OXiane INSTITUT

make it **clever**

SOMMAIRE

SOMMAIRE

5. Schema registry

- Gestion des schéma (avro, json)
- API de manipulations

6. Streaming

- Introduction au concept de streaming et pipeline de données
- Comparatifs streams vs topics

7. KSQLDB

- Concepts et architecture de KSQLDB
- Requêtes KSQLDB et opérations en ligne de commande
- Traitement des données issues d'un stream
- Streams & Tables
- Jointures, agrégations et fenêtres de temps et de taille
- Développer une extension KSQLDB spécifique

Concepts fondamentaux

1. Producers sont des applications ou des services qui envoient des données à Kafka. Ils publient des **messages** dans des **topics**. Un topic est une catégorie ou un flux de données dans Kafka.

- Les producteurs envoient des messages dans des topics spécifiques.
- Ils peuvent partitionner les messages pour améliorer la distribution et la scalabilité.
- Les messages sont envoyés de manière synchrone ou asynchrone.

2. Consumers sont des applications ou des services qui lisent des données de Kafka. Ils s'abonnent à des topics et consomment les messages publiés par les producteurs.

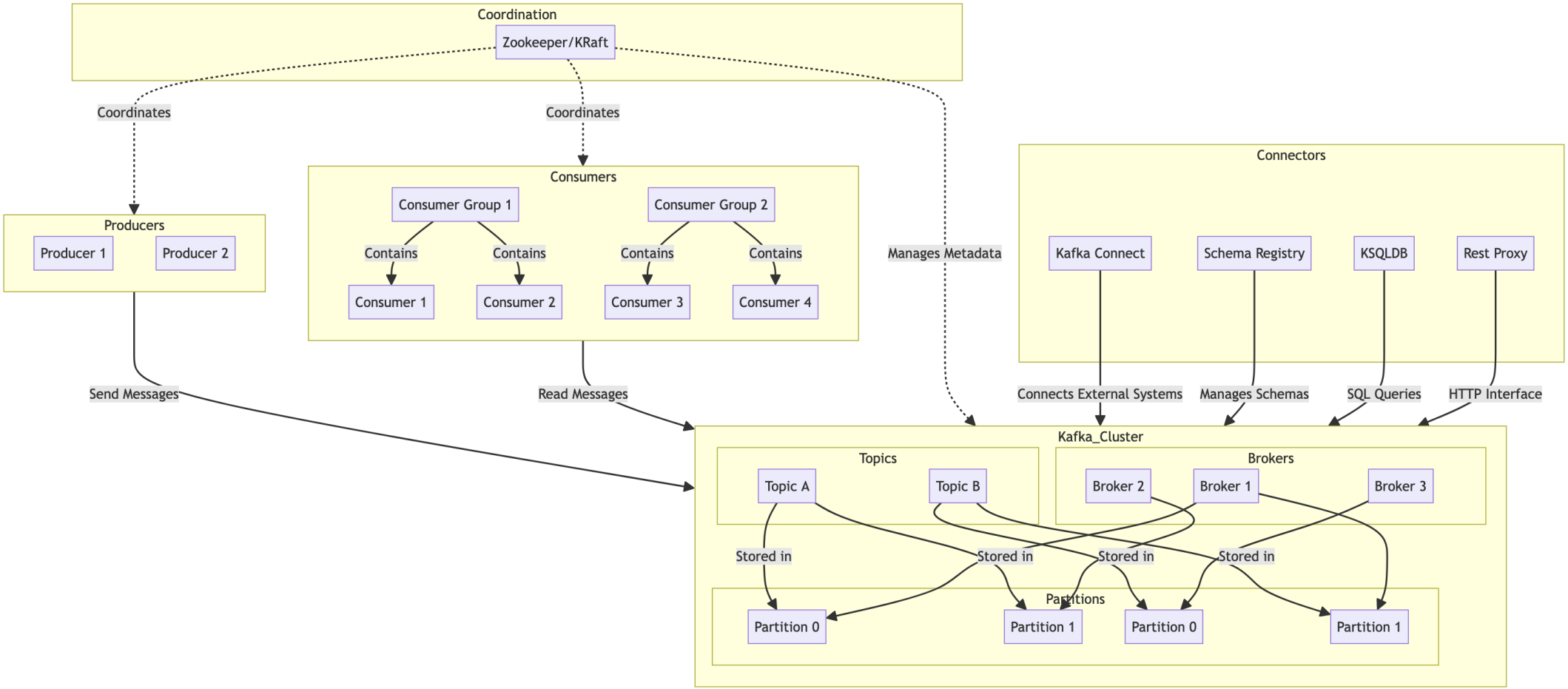
- Les consommateurs lisent les messages d'un topic.
- Chaque consommateur dans un **consumer group** lit des messages de partitions spécifiques, permettant une lecture parallèle.
- Les offsets des messages (positions de lecture) sont utilisés pour suivre les messages déjà consommés.

