers GlobalKTable	Pas applicable	Non	Non	Non



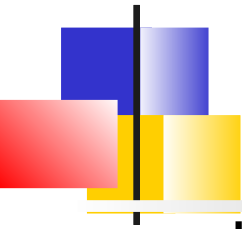
Contraintes sur le partitionnement

Pour effectuer une jointure sur la clé, i.e `leftRecord.key == rightRecord.key`, il faut s'assurer que les enregistrements joints soient traités par la même tâche :

- Les topics d'entrée doivent avoir le même nombre de partitions
- La stratégie de partitionnement des producteurs des 2 topics doivent être identiques

2 exceptions : Les jointures `kStream` vers `KglobalKTable` et `Ktable` vers `Ktable` avec une clé étrangère

Kafka vérifie la contrainte sur le nombre de partitions au moment de la création de la topologie et génère une `TopologyBuilderException` si ce n'est pas le cas.



Kstream vers KStream

Les jointures KStream-KStream sont toujours des jointures fenêtrées, car sinon la taille du store interne augmenterait indéfiniment.

Par contre le flux de sortie n'est pas fenêtré

Un enregistrement d'un côté peut produire plusieurs sorties de jointure (1 pour les chaque enregistrement qui match de l'autre côté)

Les enregistrements de sortie sont créés à partir d'un ***ValueJoiner*** fourni par l'utilisateur

Les enregistrements avec une clé ou une valeur nulle ne provoque pas jointure



Exemple

// Inner-join

```
KStream<String, String> joined = left.join(right,  
    (lValue, rValue) -> "left=" + lValue + ", right=" + rValue, /* ValueJoiner */  
    JoinWindows.of(Duration.ofMinutes(5)),  
    Joined.with(  
        Serdes.String(), /* key */  
        Serdes.Long(),   /* left value */  
        Serdes.Double()) /* right value */  
    );
```

// Left-join, pour les enregistrements à gauche qui n'ont pas de correspondant à droite,

// le ValueJoiner est appelé avec (leftRecord.value, null)

```
KStream<String, String> joined = left.leftJoin(right,  
    (lValue, rtValue) -> "left=" + lValue + ", right=" + rValue, /* ValueJoiner */  
    JoinWindows.of(Duration.ofMinutes(5)),  
    Joined.with(  
        Serdes.String(), /* key */  
        Serdes.Long(),   /* left value */  
        Serdes.Double()) /* right value */  
    );
```



Exemple (2)

```
// Outer-join pour les enregistrements à droite
// qui n'ont pas de correspondant à gauche,
// le ValueJoiner est appelé avec (null, rightRecord.value)
KStream<String, String> joined = left.outerJoin(right,
    (lValue, rValue) ->
    "left=" + lValue + ", right=" + rValue, /* ValueJoiner */
    JoinWindows.of(Duration.ofMinutes(5)),
    Joined.with(
        Serdes.String(), /* key */
        Serdes.Long(),   /* left value */
        Serdes.Double()) /* right value */
    );
```



Ktable vers KTable

Les jointures KTable-KTable sont toujours des jointures non fenêtrés, elles sont similaires avec les jointures des bases de données relationnelles.

Le résultat de la jointure est une nouvelle *KTable* qui représente le flux du journal des modifications des 2 *KTables*.

Les enregistrements de sortie sont créés à partir d'un ***ValueJoiner***

- Les enregistrements ayant une clé nulle ne sont pas traités
- Lors d'inner join, les enregistrements ayant une valeur nulle ont pour conséquence de supprimer l'enregistrement dans la *Ktable* résultante



Example

// Inner join

```
KTable<String, String> joined = left.join(right,  
    (lValue, rValue) ->  
        "left=" + lValue + ", right=" + rValue /* ValueJoiner */  
    );
```

// Left join

```
KTable<String, String> joined = left.leftJoin(right,  
    (lValue, rValue) ->  
        "left=" + lValue + ", right=" + rValue /* ValueJoiner */  
    );
```

// Outer join

```
KTable<String, String> joined = left.outerJoin(right,  
    (lValue, rValue) ->  
        "left=" + lValue + ", right=" + rValue /* ValueJoiner */  
    );
```



KTable vers KTable avec clé étrangère

Les jointures de clé étrangères Ktable vers sont toujours non fenêtré.

