





Kafka Streams & ksqlDB

David THIBAU – 2024

david.thibau@gmail.com



Agenda

Rappels Kafka

- Kafka, ses cas d'usage
- Concepts
- Kafka APIs

KafkaStream

- Concepts
- Démarrage
- Configuration
- Opérateurs stateless
- Opérateurs Stateful
- Requêtes interactives

kSqlDB

- Démarrage
- SQL
- Statements
- Fonctions



Rappels Kafka

Kafka et ses cas d'usage

Concepts
Kafka APIs



Projet Kafka

Le projet Kafka initié par LinkedIn est désormais un projet **Apache** sous licence OpenSource

La société **Confluent** Créé en 2014 par *Jay Kreps, Neha Narkhede*, et *Jun Rao* est le *mainteneur* principal d'Apache Kafka

Elle fournit également *Confluent* :

- Des distributions commerciales de Kafka
- Des services Cloud
- Des librairies ou produits connexes : Clients dans d'autres langages que Java, Sérialiseurs, SchemaRegistry
- De la documentation



Fonctionnalités

Kafka a trois capacités clés:

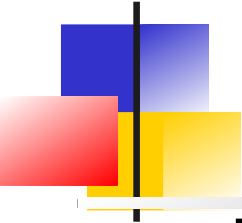
- Publier et s'abonner à des flux de messages¹ avec certaines garanties de fiabilité.
- Stocker les flux de messages de manière durable et tolérante aux pannes.
- Traiter, transformer les flux de messages au fur et à mesure qu'ils se produisent.

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



Garanties Kafka

- En fonction de sa configuration Kafka peut garantir 3 niveaux de livraison de messages malgré des défaillances :
 - **At Most Once** : Chaque message est délivré au plus une fois. Pas de doublons mais des pertes de messages. Débit le plus élevé
 - **At Least Once** : Chaque message est délivré au moins une fois. Pas de perte mais des doublons (sans incidence si le traitement est idempotent).
 - **Exactly Once** : Chaque message est délivré une et une seule fois. Le traitement consiste à produire un message Kafka vers un topic.



Cas d'usages

Les principaux cas d'usage sont :

- **Message Broker** permettant de découpler le cycle de vie des producteurs et des consommateurs de messages :
=> Support de communication entre micro-services, ESB
- **Buffering d'évènements** : Permet de bufferiser les évènements avant des traitements batch ou temps-réels :
Ex : Ingestion massive d'évènements dans pile ELK
- Backbone pour des **architectures Event-Driven**



Architecture Event-driven

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

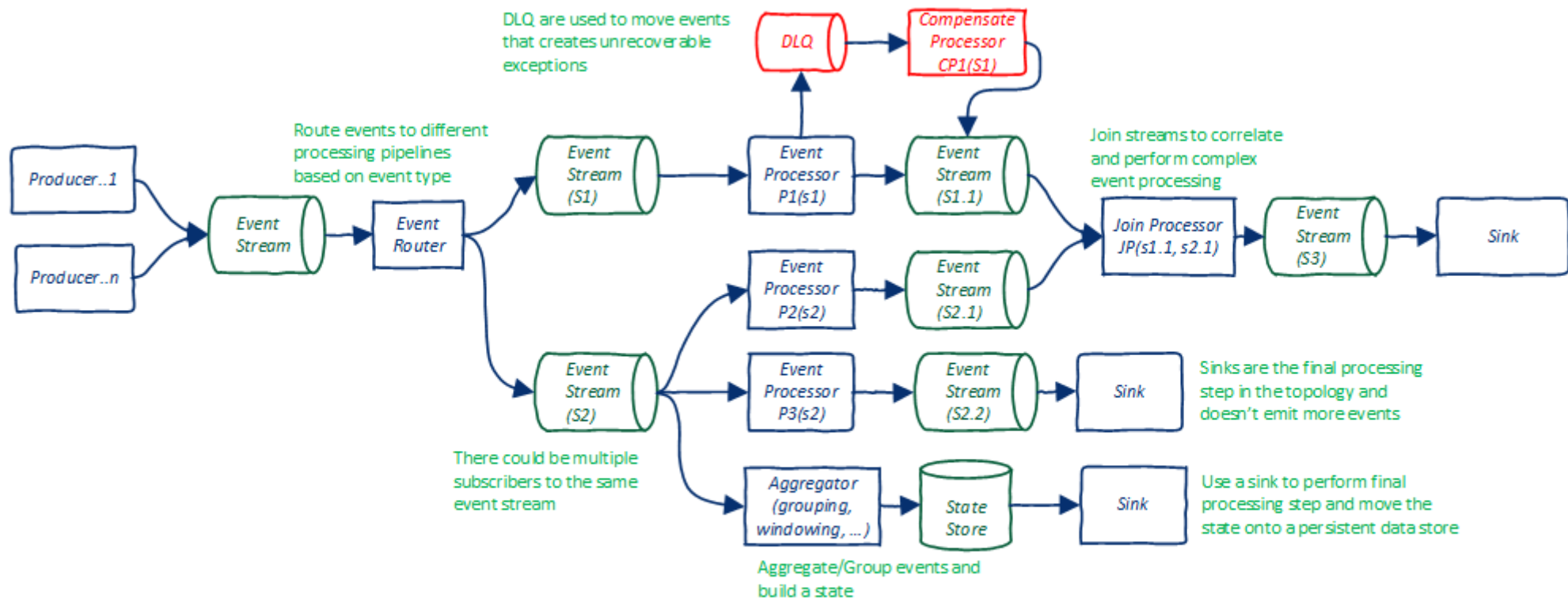
- Produit généralement des architectures plus souples et plus réactives.

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 facilite 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

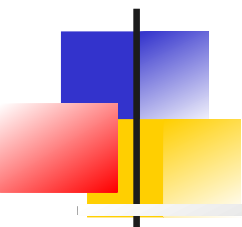


Rappels Kafka

Kafka et ses cas d'usage

Concepts

Kafka APIs



Cluster

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

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

1. Avant la version 3.x, un cluster Kafka nécessitait également un ensemble Zookeeper qui jouait le rôle de contrôleur



Topics et records

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

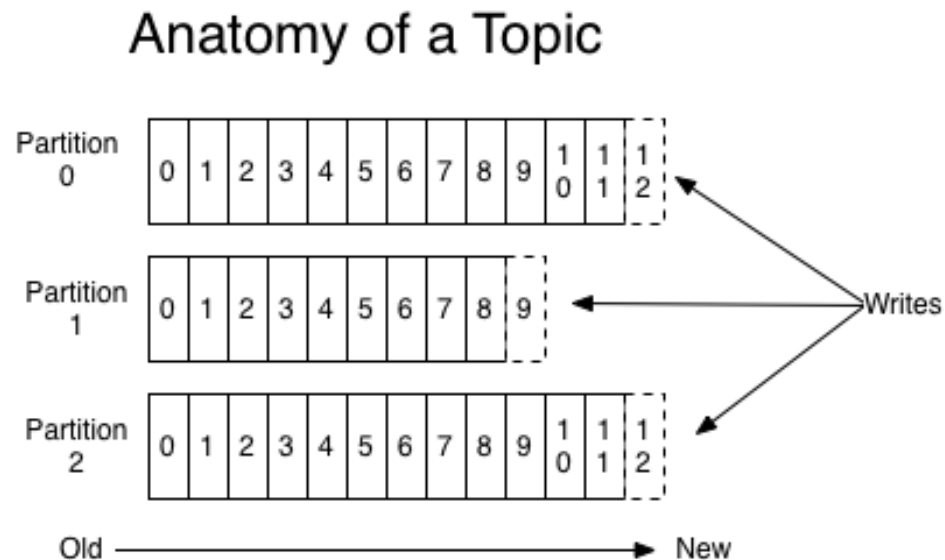
Chaque enregistrement se compose d'une
clé éventuelle, d'entêtes éventuelles,
d'une valeur et d'un horodatage.



Partitionnement des topics

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

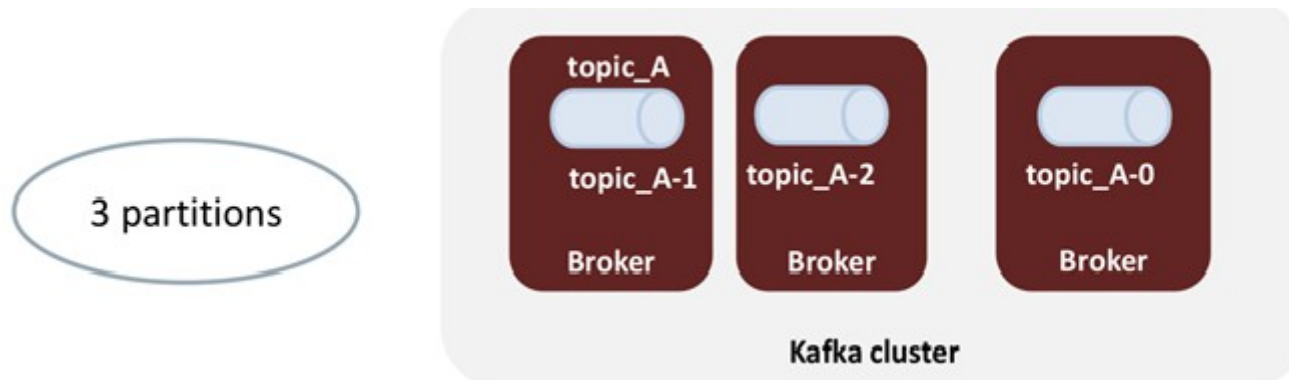
Ils peuvent être ***partitionnés***.