Concepts fondamentaux

3. Messages sont les unités de données envoyées par les producteurs à Kafka et lues par les consommateurs. Chaque message contient généralement une clé, une valeur et des métadonnées.

- Les messages sont publiés par les producteurs dans des topics.
- Les messages sont stockés de manière ordonnée dans des partitions.
- Chaque message dans une partition a un offset unique.

Architecture d'une plateforme Kafka



Architecture d'une plateforme Kafka

1. Kafka Cluster: Un cluster Kafka est constitué de plusieurs brokers (serveurs) qui stockent et gèrent les données.

- **Brokers** : Chaque broker est responsable de stocker les partitions de plusieurs topics. Les brokers communiquent entre eux pour la réplication des données et l'élection du leader.
- **Topics** : Les topics sont des flux de messages où les données sont publiées. Les topics sont divisés en partitions pour améliorer la scalabilité et la performance.
- **Partitions** : Les partitions sont des sous-ensembles d'un topic. Elles permettent de paralléliser la lecture et l'écriture des données pour améliorer la scalabilité.

2. Kafka Connect: Kafka Connect est un outil pour connecter Kafka avec des systèmes externes (bases de données, systèmes de fichiers, etc.).

- Kafka Connect utilise des connecteurs pour lire des données de systèmes externes et les publier dans des topics Kafka, ou pour lire des données de topics Kafka et les écrire dans des systèmes externes.

3. Schema Registry: Le Schema Registry est un service pour gérer les schémas de données (comme Avro, JSON Schema) utilisés dans Kafka.

- Le Schema Registry stocke et valide les schémas des messages. Les producteurs et consommateurs valident leurs messages contre ces schémas avant de les écrire ou de les lire de Kafka.

Architecture d'une plateforme Kafka

4. KSQLDB: KSQLDB est une base de données de streaming pour Kafka. Elle permet d'effectuer des requêtes SQL en temps réel sur les données en streaming.

- KSQLDB permet de créer des flux et des tables basées sur les données en temps réel dans Kafka, et de les interroger avec des requêtes SQL.

5. Rest Proxy: Le Rest Proxy fournit une interface HTTP pour interagir avec Kafka. Il permet de produire et consommer des messages via des requêtes REST.

- Les applications peuvent utiliser des requêtes HTTP pour envoyer et recevoir des messages de Kafka sans utiliser les bibliothèques clients Kafka natives.

6. Zookeeper/KRaft: Zookeeper était traditionnellement utilisé pour la gestion de la configuration et la coordination des brokers Kafka. KRaft est une alternative plus récente intégrée directement dans Kafka.

- Zookeeper gère la configuration du cluster, la coordination des brokers et des consommateurs, et le suivi des offsets. KRaft, introduit dans Kafka 2.8.0, élimine la dépendance à Zookeeper en intégrant ces fonctions directement dans Kafka.

Architecture d’une plateforme Kafka

Interactions entre les Composants

1. **Producers et Topics** : Les producteurs publient des messages dans des topics Kafka.
2. **Partitions et Brokers** : Les messages dans les topics sont stockés dans des partitions, et chaque partition est gérée par un broker spécifique.
3. **Consumers et Topics** : Les consommateurs s'abonnent aux topics et lisent les messages des partitions, en suivant les offsets pour savoir où ils en sont.
4. **Kafka Connect** : Connecte Kafka à des systèmes externes, permettant l'ingestion et l'exportation de données.
5. **Schema Registry** : Gère les schémas des messages pour assurer la compatibilité des données.
6. **KSQLDB** : Permet de traiter et d'interroger les données en temps réel avec SQL.
7. **Rest Proxy** : Offre une interface HTTP pour interagir avec Kafka.
8. **Zookeeper/KRaft** : Gère la configuration et la coordination des composants du cluster Kafka.