Elles sont analogues aux jointures en SQL.

Elles produisent une KTable en sortie.

Une **fonction d'extraction** de la clé-étrangère est appliquée à l'enregistrement de gauche

- Si elle retourne null, la jointure n'est pas déclenchée

La table de gauche peut avoir plusieurs enregistrements avec la même clé étrangère

Les topics ne sont pas obligatoirement co-partitionnés



Exemple

```
// Extraction utilisant la valeur gauche à faire correspondre  
// avec la clé à droite.
```

```
Function<Long, Long> foreignKeyExtractor = (x) -> x;
```

```
// Inner join
```

```
KTable<String, String> joined = left.join(right, foreignKeyExtractor,  
    (leftValue, rightValue) ->  
    "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */  
    );
```

```
// Left Join
```

```
KTable<String, String> joined = left.join(right, foreignKeyExtractor,  
    (leftValue, rightValue) ->  
    "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */  
    );
```




KStream vers KTable

Les jointures *KStream-KTable* sont toujours des jointures non fenêtrées.

Elles permettent d'effectuer des recherches sur un *KTable* (flux de journal des modifications) lors de la réception d'un nouvel enregistrement du *KStream*.

Les enregistrements de sortie sont créés comme à partir du ***ValueJoiner***

- Les entrées du *KStream* avec une clé ou valeur nulle ne déclenchent pas la jointure
- Les entrées de la *KTable* avec une clé nulle provoque la suppression dans la *KTable*



Example

```
// Inner join
```

```
KStream<String, String> joined = left.join(right,  
    (leftValue, rightValue) ->  
    "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */  
    Joined.keySerde(Serdes.String()) /* key */  
    .withValueSerde(Serdes.Long()) /* left value */  
    );
```

```
// Left join
```

```
KStream<String, String> joined = left.leftJoin(right,  
    (leftValue, rightValue) ->  
    "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */  
    Joined.keySerde(Serdes.String()) /* key */  
    .withValueSerde(Serdes.Long()) /* left value */  
    );
```



Kstream vers GlobalKTable

Les jointures KStream-GlobalKTable sont très similaires aux jointures KStream-KTable.

Elles offrent cependant plus de flexibilité :

- Pas besoin d'être co-partitionnés
- Idéal pour les tables de références
- Elles permettent des jointures par clés étrangères
- Plus efficaces les KTable lorsque l'on doit devez effectuer plusieurs jointures successives



Exemple

```
// Inner Join
```

```
KStream<String, String> joined = left.join(right,  
    /* détermine la clé étrangère à gauche */  
    (leftKey, leftValue) -> leftKey.length(),  
    /* ValueJoiner */  
    (leftValue, rightValue) ->  
        "left=" + leftValue + ", right=" + rightValue  
);
```

```
// Left join
```

```
KStream<String, String> joined = left.leftJoin(right,  
    /* détermine la clé étrangère à gauche */  
    (leftKey, leftValue) -> leftKey.length(),  
    /* ValueJoiner */  
    (leftValue, rightValue) ->  
        "left=" + leftValue + ", right=" + rightValue  
);
```



Retour vers un topic

Les Kstream et Ktables peuvent être réécrits dans un topic Kafka.

Les données de sortie peuvent alors être re-partitionnées.

2 méthodes sont fournies :

- To : Opération terminale qui enregistre l'enregistrement dans un topic
- Through : Qui enregistre dans un topic utilisé pour recréer un KStream ou KTable



to

Éventuellement, fournir des
séréaliseurs/déséréaliseurs pour les clés/valeurs

Éventuellement, fournir un *StreamPartitionner* pour
contrôler la répartition des partitions

Éventuellement, utiliser un *TopicExtractor* pour
router vers des topics différents

```
// Stream
```

```
stream.to("my-stream-output-topic");
```

```
// Table
```

```
table.to("my-table-output-topic");
```

```
// En spécifiant des séréaliseurs.
```

```
stream.to("my-stream-output-topic", Produced.with(Serdes.String(),  
    Serdes.Long()));
```



Examples : through

// Stream

```
KStream<String, Long> newStream =  
    stream.through("user-clicks-topic").map(...);
```

// `Table

```
KTable<String, Long> newTable = table.through("my-  
table-output-topic").map(...)
```