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*

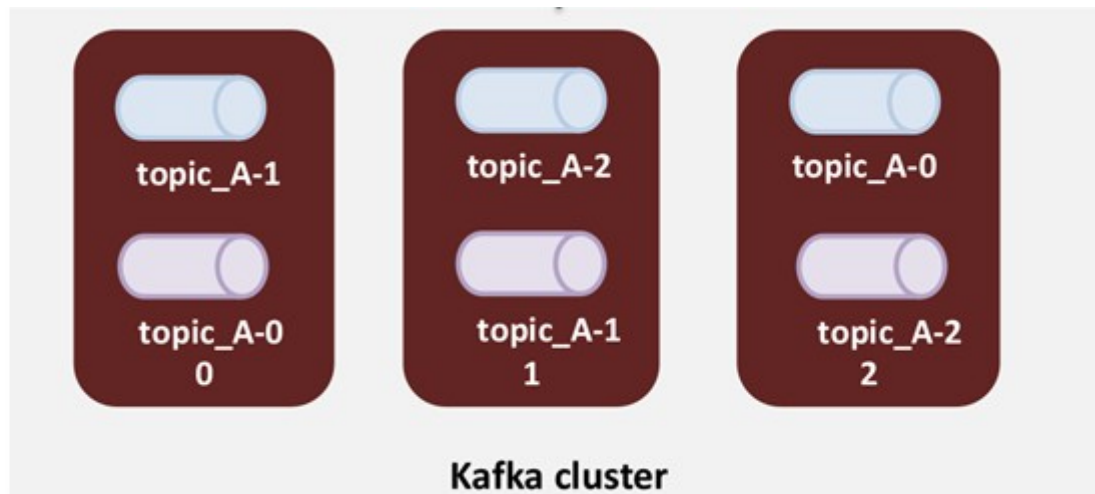




Réplication

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 défaille, un processus d'élection choisit un autre maître parmi les répliques



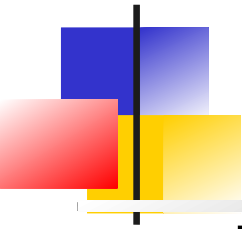
Partition et offset

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.

Les vieux messages sont supprimés en utilisant une **période de rétention** configurable ainsi qu'une stratégie d'archivage (**delete** ou **compact**).

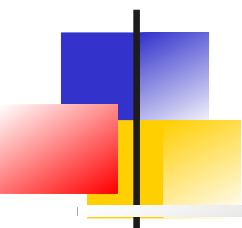


Routing des messages

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 par Kafka est son **offset** du journal¹.

Cet offset est contrôlé par le consommateur:

- Le consommateur signale à Kafka son avancée dans sa lecture des enregistrements.
Plusieurs techniques sont possibles et ont des incidences sur les garanties de livraison. (Auto-commit, commit manuel, transactions)
- Mais, il peut consommer dans l'ordre qu'il souhaite.
Par exemple, retraiter les données les plus anciennes ou repartir d'un offset particulier.

1. Information stockée dans un topic interne nommé `__consumer_offsets`



Rappels Kafka

Kafka et ses cas d'usage

Concepts

Kafka APIs



Applications clients

La communication entre les clients et les brokers Kafka 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.

Les clients sont configurés avec la propriété ***bootstrap-servers*** qui liste un sous-ensemble des Ips du cluster.

Après qu'un client accède à une de ses Ips, il découvre l'intégralité des brokers du cluster

Apache fournit les librairies clientes Java, Confluent les fournit dans d'autres langages



APIs

5 principales APIs sont disponibles :

- L'API **Producer** permet de publier un flux
- L'API **Consumer** permet à une application de s'abonner à un ou plusieurs topics.
- 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

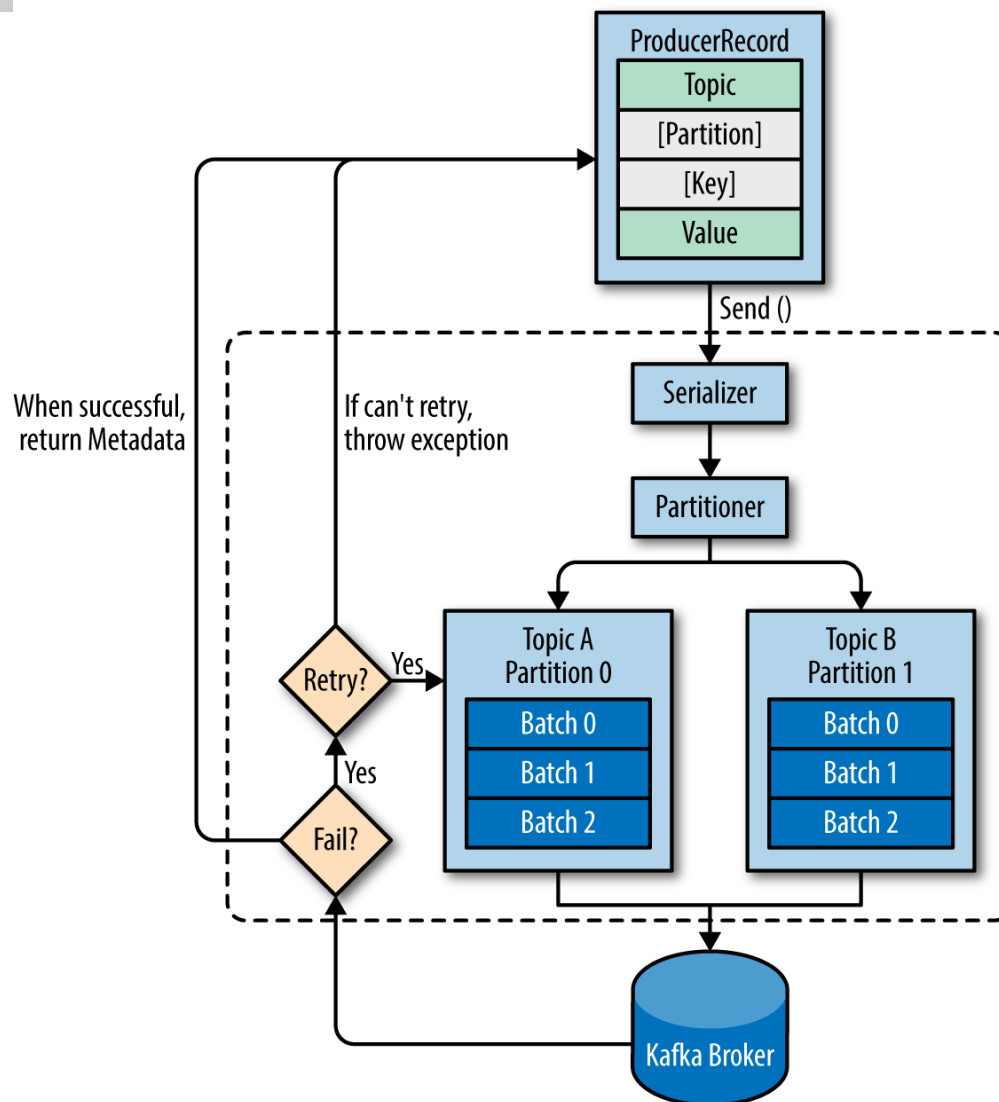


Étapes lors de l'envoi d'un message

L'envoi de message est constitué de plusieurs étapes :

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

Envoi de message



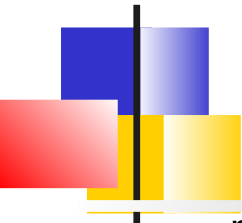


Construire un Producteur

La première étape pour l'envoi consiste à instancier un ***KafkaProducer*** en lui passant des propriétés de configuration

3 propriétés de configurations sont obligatoires :

- ***bootstrap.servers*** : Liste de brokers que le producteur contacte au départ pour découvrir le cluster
- ***key.serializer*** : La classe utilisée pour la sérialisation de la clé
- ***value.serializer*** : La classe utilisée pour la sérialisation du message ...
- Pour la sémantique Exactly Once, les propriétés ***enable.idempotence*** et ***transactional.id*** sont généralement positionnées



Exemple Java : Envoi asynchrone avec call-back

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

producer.send(record, new Callback() {
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace();
        }
    }
});
```



Sérialiseurs

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

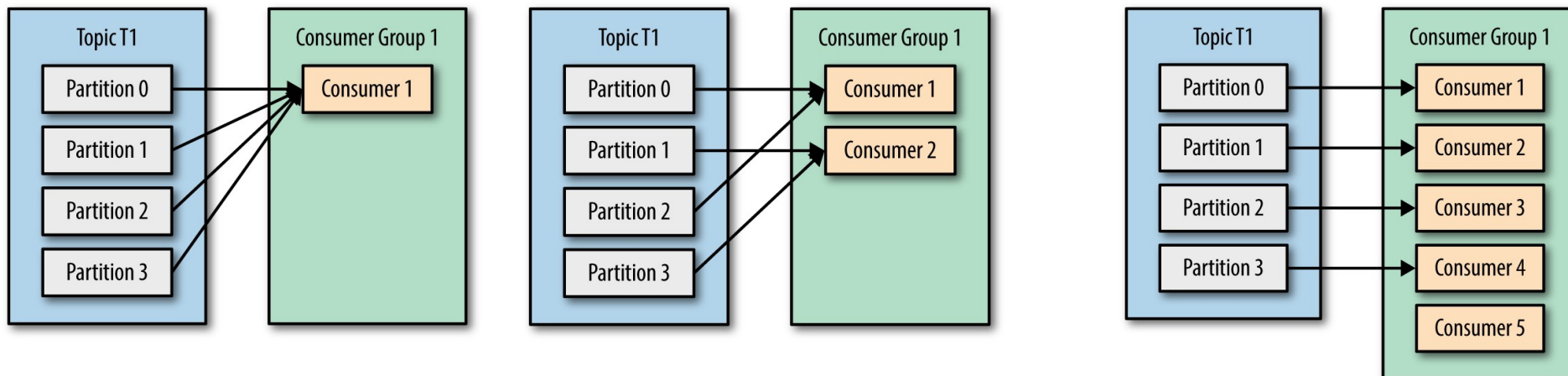
Pour des objets du domaine, il faut implémenter ses propres sérialiseurs/désérialiseurs en s'appuyant sur des formats comme *Avro*, *Protobuf* ou *JSON*

L'utilisation de ***Schema Registry*** permet de s'assurer que les évolutions de format sont compatibles.

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

- Durant le rééquilibrage les messages ne sont pas consommés
- Le rééquilibrage peut donner lieu à des pertes de messages ou à des doublons (en fonction de la configuration)



KafkaConsumer

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

Les propriétés obligatoires sont :

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

Une fois créé le consumer s'abonne à une liste de topic et poll régulièrement les messages.

Un poll récupère un lot de messages.



Exemple Java

```
private Properties kafkaProps = new Properties();
// Renseignement de bootstrap-servers, key et value Deserializer + group.id
// Plus d'autres configurations optionnelles
...
consumer = new KafkaConsumer<String, String>(kafkaProps);