Développement pour Kafka

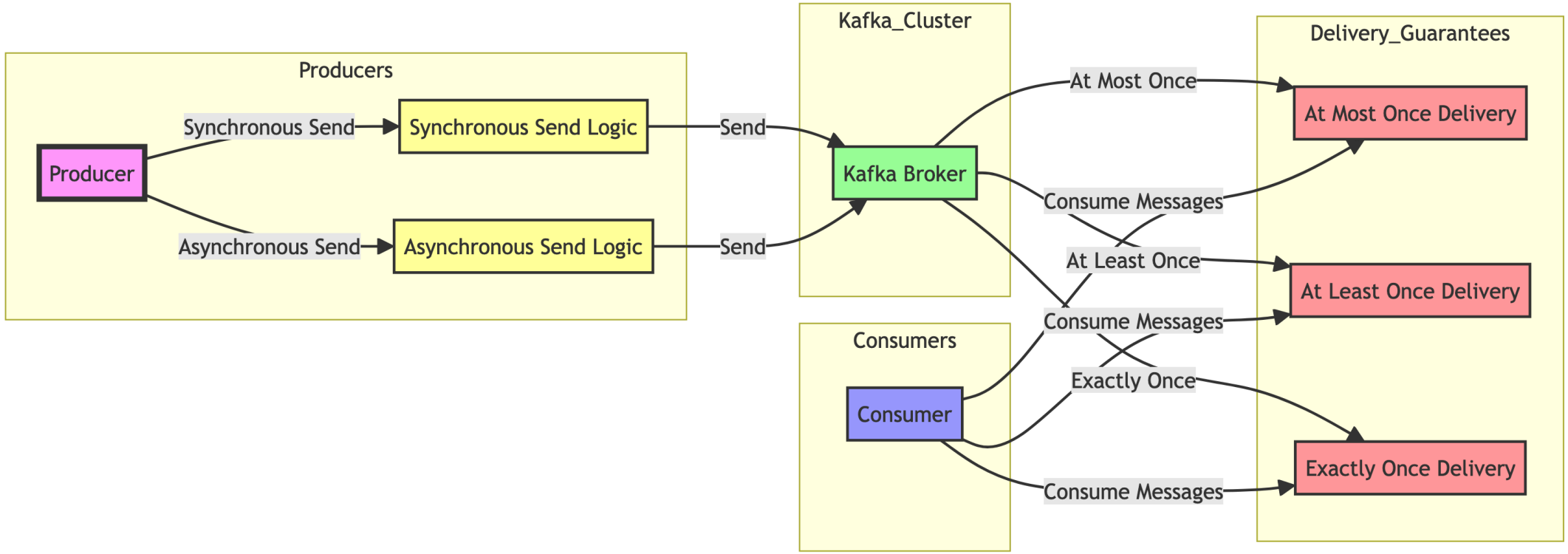
1. Java Development Kit (JDK) 8 ou supérieur
2. Apache Kafka
3. Maven ou Gradle
4. Spring Boot
5. Ajouter les dépendances Kafka :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
</dependencies>
```

6. Configurer Kafka dans Spring Boot :

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=my-group
spring.kafka.consumer.auto-offset-reset=earliest
```

Développement pour Kafka - Producer



Développement pour Kafka - Producer

1. Producers (Producteurs)

- **Synchronous Send** : Le producteur envoie un message et attend une confirmation avant de continuer. Cette méthode garantit que le message a été reçu par le broker.
- **Asynchronous Send** : Le producteur envoie un message sans attendre de confirmation immédiate. Il utilise des callbacks pour traiter les confirmations ou les erreurs de manière asynchrone.

2. Kafka Cluster

- Le broker Kafka reçoit les messages des producteurs. Il gère la persistance des messages dans les partitions des topics.

3. Delivery Guarantees (Garanties de Livraison)

- **At Most Once** : Les messages peuvent être perdus mais ne sont jamais doublés. Utilisé lorsque la perte de quelques messages est acceptable.
- **At Least Once** : Les messages sont toujours livrés, mais peuvent être doublés en cas de réessai. Utilisé lorsque la duplication est acceptable mais la perte de messages ne l'est pas.
- **Exactly Once** : Les messages sont livrés exactement une fois sans perte ni duplication. C'est la garantie la plus stricte, souvent utilisée dans les transactions critiques.

Développement pour Kafka - Producer

1. **KafkaTemplate**: `KafkaTemplate` est l'objet principal utilisé pour l'envoi de messages dans Kafka. Il est fourni par Spring Kafka et offre des méthodes pour envoyer des messages de manière synchrone et asynchrone.
2. **ProducerFactory**: `ProducerFactory` est utilisé pour créer des instances de `KafkaProducer`. Il configure les propriétés du producteur, telles que les serveurs bootstrap, les sérialiseurs, etc.
3. **ProducerRecord**: `ProducerRecord` est un objet représentant un message Kafka à envoyer. Il contient les informations du topic, de la clé (facultative), et de la valeur.
4. **SendResult**: `SendResult` est l'objet retourné après l'envoi d'un message, contenant des informations sur le message envoyé, telles que le topic, la partition, et l'offset.

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.acks=all
spring.kafka.producer.retries=10
spring.kafka.producer.batch-size=16384
spring.kafka.producer.linger-ms=1
spring.kafka.producer.buffer-memory=33554432
```

Développement pour Kafka - Producer

1. Production Synchrone

Dans la production synchrone, le producteur attend une confirmation de Kafka avant de continuer. Cela garantit que le message a été reçu par Kafka.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;

import java.util.concurrent.ExecutionException;

@Service
public class SynchronousProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public SynchronousProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);
        try {
            SendResult<String, String> result = future.get();
            System.out.printf("Sent message to topic %s partition %d offset %d%n",
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Développement pour Kafka - Producer

2. Production Asynchrone

Dans la production asynchrone, le producteur envoie un message sans attendre de confirmation immédiate. Un callback est utilisé pour traiter la confirmation ou les erreurs de manière asynchrone.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.CompletableFuture;
import org.springframework.util.concurrent.ListenableFutureCallback;

@Service
public class AsynchronousProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public AsynchronousProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        CompletableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);

        future.whenComplete((result, ex) -> {
            if (ex != null) {
                System.err.println("Failed to send message: " + ex.getMessage());
            } else {
                System.out.printf("Sent message to topic %s partition %d offset %d%n",
                        result.getRecordMetadata().topic(),
                        result.getRecordMetadata().partition(),
                        result.getRecordMetadata().offset());
            }
        });
    }
}
```

Développement pour Kafka - Producer

3. Production avec Accusés de Réception

Pour garantir que les messages sont reçus par tous les réplicas des partitions, vous pouvez configurer `acks=all`.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.support.SendResult;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException;

@Service
public class AckProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public AckProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        CompletableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, message);
        try {
            SendResult<String, String> result = future.get();
            System.out.printf("Sent message to topic %s partition %d offset %d%n",
                    result.getRecordMetadata().topic(),
                    result.getRecordMetadata().partition(),
                    result.getRecordMetadata().offset());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Développement pour Kafka - Producer

4. Production avec Transactions

Pour garantir l'atomicité des messages envoyés, vous pouvez utiliser les transactions Kafka.

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class TransactionalProducer {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public TransactionalProducer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
        this.kafkaTemplate.setTransactionIdPrefix("txn-");
    }