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

try {
while (true) {
    // poll timeout de 100ms pour récupérer les messages
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        // Traitement message ..
    }
}
} finally {
    consumer.close();
}
```



Gestion des commits

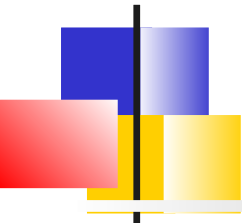
La gestion des commits (indiquer à Kafka que le message a été traité) influe sur la garantie de consommation des messages.

2 stratégies :

- Laisser le client Kafka effectuer régulièrement les commits (*`enable.auto.commit=true`*)
- Gérer soit même les commits

Si la consommation consiste à produire un message vers Kafka, on peut associer l'envoi du commit vers Kafka à la validation d'une transaction.

=> Exactly Once



Consommateur/Producteur Exactly-Once

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



Autres API

Kafka propose 2 autres APIs :

- **Admin API** : Client d'administration permettant de gérer et inspecter les topics, brokers, configurations et ACLs
- **Connect API** : Permettant l'intégration à d'autres systèmes BD, ElasticSearch, etc
- **Streams API¹** : Notre sujet !

1. Équivalent Python : *Faust*, .NET : *Streamiz*



KafkaStream

Concepts

Démarrage

Configuration

Opérateurs stateless

Opérateurs stateful

Requêtes interactives

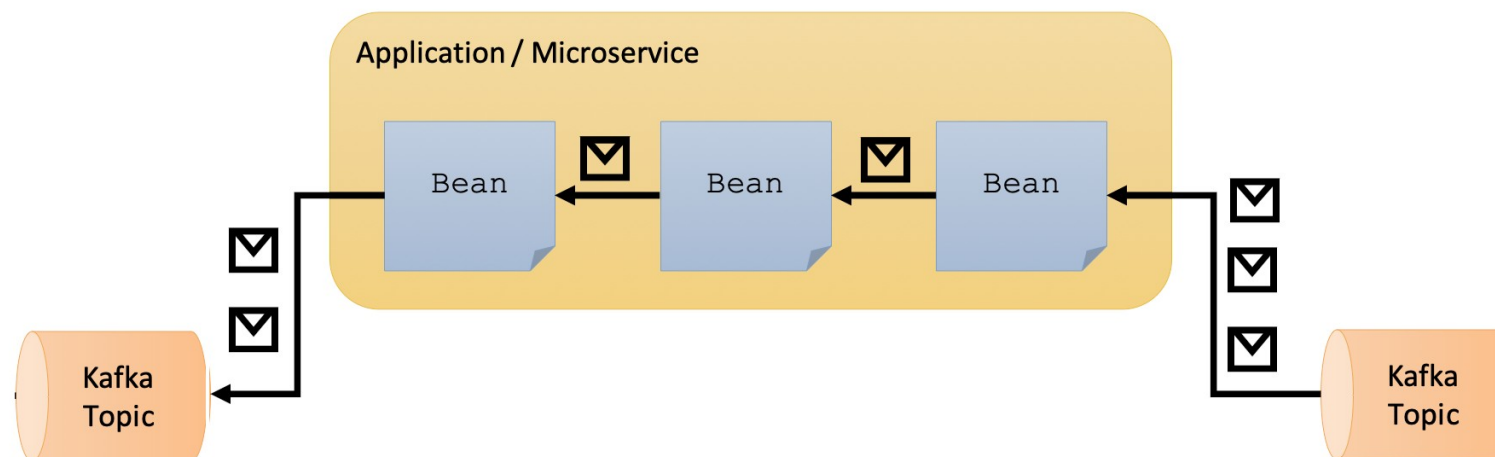
Kafka Streams

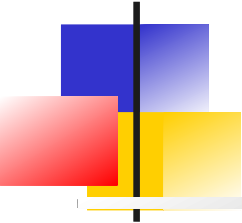
Kafka Streams API est une bibliothèque cliente pour la création d'applications, où les données d'entrée et de sortie sont stockées dans des topics Kafka

Une seule dépendance :

org.apache.kafka: kafka-streams

+ Eventuellement des sérialiseurs spécialisés





Application KafkaStream

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

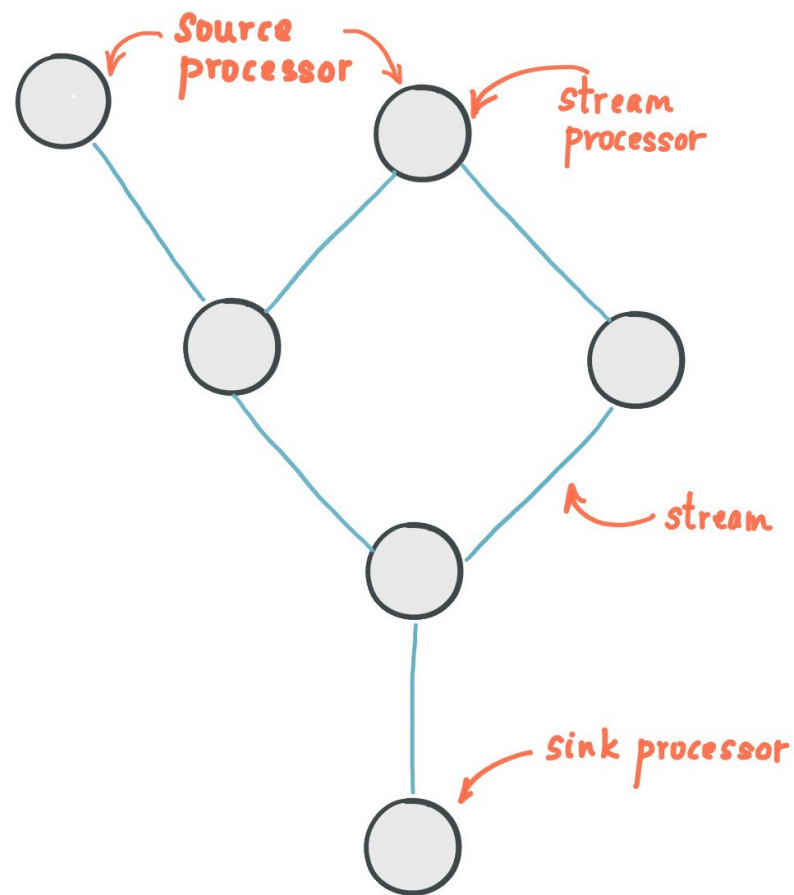
Un processeur représente une étape de traitement qui prend en entrée un évènement, le transforme puis produit un ou plusieurs évènement

Certains processeurs :

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

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

Topologie processeurs



PROCESSOR TOPOLOGY



Apports de *KafkaStream*

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



Définitions

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

Les caractéristiques de ce flux :

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

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



KStream et KTable

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

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

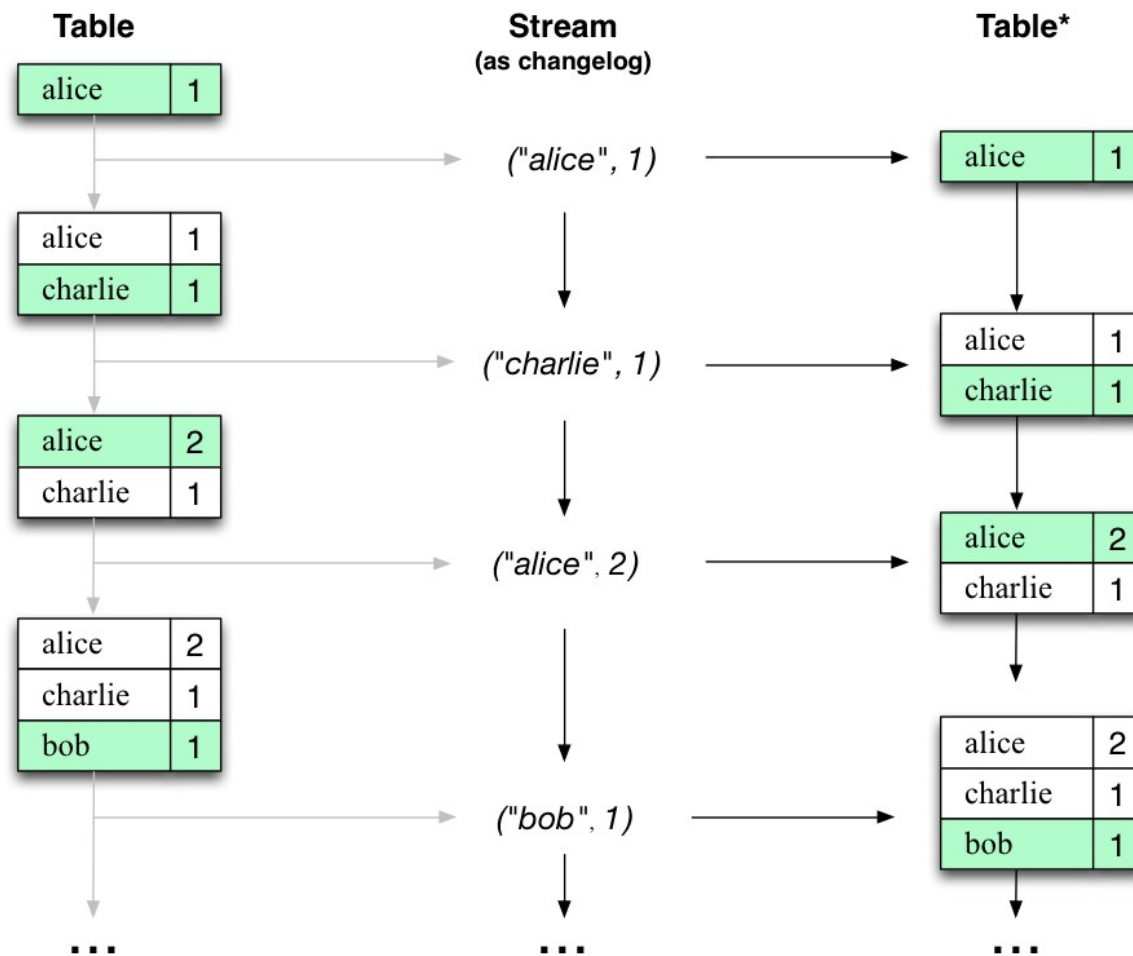
Cela peut être vu comme une table d'une base de données où il n'existe qu'une valeur pour une clé donnée.

Cela correspond à un topic Kafka avec un archivage compact

A table

key1	value1
key2	value2
key3	value3

Dualité KStream / KTable





KGlobalTable

KGlobalTable représente une table distribuée globale.

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

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



Opérateurs stateless / stateful

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

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

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

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

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



Tâches et partitions

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

Les tâches permettent la scalabilité.

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



Modèle de threads

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

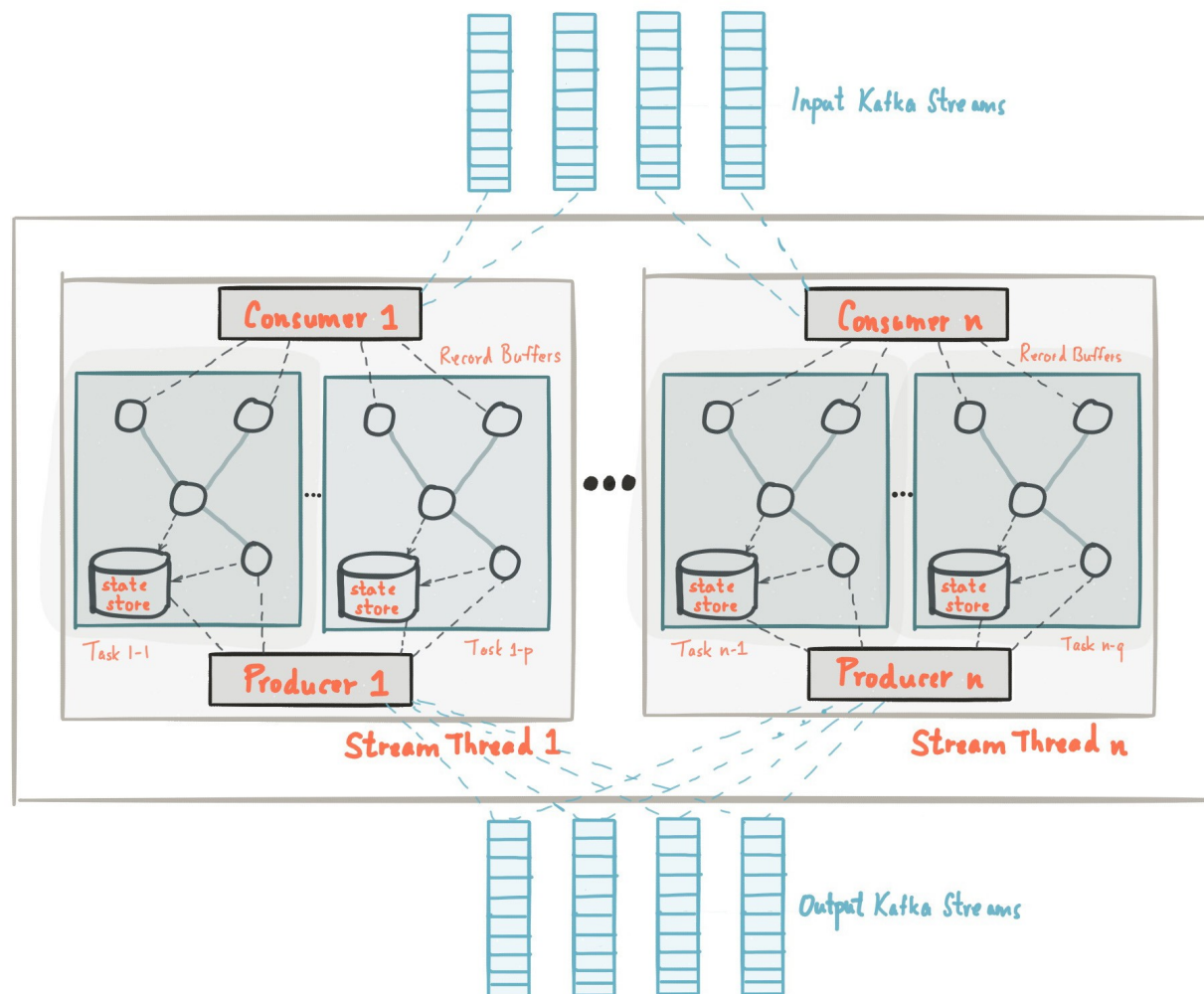
- Chaque thread peut exécuter une ou plusieurs tâches avec leurs topologies de processeur de manière indépendante.

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

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

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

Architecture et scalabilité





Tolérance aux pannes

Chaque tâche maintient son propre *LocalStore*.

Pour garantir la tolérance aux pannes,
KafkaStream crée un topic en mode compact
qui contient les dernières valeurs du Store

Lors d'une réaffectation de tâches, la nouvel
instance doit reconstruire l'état du *LocalStore*

Pour minimiser le coût de la restauration, il est
possible de répliquer les Local Store sur
différentes instances, (*num.standby.replica*) .



KafkaStream

Concepts

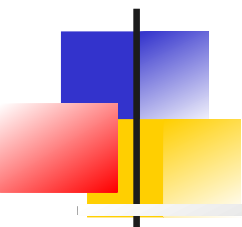
Démarrage

Configuration

Opérateurs stateless

Opérateurs stateful

Requêtes interactives



Structure d'une application

- Typiquement, une application `KafkaStream` :
 - Positionne les valeurs de configuration via une classe *Properties* (*bootstrap-servers*, *Serdes*, *ApplicationId*)
 - Construit une topologie à l'aide de ***StreamsBuilder***
 - Instancie un ***KafkaStreams*** à partir de la topologie et des propriétés
 - Implémente un moyen d'arrêter l'application proprement puis démarre le *KafkaStreams*



Exemple

// Propriétés : ID, BOOTSTRAP, Serialiseur/Désérialiseur

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

// Création d'une topologie de processeurs