    @Transactional
    public void sendMessage(String topic, String message) {
        kafkaTemplate.executeInTransaction(operations -> {
            operations.send(topic, message);
            return true;
        });
    }
}
```

Développement pour Kafka - Producer

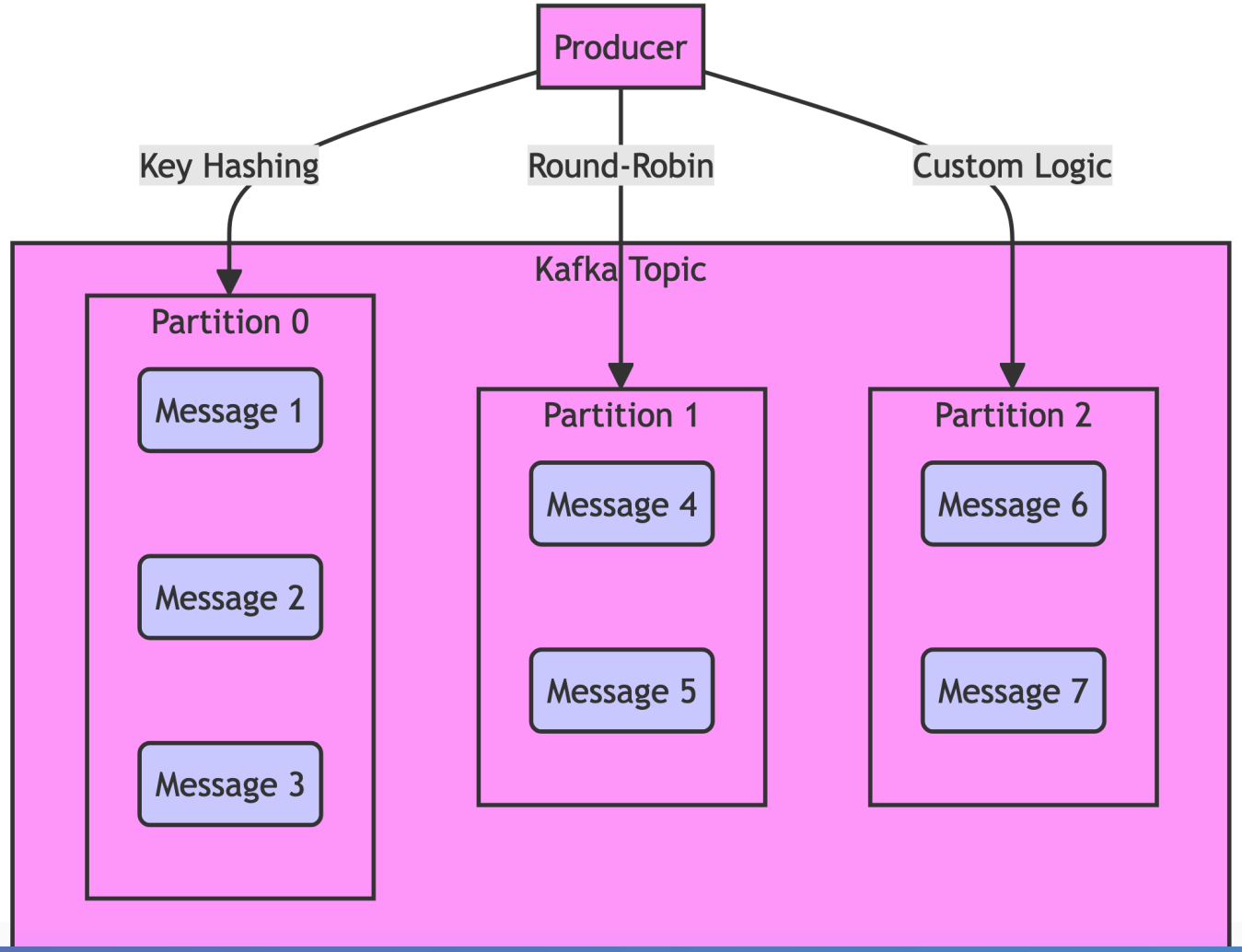
- **spring.kafka.bootstrap-servers** : Spécifie les adresses des serveurs Kafka à utiliser.
- **spring.kafka.producer.key-serializer** : Sérialiseur de clé pour le producteur.
- **spring.kafka.producer.value-serializer** : Sérialiseur de valeur pour le producteur.
- **spring.kafka.producer.acks** : Détermine le niveau d'accusés de réception souhaité. `all` garantit que tous les réplicas reconnaissent le message.
- **spring.kafka.producer.retries** : Nombre de tentatives de réessai en cas d'échec de l'envoi.
- **spring.kafka.producer.batch-size** : Taille des lots de messages à envoyer.
- **spring.kafka.producer.linger-ms** : Temps d'attente avant d'envoyer un lot de messages.
- **spring.kafka.producer.buffer-memory** : Taille de la mémoire tampon pour le producteur.

Exercice 1

Vous travaillez pour une entreprise qui développe un système de notifications pour un réseau social. Les notifications peuvent être envoyées pour divers événements tels que des likes, des commentaires, ou des nouveaux messages. Pour garantir la réactivité et la scalabilité du système, vous allez utiliser Apache Kafka pour gérer ces notifications. Vous devez implémenter deux types de producteurs Kafka : un producteur synchrone pour les notifications critiques (comme les nouveaux messages) et un producteur asynchrone pour les notifications moins critiques (comme les likes).

1. **Implémentation du Producteur Synchrone**
   - Implémentez un producteur Kafka synchrone pour envoyer des notifications critiques.
   - Assurez-vous que le producteur attend la confirmation de Kafka avant de continuer.
2. **Implémentation du Producteur Asynchrone**
   - Implémentez un producteur Kafka asynchrone pour envoyer des notifications générales.
   - Utilisez des callbacks pour gérer les confirmations ou les échecs de manière asynchrone.
3. **Test des Producteurs**
   - Créez des endpoints REST pour tester l'envoi de messages via les producteurs synchrones et asynchrones.
   - Envoyez des notifications critiques et générales et observez les comportements.

Partitionnement



Partitionnement

Dans Kafka, un "topic" est une catégorie ou un flux de messages. Pour permettre la scalabilité horizontale et une meilleure performance de lecture/écriture, chaque topic peut être divisé en plusieurs "partitions". Chaque partition est, en essence, un fichier journal immuable où les messages sont écrits de manière séquentielle.

Avantages du partitionnement :

- **Parallélisme** : Les partitions permettent à plusieurs consommateurs de lire les messages en parallèle, augmentant ainsi la capacité de traitement du système.
- **Réplication** : Chaque partition peut être répliquée sur plusieurs nœuds pour assurer la haute disponibilité et la durabilité des données.
- **Équilibrage de charge** : Distribuer les messages sur plusieurs partitions peut aider à équilibrer la charge entre différents serveurs.

Comment fonctionne le partitionnement ?

- Lorsque vous produisez un message, vous pouvez spécifier une clé. Kafka utilise cette clé pour attribuer le message à une partition spécifique en utilisant une fonction de hachage, ce qui garantit que tous les messages avec la même clé vont à la même partition.
- Si aucune clé n'est spécifiée, Kafka distribue les messages de manière round-robin entre toutes les partitions disponibles du topic, assurant ainsi une distribution équilibrée des messages.

Partitionnement personnalisé dans Apache Kafka

Le partitionnement personnalisé vous permet de contrôler la logique de distribution des messages à des partitions spécifiques, basée sur des critères autres que le simple hachage de clés ou le round-robin. Cela peut être utile pour des raisons de performances ou pour des exigences métier spécifiques.

Implémentation avec Kafka :

1. **Créer une classe de partitionneur personnalisé** : Implémentez l'interface `Partitioner` dans votre code. Cette interface requiert la définition de la méthode `partition()`, où vous spécifiez comment les messages sont attribués aux partitions.
2. **Configurer le producteur** : Lorsque vous configurez votre producteur Kafka, spécifiez votre classe de partitionneur personnalisée en utilisant la propriété `partitioner.class`.

Exercice 2: Gestion des Partitions

- Intégrer une gestion personnalisée des partitions pour diriger les notifications vers des partitions spécifiques en fonction de leur type et de leur importance.

1. **Définition des Règles de Partitionnement**
   - Définissez des règles pour partitionner les messages envoyés par les producteurs. Par exemple, toutes les notifications de "nouveaux messages" peuvent aller à une partition spécifique, tandis que les "likes" peuvent être répartis de manière équilibrée entre les autres partitions.
   - Implémentez une classe de partitionnement personnalisée dans Kafka pour appliquer ces règles.
2. **Modification des Producteurs pour Utiliser le Partitionnement Personnalisé**
   - Modifiez le producteur synchrone et asynchrone pour utiliser votre partitionneur personnalisé. Assurez-vous que le type de notification influence la partition à laquelle le message est envoyé.
   - Pour le producteur synchrone, garantissez que les notifications critiques sont envoyées à des partitions qui sont moins susceptibles de surcharger, pour assurer une livraison rapide.
   - Pour le producteur asynchrone, utilisez un partitionnement qui optimise l'utilisation des ressources et équilibre la charge entre les partitions.

Développer un consumer pour s’abonner à un topic kafka

Dans Apache Kafka, les consommateurs jouent un rôle crucial dans le traitement et la consommation des messages stockés dans les topics. Kafka offre des mécanismes flexibles pour que les consommateurs récupèrent les messages, principalement à travers deux modes de consommation : **subscribe** (s'abonner) et **assign** (attribuer).