```
final StreamsBuilder builder = new StreamsBuilder();
builder.<String, String>stream("streams-plaintext-input")
    .flatMapValues(value -> Arrays.asList(value.split("\\W+")))
    .to("streams-linesplit-output");
```

```
final Topology topology = builder.build();
```

// Instanciation du Stream à partir d'une topologie et des propriétés

```
final KafkaStreams streams = new KafkaStreams(topology, props);
```



Exemple (2)

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

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

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



Serde

Les applications KafkaStream ont besoin de sérialiser et désérialiser les clé et les valeurs du record

Le type **Serde** encapsule donc un sérialiseur et un désérialiseur.

```
public interface Serde<T> extends Closeable {  
    Serializer<T> serializer();  
    Deserializer<T> deserializer();  
}
```

Les Serde pour la clé peut être différent de celui utilisé pour la valeur.

Les Serde à utiliser peuvent être spécifié :

- En définissant des Serde par défaut
- En spécifiant explicitement les Serde lors de l'appel des méthodes de l'API.

Durant l'exécution d'une topologie, plusieurs Serdes sont généralement utilisés.



Serdes

KafkaStream fourni des Serdes pour les types de base : String, Double, Integer, Long, UUID, byte[], ByteBuffer, Void

On peut surcharger ou implémenter des propres Serdes

- Cela demande d'implémenter 3 classes : un sérialiseur, un désérialiseur un Serde
-

Confluent fournit des Serde pour les formats classiques : Json, Avro, Protobuf



Exemples

```
Properties settings = new Properties();
// Serde par défaut pour les clés
settings.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass().getName());
// Serde par défaut pour les valeurs
settings.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    Serdes.Long().getClass().getName());

---
// Précision du Serde pour la consommation
KStream<String, Coursier> coursierStream = builder.stream(
    "input-topic",
    Consumed.with(Serdes.String(), new CoursierSerde())
);

---
// Précision pour la production
KStream<String, Long> userCountByRegion = ...;
userCountByRegion.to("RegionCountsTopic", Produced.with(stringSerde, longSerde));
```



Exemple JsonSerde

```
public class JsonSerde<T> implements Serde<T> {
    private final ObjectMapper objectMapper = new ObjectMapper();
    private final Class<T> targetType;

    public JsonSerde(Class<T> targetType) {
        this.targetType = targetType;
    }

    @Override
    public Serializer<T> serializer() {
        return (topic, data) -> {
            try {
                return objectMapper.writeValueAsBytes(data);
            } catch (Exception e) {
                throw new RuntimeException("Erreur de s rialisation", e);
            }
        };
    }

    @Override
    public Deserializer<T> deserializer() {
        return (topic, data) -> {
            try {
                return objectMapper.readValue(data, targetType);
            } catch (Exception e) {
                throw new RuntimeException("Erreur de d s rialisation", e);
            }
        };
    }
}
```



Confluent Avro Serde

La bibliothèque ***io.confluent :kafka-streams-avro-serde*** fournit
2 Serde pour Avro :

- ***GenericAvroSerde*** permet de dé/sérialiser des GenericRecord de Avro
- ***SpecificAvroSerde*** permet de dé/sérialiser des classes spécifiques générées par les outils Avro

Exemple SpecificAvroSerde :

```
final Map<String, String> serdeConfig = Collections.singletonMap("schema.registry.url",  
                                                                "http://my-schema-registry:8081");  
  
// `Bar` est généré à partir d'un Schéma Avro  
final Serde<Bar> valueSpecificAvroSerde = new SpecificAvroSerde<>();  
valueSpecificAvroSerde.configure(serdeConfig, false); // `false` pour valeurs, true pour clé  
  
KStream<Foo, Bar> textLines = builder.stream("my-avro-topic", Consumed.with(Serdes.string(),  
                                   valueSpecificAvroSerde));
```



Création de Stream

L'API permet de créer des *KStream* à partir de topics

Soit les *Serde* par défaut sont utilisés, soit ils sont précisés via l'API

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



Création de Table ou GlobalTable

Les tables peuvent également être construites à partir d'un topic.

Un nom, identifiant le Store associé doit être fourni pour permettre les requêtes interactives.

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



Opérateur to

L'opérateur **to** permet de déverser un *KStream* dans un topic.

```
aStream.to(OUTPUT_TOPIC,  
    Produced.with(Serdes.String(), valueSerde));
```

L'opérateur ***streamToTable*** permet de convertir une table en *KStream* avant de le déverser dans un topic :

```
KStream<String, Long> streamFromTable = table.toStream();  
// Écrire le KStream résultant dans un topic Kafka  
streamFromTable.to("output-topic", Produced.with(Serdes.String(),  
    Serdes.Long()));
```



KafkaStream

Concepts

Démarrage

Configuration

Opérateurs stateless

Opérateurs stateful

Requêtes interactives



Configuration

La configuration s'effectue via une classe ***Properties***

```
Properties settings = new Properties();

// Set a few key parameters
settings.put(StreamsConfig.APPLICATION_ID_CONFIG,
"my-first-streams-application");
settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"kafka-broker1:9092");
// Any further settings
settings.put(... , ...);
```




Configuration

Paramètres requis :

- ***application.id*** : Identifiant unique de l'application (groupe de consommateurs)
- ***bootstrap-servers*** :

Configuration pour la résilience :

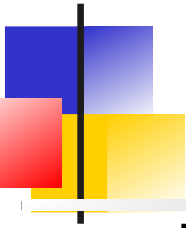
- ***acks*** : Défaut all
- ***replication.factor*** : Le facteur de réplication des topics internes créés par KafkaStream. Recommandation > 1 et généralement égal au facteur de réplication du topic source
- ***processing.guarantee*** : ***At Least Once*** ou ***Exactly Once***



Configuration

Principales configuration optionnelles

- ***default.keySerde, default.valueSerde*** : Les sérialiseur/désérialiseur par défaut pour les clés et les valeurs
- ***num.stream.threads*** : Nombre de threads par instance d'application.
- ***default.deserialization.exception.handler, default.production.exception.handler*** : Gestionnaires d'erreur. KafkaStream en fournit 2 prédéfinis : *LogAndContinueExceptionHandler, LogAndFailExceptionHandler*
- ***default.timestamp.extractor*** : La classe responsable de résoudre le timestamp du message. Important pour le windowing

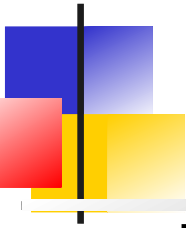


Configuration des clients Kafka

Les clients interne de KafkaStream
(Consommateur, Producteur, Admin)
peuvent également être configurés.

KafkaStream propose un mécanisme
permettant de fixer une propriété pour
tous les clients ou seulement l'un d'entre
eux.

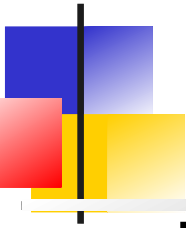
Certaines configurations ne peuvent pas
être surchargées.



Configuration des clients Kafka

Les propriétés peuvent être préfixées par consumer., produce. ou admin.

```
Properties streamsSettings = new Properties();  
// Même valeur pour tous les clients  
streamsSettings.put("PARAMETER_NAME", "value");  
// Valeur différente pour chaque client  
streamsSettings.put("consumer.PARAMETER_NAME", "consumer-value");  
streamsSettings.put("producer.PARAMETER_NAME", "producer-value");  
streamsSettings.put("admin.PARAMETER_NAME", "admin-value");  
// Idem avec constante  
streamsSettings.put(StreamsConfig.consumerPrefix("PARAMETER_NAME"),  
    "consumer-value");  
streamsSettings.put(StreamsConfig.producerPrefix("PARAMETER_NAME"),  
    "producer-value");  
streamsSettings.put(StreamsConfig.adminClientPrefix("PARAMETER_NAME"),  
    "admin-value");
```



Configuration consommateurs

Les différents consommateurs peuvent également être distingués :

- les consommateurs principaux consommant le topic source (*main.consumer*),
- les consommateurs des *LocalStore* (*restore.consumer*)
- ou des *GlobalTable* (*global.consumer*)

```
Properties streamsSettings = new Properties();  
// Configuration pour tous les consommateurs  
streamsSettings.put("consumer.PARAMETER_NAME", "general-consumer-value");  
// Seulement pour le restore.consumer  
streamsSettings.put("restore.consumer.PARAMETER_NAME", "restore-consumer-value");  
// Idem avec constante  
streamsSettings.put(StreamsConfig.restoreConsumerPrefix("PARAMETER_NAME"), "restore-consumer-value");
```



Configuration des topics internes

De la même façon, les topics utilisés en interne ***changelog*** et ***repartition*** peuvent être configurés en utilisant le préfixe topic

```
Properties streamsSettings = new Properties();  
// Surcharge de la valeur par défaut pour tous les topics  
streamsSettings.put("topic.PARAMETER_NAME", "topic-value");  
// idem  
streamsSettings.put(StreamsConfig.topicPrefix("PARAMETER_NAME"),  
    "topic-value");
```



Valeurs par défaut

client.id=<application.id>-<random-UUID>

Consommateur :

- `auto.offset.reset=earliest`
- `max.poll.records=1000`

Producteur :

- `linger.ms=100`

Si Exactly Once :

- `transaction.timeout.ms = 10000`
- `delivery.timeout.ms = Integer.MAX_VALUE`



Paramètres contrôlé par KafkaStreams

Certains paramètres ne peuvent pas être surchargés :

- *allow.auto.create.topics=false* (pour le consommateur)
- *group.id = application.id*
- *enable.auto.commit=false*
- *partition.assignment.strategy=StreamsPartitionAssignor*

Si Exactly Once:

- *isolation.level=read_committed*
- *enable.idempotence=true*



Client.id

client.id est utilisé pour dériver d'autres noms utilisés en interne.

- Consommateur principal :
<client.id>-StreamThread-<threadIdx>-consumer
- Consommateur de restauration :
<client.id>-StreamThread-<threadIdx>-restore-consumer
- Consommateur pour les tables globales :
<client.id>-global-consumer
- Producteur :
<client.id>-StreamThread-<threadIdx>-producer
- Pour l'admin :
<client.id>-admin



KafkaStream

Concepts

Démarrage

Configuration

Opérateurs stateless

Opérateurs stateful

Requêtes interactives

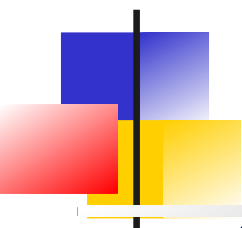


Processeurs stateless

Les processeurs stateless ne nécessitent pas de *StateStore*.

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

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



Les processeurs stateless

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

map, mapValues : Transformation du message

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

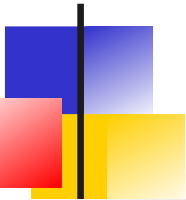
branch : Split le flux d'entrée en plusieurs flux

merge : Fusionne 2 flux

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

print : Opération terminale affichant le message sur la console

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



Les processeurs stateless (2)

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

groupByKey : Regroupe en utilisant la clé

coGroup : Agrégation de plusieurs stream ayant les mêmes clés

selectKey : Permet de définir une nouvelle clé

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

tableToStream, streamToTable :

repartition : Repartitionne le flux avec le nombre de partitions voulus



Examples

// Branch

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

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



Examples

// FkatMap

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

// Merge

```
KStream<byte[], String> merged = stream1.merge(stream2);
```

// Re-partitionnement

```
KStream<byte[], String> repartitionedStream =  
    stream.repartition(Repartitioned.numberOfPartitions(10));
```



Exemples Side Effect

```
// ForEach : Kstream → void, KTable → void
```

```
stream.foreach((key, value) -> System.out.println(key + " => " + value));
```

```
// Print : Kstream → void
```

```
stream.print(Printed.toFile("streams.out").withLabel("streams"));
```

```
// Peek : Kstream → Kstream
```

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




Exemples, groupes

```
// SelectKey : KStream → KStream
```

```
KStream<String, String> rekeyed = stream.selectKey((key, value) -> value.split(" ")[0])
```

```
// GroupByKey : KStream → KgroupedStream
```

```
KGroupedStream<byte[], String> groupedStream = stream.groupByKey();
```

```
// GroupBy : KStream → KgroupedStream ou KTable → KGroupedTable
```

```
KGroupedTable<String, Integer> groupedTable = table.groupBy(  
    (key, value) -> KeyValue.pair(value, value.length()),  
    Grouped.with(  
        Serdes.String(), /* key (note: type was modified) */  
        Serdes.Integer()) /* value (note: type was modified) */  
    );
```

```
// Cogroup : KGroupedStream → CogroupedKStream, CogroupedKStream → CogroupedKStream
```

```
KGroupedStream<byte[], String> groupedStream = stream.groupByKey();
```

```
KGroupedStream<byte[], String> groupedStream2 = stream2.groupByKey();
```

```
CogroupedKStream<byte[], String> cogroupedStream =  
    groupedStream.cogroup(agggregator1).cogroup(groupedStream2, aggregator2);
```



KafkaStream

Concepts
Démarrage
Configuration
Opérateurs stateless
Opérateurs stateful
Requêtes interactives



Processeurs stateful

Les opérateurs stateful disponibles sont :

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



State Store associés

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

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

3 types de StateStore sont possibles :

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

1. Les tables versionnées apportent des restrictions



Agrégations

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

Ces opérateurs génèrent :

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

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



Opérations d'agrégation

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

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

Prend en arguments :

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

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

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



Exemple count

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

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



Agrégation en continu

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

```
// Aggregating a KGroupedStream
```

```
KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
```

```
    () -> 0L, /* Initialisation */
```

```
    (aggKey, newValue, aggValue) ->  
        aggValue + newValue.length(), /* Ajout */
```

```
    Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>  
        as("aggregated-stream-store") /* Nom du stateStore */
```

```
        .withValueSerde(Serdes.Long()); /* serde pour la valeur agrégée*/
```




Fenêtrage

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

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

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

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



Types de fenêtres

Le DSL supporte différents types de fenêtres :

- **Hopping time** : Fenêtres de taille fixe qui se chevauchent
- **Tumbling time** : Fenêtres de taille fixe, sans chevauchement et sans espace
- **Sliding time** : Fenêtres de taille fixe se chevauchant et qui fonctionnent sur les différences entre les horodatages d'enregistrement
- **Session** : Fenêtres de taille dynamique, sans chevauchement et pilotées par les données.
Timeout d'inactivité



Agrégation avec fenêtrage

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

```
// Aggregation avec des fenêtres basculant tous les 5 minutes
```

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



Reduce

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