1. Subscribe (S'abonner)

Quand des consommateurs utilisent `subscribe`, ils s'abonnent à un ou plusieurs topics. Cela signifie qu'ils indiquent à Kafka leur intention de recevoir des messages de ces topics spécifiques. Les caractéristiques de ce mode incluent :

- **Groupe de Consommateurs** : Les consommateurs qui s'abonnent à un topic sont généralement organisés en groupes de consommateurs. Kafka garantit que chaque partition d'un topic est consommée par un seul membre du groupe à la fois, ce qui permet une consommation équilibrée et une haute disponibilité.
- **Rééquilibrage Automatique** : Si un nouveau consommateur rejoint le groupe ou si un consommateur existant quitte le groupe ou échoue, Kafka réorganise automatiquement les partitions parmi les consommateurs actifs du groupe. Cela s'assure que toutes les partitions restent consommées et aide à éviter des points de défaillance.

Développer un consumer pour s’abonner à un topic kafka

2. Assign (Attribuer)

Dans le mode `assign`, les consommateurs spécifient explicitement les partitions des topics qu'ils souhaitent consommer, sans utiliser le concept de groupe de consommateurs. Les points clés ici sont :

- **Contrôle Granulaire** : Cette méthode donne aux consommateurs un contrôle précis sur les partitions à consommer, ce qui peut être utile pour des cas d'utilisation avancés où un rééquilibrage automatique n'est pas souhaité ou nécessaire.
- **Pas de Rééquilibrage** : Il n'y a pas de rééquilibrage automatique des partitions comme avec `subscribe`. Le consommateur est entièrement responsable de la gestion des partitions qui lui sont attribuées.

Développer un consumer pour s’abonner à un topic kafka

- **Partitionnement Efficace** : Avoir plus de partitions permet une parallélisation accrue, mais aussi peut augmenter le temps de latence et les coûts de gestion. Le choix du nombre de partitions peut affecter significativement les performances et la scalabilité.
- **Facteur de Réplication** : Chaque partition peut être répliquée sur plusieurs serveurs pour garantir la durabilité et la haute disponibilité des données.

3. Mécanisme de Commit dans Kafka

Kafka maintient ce qu'on appelle un "offset" pour chaque consommateur dans chaque partition. L'offset est un pointeur qui indique la position du dernier message consommé dans la partition. Quand un consommateur lit des messages d'une partition, il avance cet offset message par message.

Le commit de l'offset permet de:

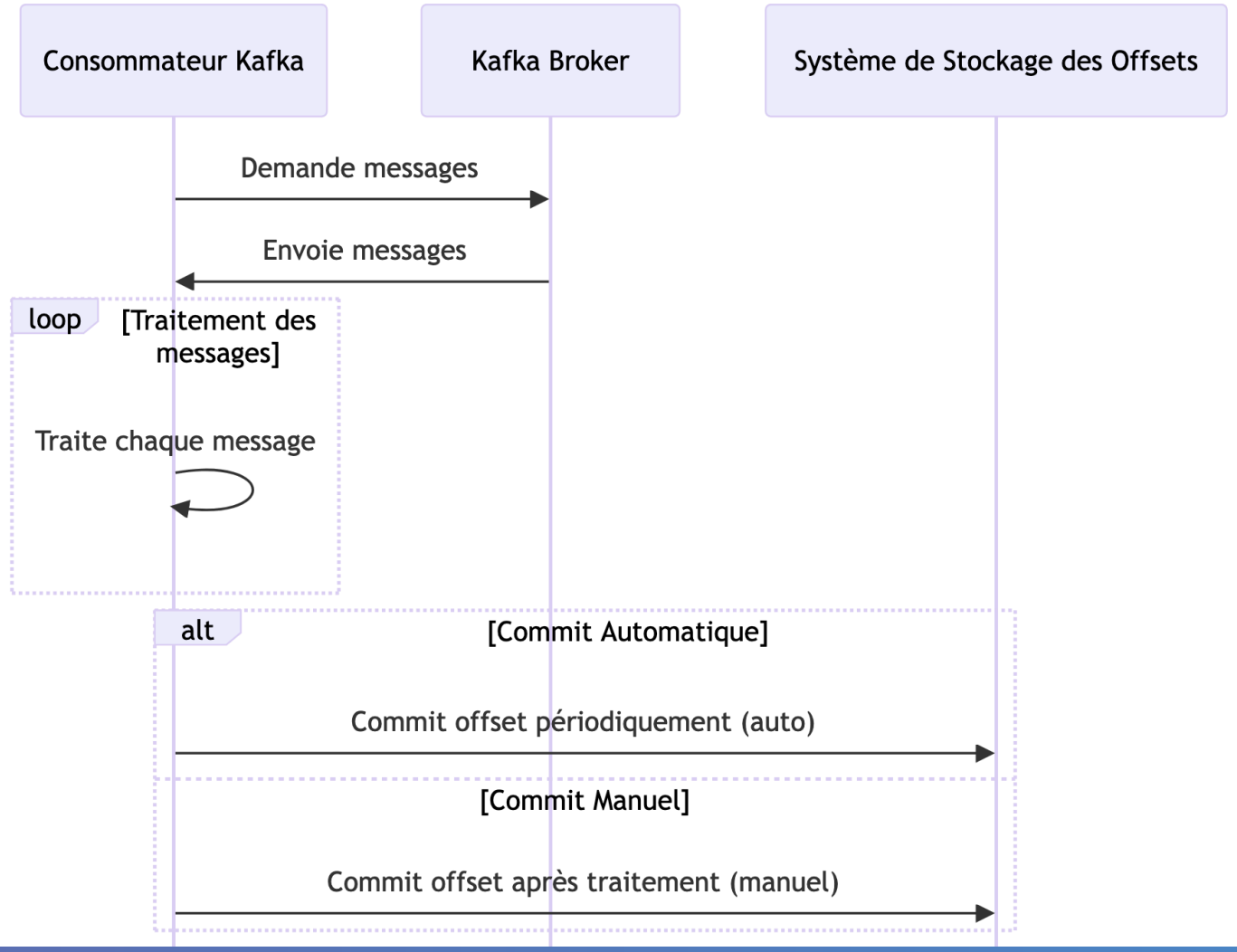
1. **Assurer la fiabilité** : En cas de panne du consommateur ou de redémarrage du système, Kafka sait à partir de quel message recommencer la lecture, grâce aux offsets committés. Cela garantit que les messages ne sont pas perdus ou consommés en double.
2. **Contrôle de la consommation** : Les consommateurs peuvent gérer la vitesse à laquelle ils consomment les messages et s'assurer qu'ils ne procèdent au commit que lorsque le traitement des messages est terminé.