```
// KGroupedStream
```

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

```
// KgroupedTable, utilisation d'un subtractor pour mettre à jour un
```

```
// changement de groupe
```

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



	KTable <code>userProfiles</code>			KGroupedTable <code>groupedTable</code>			KTable <code>aggregate</code> d
Timestamp	Input record	Interpreted as	Grouping	Initializer	Adder	Subtractor	State
1	(alice, E)	INSERT alice	(E, 5)	0 (for E)	(E, 0 + 5)		(E, 5)
2	(bob, A)	INSERT bob	(A, 3)	0 (for A)	(A, 0 + 3)		(A, 3) (E, 5)
3	(charlie, A)	INSERT charlie	(A, 7)		(A, 3 + 7)		(A, 10) (E, 5)
4	(alice, A)	UPDATE alice	(A, 5)		(A, 10 + 5)	(E, 5 - 5)	(A, 15) (E, 0)
5	(charlie, null)	DELETE charlie	(null, 7)			(A, 15 - 7)	(A, 8) (E, 0)
6	(null, E)	<i>ignored</i>					(A, 8) (E, 0)
7	(bob, E)	UPDATE bob	(E, 3)		(E, 0 + 3)	(A, 8 - 3)	(A, 5) (E, 3)



Topics internes

Lors d'opérations d'agrégation, des topics internes appelés **changelog** et **repartition** sont créés automatiquement.

Ils servent à assurer la durabilité, la tolérance aux pannes et à faciliter le traitement distribué.

- **changelog** stocke durablement les modifications apportées à un (state store) associé à une *KTable* ou à une agrégation. Il permet de reconstituer le Store en cas de défaillance ou lors d'un redémarrage
- **repartition** est utilisés lorsque les données doivent être re-partitionnées en fonction d'une nouvelle clé. (clé différente de la clé de partitionnement d'origine)



Jointures

Les jointures peuvent s'appliquer :

- Entre 2 stream, le fenêtrage s'applique.
Le résultat est un *KStream*
- Entre un Stream et une KTable, pas de fenêtrage.
Le résultat est un *KStream*
- Entre 2 tables, pas de fenêtrage.
Le résultat est une KTable

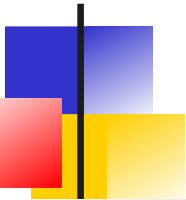


Pré-requis pour les jointures

Pour la plupart des jointures, les données d'entrée doivent être copartitionnées.

- Cela garantit que les enregistrements d'entrée avec la même clé des deux côtés de la jointure sont transmis à la même tâche de flux pendant le traitement.

Cette contrainte n'est pas requise pour les jointures vers les *GlobalKTable* ou avec les clé-étrangères



Jointure KStream / KStream

Ces jointures sont toujours fenêtrées.

Chaque nouvel enregistrement d'un des topics produira une sortie de jointure pour chaque enregistrement correspondant de l'autre côté.

Les enregistrements de sortie de jointure sont créés avec un **ValueJoiner** fourni par l'utilisateur :

```
KeyValue<K, LV> leftRecord = ...;
KeyValue<K, RV> rightRecord = ...;
ValueJoiner<LV, RV, JV> joiner = ...;

KeyValue<K, JV> joinOutputRecord = KeyValue.pair(
    leftRecord.key,
    joiner.apply(leftRecord.value, rightRecord.value)
);
```



Exemple Inner Join

```
KStream<String, Long> left = ...;
KStream<String, Double> right = ...;

// Jointure leftRecord.key == rightRecord.key pour une fenêtre de 5mn
KStream<String, String> joined = left.join(right,
    // Valeur résultante : Value Joiner
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofMinutes(5)),
    Joined.with(
        Serdes.String(), /* key */
        Serdes.Long(),   /* left value */
        Serdes.Double()) /* right value */
    );
```



Exemple Outer Join

```
KStream<String, Long> left = ...;
KStream<String, Double> right = ...;

// Les enregistrements arrivant à droite et n'ayant pas de correspondance à gauche
// génèrent une valeur avec leftValue = null
KStream<String, String> joined = left.outerJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /*
    ValueJoiner */
    JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofMinutes(5)),
    Joined.with(
        Serdes.String(), /* key */
        Serdes.Long(), /* left value */
        Serdes.Double()) /* right value */
```



Jointure KTable / KTable

Ces jointures sont non fenêtrées et sont équivalentes au modèle relationnel

Les données doivent être co-partitionnées.

Exemple *leftJoin* :

```
KTable<String, Long> left = ...;
```

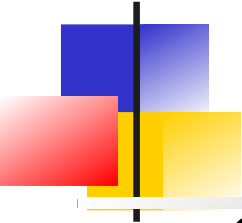
```
KTable<String, Double> right = ...;
```

```
KTable<String, String> joined = left.leftJoin(right,
```

```
    // ValueJoiner
```

```
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" +  
    rightValue
```

```
);
```



Jointure KTable / KTable avec une clé étrangère

Ces jointures sont non fenêtrées et sont équivalent au modèle relationnel

Les données doivent être co-partitionnés.

La *KTable* de gauche peut contenir plusieurs enregistrements qui correspondent à la même clé sur la *KTable* de droite.

- Une mise à jour d'une entrée de la *KTable* de gauche peut entraîner un seul événement de sortie (à condition que la clé correspondante existe dans la *KTable* de droite).
- Une seule mise à jour d'une entrée de la *KTable* de droite entraînera une mise à jour pour chaque enregistrement de la *KTable* de gauche qui a la même clé étrangère.



Exemple Clé-étrangère

```
KTable<String, Long> left = ...;  
KTable<Long, Double> right = ...;
```

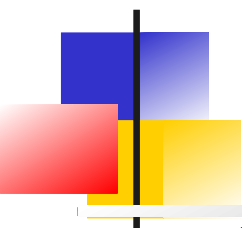
```
// foreignKeyExtractor simply uses the left-value to map to the right-key.
```

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

```
// La jointure s'effectue quand :
```

```
// foreignKeyExtractor.apply(leftRecord.value) == rightRecord.key
```

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



Jointure KStream / KTable

Ces jointures sont non fenêtrées, elles permettent d'enrichir le flux avec des données de la table.

Les données doivent être co-partitionnés.

La jointure s'effectue sur la clé : *leftRecord.key == rightRecord.key*

- Seuls les messages du (flux) déclenchent la jointure. Les messages de la table ne mettent à jour uniquement l'état de jointure interne du côté droit.
- Les messages avec une clé nulle ou une valeur nulle sont ignorés
- Les enregistrements d'entrée de la table avec une valeur nulle indique la suppression de la clé de la table.



Exemple leftJoin :

```
KStream<String, Long> left = ...;
KTable<String, Double> right = ...;

KStream<String, String> joined = left.join(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" +
    rightValue,
    Joined.keySerde(Serdes.String()) /* key */
    .withValueSerde(Serdes.Long()) /* left value */
    .withGracePeriod(Duration.ZERO)
    // Grace period : tolérance pour attendre
    // une maj tardive de la table
);
```




Jointure *KStream* / *KGlobalTable*

Ces jointures sont non fenêtrées, elles permettent d'enrichir le flux avec des données de la table.

Les données ne nécessitent pas d'être co-partitionnés.

La jointure sur clé étrangère est possible

Plus efficaces qu'une KTable partitionné lorsque vous devez effectuer plusieurs jointures successivement.

```
KStream<String, Long> left = ...;  
GlobalKTable<Integer, Double> right = ...;
```

```
KStream<String, String> joined = left.join(right,  
    (leftKey, leftValue) -> leftKey.length(), /* Nouvelle clé pour la jointure */  
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue  
);
```



KafkaStream

Concepts
Démarrage
Configuration
Opérateurs stateless
Opérateurs stateful
Requêtes interactives



Introduction

Les requêtes interactives permettent d'interroger les *StateStore* de l'application.

Les requêtes peuvent être locales ou remote
(Ajout d'une couche RPC)

Elles peuvent concerner tout type de store
(Windowed ou non)



Requêtes locales

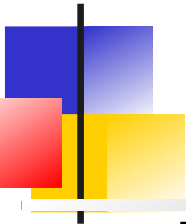
Une application Kafka Streams s'exécute généralement sur plusieurs instances.

L'état disponible localement sur une instance donnée n'est qu'un sous-ensemble de l'état complet de l'application.

Les requêtes locales ne renverront que les données disponibles localement sur cette instance particulière.

La méthode `KafkaStreams#store(...)` recherche les `StateStore` locaux d'une instance d'application par son nom et type.

- Le nom du `StateStore` est donné à la création
- 2 types sont supportés par Kafka
 - *KeyValueStore*
 - *WindowStore*



Création des KeyValueStore

Les stateStore sont nécessaires pour les opérations d'agrégation.

```
groupedByWord.count(Materialized.<String, String,  
    KeyValueStore<Bytes, byte[]>as("CountsKeyValueStore"));
```

Mais il est également possible de matérialiser le résultat d'une opération stateless dans un store

```
KTable<String, Integer> oddCounts =  
    numberLines.filter((region, count) -> (count % 2 != 0),  
    Materialized.<String, Integer, KeyValueStore<Bytes,  
    byte[]>as("queryableStoreName"));
```



Requêtage de KeyValueStore

Après avoir démarré l'application, on récupère une instance de ***ReadOnlyKeyValueStore*** et on l'interroge via les clés .

```
ReadOnlyKeyValueStore<String, Long> keyValueStore =  
    streams.store("CountsKeyValueStore", QueryableStoreTypes.keyValueStore());  
  
// Get value by key  
System.out.println("count for hello:" + keyValueStore.get("hello"));  
  
// Get the values for a range of keys available in this application instance  
KeyValueIterator<String, Long> range = keyValueStore.range("all", "streams");  
while (range.hasNext()) {  
    KeyValue<String, Long> next = range.next();  
    System.out.println("count for " + next.key + ": " + next.value);  
}  
  
// Get the values for all of the keys available in this application instance  
KeyValueIterator<String, Long> range = keyValueStore.all();  
while (range.hasNext()) {  
    KeyValue<String, Long> next = range.next();  
    System.out.println("count for " + next.key + ": " + next.value);  
}
```



Création de WindowStore

Un magasin de fenêtres peut potentiellement avoir de nombreux résultats pour une clé donnée, car la clé peut être présente dans plusieurs fenêtres. Cependant, il n'y a qu'un seul résultat par fenêtre pour une clé donnée.

Le store est créé par des applications stateful.

```
groupedByWord.windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofSeconds(60)))  
    .count(Materialized.<String, Long, WindowStore<Bytes, byte[]>as("CountsWindowStore"));
```



Requêtage de WindowStore

Après avoir démarré l'application, on récupère une instance de ***ReadOnlyWindowStore*** et on l'interroge via les clés et un intervalle de temps .

```
ReadOnlyWindowStore<String, Long> windowStore =  
    streams.store("CountsWindowStore", QueryableStoreTypes.windowStore());  
  
// Récupère les valeurs pour la clé "world" pour toutes les fenêtres disponibles  
// L'intervalle est à partir du début du temps jusqu'à maintenant.  
Instant timeFrom = Instant.ofEpochMilli(0);  
Instant timeTo = Instant.now();  
WindowStoreIterator<Long> iterator = windowStore.fetch("world", timeFrom,  
    timeTo);  
while (iterator.hasNext()) {  
    KeyValue<Long, Long> next = iterator.next();  
    long windowTimestamp = next.key;  
    System.out.println("Count of 'world' @ time " + windowTimestamp + " is " +  
        next.value);  
}
```




Requêtes distantes

Pour pouvoir interroger à distance l'état global d'une application KafkaStream, plusieurs étapes sont nécessaires :

- Ajoutez une couche RPC afin que les instances de l'application puissent interagir via le réseau (par exemple, une API REST, etc.).
- Exposez les endpoint RPC via le paramètre de configuration ***application.server*** de Kafka Streams.
Chaque instance a sa propre valeur pour ce paramètre de configuration.
- Dans la couche RPC, découvrir les instances d'application distantes et leurs StateStore et interrogez les magasins d'état disponibles localement.
 - => Les instances d'application distantes peuvent transmettre des requêtes à d'autres instances d'application si une instance particulière ne dispose pas des données locales pour répondre à une requête.
 - => Les magasins d'état disponibles localement peuvent répondre directement aux requêtes.



Exemple Couche RPC

```
KafkaStreams streams = ...;

// Retrouver tous les emplacements des StateStore nommé "word-count"
Collection<StreamsMetadata> wordCountHosts = streams.allMetadataForStore("word-
count");

HttpClient http = ...;

// Get the word count for word (aka key) 'alice'
//
// Retrouver le store qui contient la clé 'alice'
StreamsMetadata metadata = streams.metadataForKey("word-count", "alice",
    Serdes.String().serializer());
// Appel du endpoint renvoyant le résultat
Long result = http.getLong("http://" + metadata.host() + ":" + metadata.port()
    + "/word-count/alice");
```

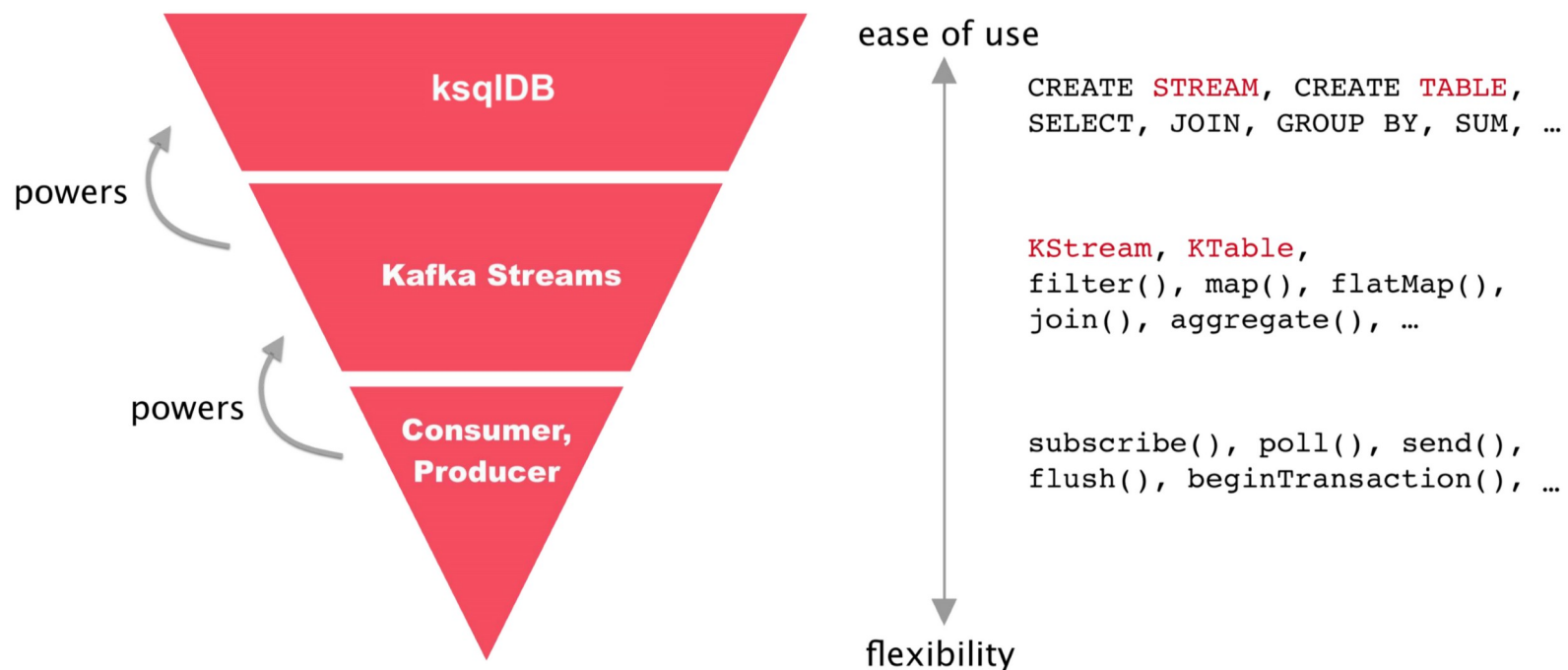


ksqlDB

Présentation
SQL
Statement
Fonctions disponibles

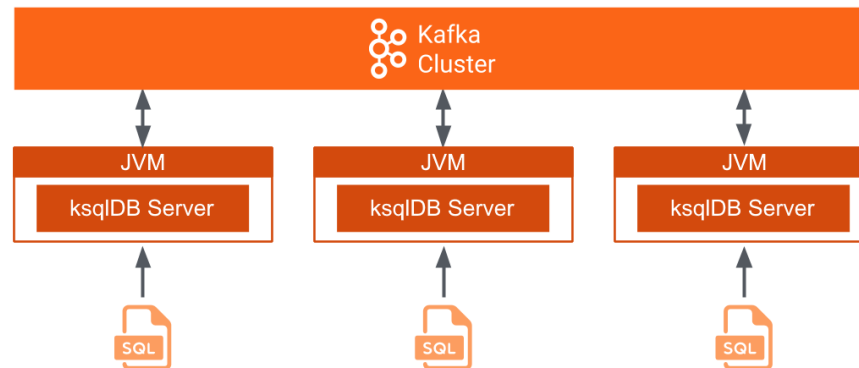
ksqlDB

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



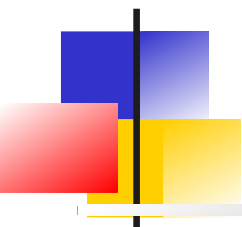
Architecture ksqlDB

ksqlDB Standalone Application (Headless Mode)



Intégration via

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

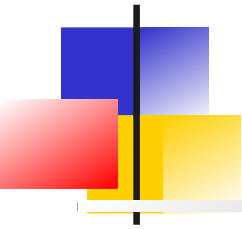


ksqlDB

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.

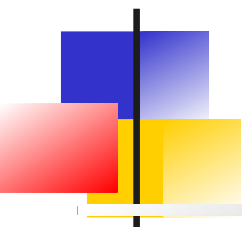


ksqlDB

ksqlDB est donc un serveur

L'interaction avec le serveur peut se faire via :

- Son API REST
- Un client (Java, .NET, Python)
- 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;
```



ksqlDB

Présentation
Données
Statement
Fonctions disponibles



Syntaxe

La grammaire SQL de ksqlDB a été initialement construite autour de la grammaire de Presto et a été étendue.

ksqlDB va au-delà de SQL-92, car la norme n'a actuellement aucune construction pour les requêtes en streaming.

Une instruction est composée :

- De mots clés (SELECT, INSERT, CREATE, ...)
- D'identifieurs, i.e. noms pour les streams, les tables, les colonnes, généralement entouré par des backquote
- Des constantes typées (string, number, boolean)
- Des opérateurs (=, !=, LIKE, BETWEEN)
- De commentaires (lignes préfixées par --)



Données lignes/colonnes

ksqlDB abstrait les événements sous forme de lignes avec des colonnes et les stocke dans des stream et des tables.

Les streams et les tables modélisent des collections d'événements qui s'accumulent au fil du temps.

Les deux sont représentés sous la forme d'une série de lignes et de colonnes avec un schéma, à la manière d'une table de base de données relationnelle.

Les lignes représentent des événements individuels. Les colonnes représentent les attributs de ces événements.

Chaque colonne a un type



Colonnes Clé / Valeur

Une colonne est soit une colonne clé soit une colonne valeur

La colonne clé contrôle la partition où réside la ligne

Il est obligatoire de créer une PRIMARY KEY sur une Table

Chaque ligne représente un enregistrement d'un topic Kafka

Si une colonne est déclarée dans un schéma, mais qu'aucun attribut n'est présent dans l'enregistrement Kafka sous-jacent, la valeur de la colonne de la ligne est renseignée comme nulle.



Streams

Un stream est partitionné, immuable et append-only

CREATE STREAM permet de créer un stream à partir d'un topic Kafka, précréé ou créé automatiquement lors de l'exécution du statement

```
CREATE STREAM s1 (  
    k VARCHAR KEY,  
    v1 INT,  
    v2 VARCHAR  
) WITH (  
    kafka_topic = 's1',  
    partitions = 3, // Pas précisé si le topic existe  
    value_format = 'json'  
);
```




Tables

Une table est partitionnée et muable

CREATE TABLE permet de créer une table à partir d'un topic Kafka, précréé ou créé automatiquement lors de l'exécution du statement

```
CREATE TABLE current_location (  
    person VARCHAR PRIMARY KEY,  
    location VARCHAR  
) WITH (  
    kafka_topic = 'current_location',  
    partitions = 3,  
    value_format = 'json'  
);
```



PseudoColonnes

Une pseudo-colonne est une colonne automatiquement renseignée par ksqlDB . Elles ne sont pas renvoyées lors d'un SELECT *.

Les pseudo-colonnes disponibles :

- **HEADERS** : les entêtes de l'enregistrement Kafka.
- **ROWOFFSET** : L'offset de l'enregistrement .
- **ROWPARTITION** : La partition de l'enregistrement source.
- **ROWTIME** : Le timestamp

Exemple :

```
SELECT ROWTIME, * FROM s1 EMIT CHANGES;
```



Clé-primaires

ksqlDB utilise les clés primaires mais pas de la même façon qu'une BD relationnelle :

- Seules les tables peuvent avoir des clés primaires. Les Stream ne les prennent pas en charge.
- L'ajout de plusieurs lignes à une table avec la même clé primaire n'entraîne pas le rejet des lignes suivantes.

Les clés primaires ne peuvent pas être nulles et doivent être utilisées dans toutes les tables déclarées:



Types de données

Booléen : *boolean*

Caractères : *varchar, string, bytes* (taille variable)

Numériques : *int, bigint, double, decimal*
(précision et scale)

Temporel : *time, date, timestamp* (java.sql.*)

Types composés : *array, struct* (package Kafka, i.e données structurées typées), *map*

Types Custom : Peuvent être créés par *CREATE TYPE*



Types composés

Ils peuvent être définis lors d'un CREATE (TABLE/STREAM) ou créer à la volée dans une requête SELECT

```
CREATE STREAM orders (  
  order_id VARCHAR,  
  delivery_address STRUCT<  
    street VARCHAR,  
    city VARCHAR,  
    postal_code VARCHAR  
  >  
  >
```

```
) WITH (  
  KAFKA_TOPIC='orders_topic',  
  VALUE_FORMAT='JSON'  
);
```

```
SELECT STRUCT(f1 := v1, f2 := v2) FROM s1 EMIT CHANGES;
```

```
SELECT MAP(k1:=v1, k2:=v1*2) FROM s1 EMIT CHANGES;
```



Timestamp

Les opérations basées sur le temps, comme le fenêtrage, traitent les enregistrements en fonction de ROWTIME. La précision est d'une milliseconde.

La propriété **TIMESTAMP** permet de remplacer ROWTIME par le contenu de la colonne spécifiée.

```
CREATE STREAM TEST (id BIGINT KEY, event_timestamp VARCHAR)
WITH (
    kafka_topic='test_topic',
    value_format='JSON',
    timestamp='event_timestamp',
    timestamp_format='yyyy-MM-dd' 'T' 'HH:mm:ssX'
);
```



ksqlDB

Présentation
Données
Statement
Fonctions disponibles



PULL QUERY

Exécute la requête et se termine

Le résultat n'est pas conservé dans un topic Kafka, il est affiché dans la console ou renvoyé au client.

Par défaut, seules les recherches de clés sont activées. :

- Les colonnes clés doivent utiliser une comparaison d'égalité avec un littéral (par exemple, KEY = 'abc').
- Sur les tables fenêtrées, WINDOWSTART et WINDOWEND peuvent éventuellement être comparés à des littéraux.

Si la propriété 'ksql.query.pull.table.scan.enabled'='true'; ces restrictions sur la clause WHERE sont levées

```
SELECT select_expr [, ...]  
  FROM from_item -- Materialized View, Stream ou Table  
  [ WHERE where_condition ]  
  [ AND window_bounds ]  
  [ LIMIT count ];
```




Exemples PULL QUERY

-- Création d'une table, la table n'est pas requêtable en PULL

```
CREATE TABLE GRADES (ID INT PRIMARY KEY, GRADE STRING, RANK INT) WITH  
  (kafka_topic = 'test_topic', value_format = 'JSON', partitions = 4);
```

-- Création d'une table dérivée avec un SELECT

```
CREATE TABLE TOP_TEN_RANKS  
  AS SELECT ID, RANK  
  FROM GRADES  
  WHERE RANK <= 10;
```

-- Récupération via un ID

```
SELECT * FROM TOP_TEN_RANKS  
  WHERE ID = 5;
```

-- Autorisation du scan de table

```
SET 'ksql.query.pull.table.scan.enabled'='true';
```

-- Recherche par une colonne

```
SELECT * FROM TOP_TEN_RANKS  
  WHERE RANK > 4 AND RANK < 8;
```



PUSH QUERY

Pousse un flux continu de mises à jour vers un stream ou une table.

Le résultat du select n'est pas conservé dans un topic Kafka et est imprimé uniquement dans la console ou renvoyé au client.

Permet de s'abonner aux mises à jour, et de réagir en temps réel.

```
SELECT select_expr [, ...]
  FROM from_item -- Un stream ou une table
  [[ LEFT | FULL | INNER ]
    JOIN join_item
      [WITHIN [<size> <timeunit> | (<before_size> <timeunit>, <after_size>
<timeunit>)] [GRACE PERIOD <grace_size> <timeunit>]]
    ON join_criteria]*
  [ WINDOW window_expression ]
  [ WHERE where_condition ]
  [ GROUP BY grouping_expression ]
  [ HAVING having_expression ]
  EMIT [ output_refinement ] -- CHANGE ou FINAL
  [ LIMIT count ];
```



Exemples PUSH QUERY

```
SELECT * FROM pageviews
```

```
WHERE ROWTIME >= '2017-11-17T04:53:45'  
      AND ROWTIME <= '2017-11-17T04:53:48'  
      EMIT CHANGES;
```

```
-- orders est un stream, le fenêtrage est disponible sur les stream  
SELECT windowstart, windowend, item_id, SUM(quantity)  
      FROM orders  
      WINDOW TUMBLING (SIZE 20 SECONDS)  
      GROUP BY item_id  
      EMIT CHANGES;
```



Jointures

Les jointures respectent les règles des jointures de KafkaStream

- Une jointure entre 2 Stream doivent spécifier une fenêtre temporelle avec WITHIN
- Un jointure entre STREAM et TABLE produit un STREAM, une grace periode peut être spéciiée
- Une jointure entre 2 TABLE produit une TABLE

Les jointures servent généralement à créer de nouveau STREAM ou TABLE



Exemples

```
-- Jointure STREAM STREAM
CREATE STREAM position_statut AS
  SELECT c.coursierId,
         c.statut,
         p.position->latitude AS latitude,
         p.position->longitude AS longitude
  FROM statut_stream c
  JOIN position_stream p
  WITHIN 5 MINUTES -- Fenêtre de jointure de 5 minutes
  ON c.coursierId = p.coursierId
  EMIT CHANGES;
```

```
-- Jointure STREAM TABLE
CREATE STREAM transaction_coursier AS
  SELECT t.coursierId,
         t.montant,
         c.statut
  FROM transaction_stream t
  JOIN coursier_table c
  ON t.coursierId = c.coursierId
  EMIT CHANGES
```



Exemples (2)

```
-- Jointure TABLE TABLE
CREATE TABLE coursier_position_table AS
SELECT c.coursierId,
       c.nom,
       c.statut,
       p.latitude,
       p.longitude
FROM   coursier_table c
JOIN   position_table p
ON     c.coursierId = p.coursierId
EMIT CHANGES;
```



Insertions

2 moyens pour insérer des données dans une stream ou une table

- Insertion de valeurs
- Insertions à partir d'un SELECT (seulement pour les streams)

-- Insert avec valeurs

```
INSERT INTO foo (ROWTIME, KEY_COL, COL_A) VALUES (1510923225000, 'key', 'A');
```

-- Insert avec SELECT

```
INSERT INTO large_orders  
SELECT order_id, customer, amount  
FROM orders  
WHERE amount > 100;
```



SHOW

SHOW permet de lister les objets du cluster

- STREAMS
- TABLES
- QUERIES
- TOPICS
- PROPERTIES
- CONNECTORS (KafkaConnect)
- FUNCTIONS
- VARIABLES (défini avec DEFINE)
- TYPE : Les types custom



DESCRIBE

DESCRIBE permet de lister les informations détaillées d'un objet, le mot clé EXTENDED permet d'accéder à des statistiques d'exécution

- STREAMS | TABLE : Tous les STREAM ou Toutes les table
- <Nom STREAM> | <Nom TABLE> : Seulement le STREAM/TABLE spécifié
- TOPICS
- CONNECTOR (KafkaConnect)
- FUNCTIONS



DROP

DROP permet de supprimer un objet

- DROP STREAM [IF EXISTS] stream_name [DELETE TOPIC];
- DROP TABLE [IF EXISTS] table_name [DELETE TOPIC];
- DROP CONNECTOR [IF EXISTS] connector_name;



ksqlDB

Présentation
Données
Statement
Fonctions disponibles



Fonctions

3 types de fonctions :

- Les fonctions **scalaires**
S'applique sur une colonne
- Les fonctions **d'agrégations**
Utilisées avec un GROUP BY
- Les fonctions de **table**
Retourne zéro ou plusieurs lignes



Fonctions scalaires

Numériques : Arrondis, Trigonométrie, Valeur absolue, Racine carré, Puissance, Log, Exponentiel, Distance géo, Plus grandes valeurs entre colonnes, ...

Sur les collections : Longueur, test si contient, Accès à un élément, tri, intersection, union, map, ...

Invocations de fonction (lambda) : Filter, Transform, Reduce

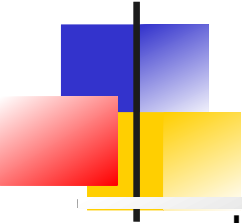
Chaînes de caractères : split, trim, regexp, substring, replace, mask, fonctions JSON

Bytes : Conversion en numérique

Null : Valeur alternative si null, Valeur null si condition, première valeur non null entre plusieurs colonnes

Date et Time : Ajout/soustraction, Parsing, Formatting, Conversion

URLs : extraction des champs (host,...), Encoding/Decoding,



Fonctions d'agrégation

Les classiques :

COUNT, COUNTDISTINCT, AVG, MIN, MAX, SUM, TOP

Les avancés :

COLLECT_LIST, COLLECT_SET, HISTOGRAM, STDEV, CORRELATION

Spécifiques Kafka:

EARLIEST_BY_OFFSET, LATEST_BY_OFFSET

Exemple :

```
CREATE TABLE customer_avg_amount AS
  SELECT customer,
         AVG(amount) AS avg_amount
  FROM orders
 GROUP BY customer
  EMIT CHANGES;
```



Fonctions de table

Les fonctions tables sont équivalent à un flatMap en programmation réactive.

- Elles ne s'appliquent que sur des STREAM
- Elles se spécifie dans la clause SELECT d'une requête
- Elle renvoie plusieurs lignes mais une seule colonne

Par exemple, appliqué à un tableau, elle renvoie toutes les valeurs du tableau

Exemple :

-- Données tu topic

```
{sensor_id:12345 readings: [23, 56 ]}  
{sensor_id:54321 readings: [12, 65, 38]}
```

-- Utilisation de EXPLODE

```
CREATE STREAM exploded_stream AS  
  SELECT sensor_id, EXPLODE(readings) AS reading FROM batched_readings;
```

-- Résultat

```
{sensor_id:12345 reading: 23}  
{sensor_id:12345 reading: 56}  
{sensor_id:54321 reading: 12}  
{sensor_id:54321 reading: 65}  
{sensor_id:54321 reading: 38}
```



Fonctions de table

EXPLODE(array) : Affiche une valeur pour chacun des éléments du tableau.
Les valeurs de sortie ont le même type que les éléments du tableau.

CUBE_EXPLODE(array[col1, ..., colN]) :
Pour le tableau de colonnes spécifié, génère toutes leurs combinaisons possibles.
Produit 2^n lignes