Développer un consumer pour s’abonner à un topic kafka

Kafka offre deux modes principaux de commit d'offsets :

1. **Commit Automatique** :
   - **Configuration** : Cela est contrôlé par la propriété de configuration `enable.auto.commit`. Si elle est définie sur `true`, Kafka committera automatiquement les offsets à intervalles réguliers spécifiés par la propriété `auto.commit.interval.ms`.
   - **Utilisation** : Ce mode est plus simple à utiliser mais peut mener à des situations où des messages sont considérés comme consommés alors qu'ils ne l'ont pas été correctement traités en cas de panne juste avant un traitement mais après un commit.
2. **Commit Manuel** :
   - **Contrôle** : Le consommateur appelle explicitement une méthode pour committer l'offset après avoir traité les messages. Cela peut être fait de manière synchrone (attendre la réponse de Kafka confirmant le commit) ou asynchrone (ne pas attendre la réponse).
   - **Utilisation** : Ce mode offre un meilleur contrôle et est généralement utilisé dans les applications où la précision de la consommation et la garantie du traitement des messages sont critiques.

Développer un consumer pour s'abonner à un topic kafka



Exercice 3

Configurer des consommateurs Kafka pour traiter les notifications de manière efficace en fonction du type de notification et de la partition à partir de laquelle ils consomment.

1. Groupes de Consommateurs Spécialisés

- Configurez différents groupes de consommateurs pour différents types de notifications. Par exemple, un groupe peut être dédié aux notifications critiques (“nouveaux messages”) et un autre aux notifications moins critiques (“likes”).
- Assurez-vous que chaque groupe est optimisé pour traiter le type de notifications qui lui est attribué.

2. Configuration des Consommateurs pour Gérer Différentes Partitions

- Chaque groupe de consommateurs doit être configuré pour consommer des partitions spécifiques, conformément à la stratégie de partitionnement définie précédemment.
- Utilisez la méthode assign pour un contrôle précis des partitions ou subscribe pour une gestion plus dynamique et automatisée avec des rééquilibrages.

3. Traitement Basé sur le Type de Notification

- Implémentez une logique dans les consommateurs pour traiter différemment les messages en fonction de leur type. Les notifications critiques peuvent nécessiter des actions immédiates, tandis que les autres peuvent être traitées de manière plus asynchrone.

Kafka Connect

Kafka Connect est un projet open-source qui fournit un cadre scalable et fiable pour intégrer Kafka avec des systèmes externes. Il simplifie le processus de création et de gestion des pipelines de données entre Kafka et diverses sources ou destinations de données, permettant une intégration de données et un traitement de flux transparents. Kafka Connect sert d'outil puissant pour les développeurs, les ingénieurs de données et les architectes qui doivent gérer efficacement les tâches d'ingestion et de réplication de données.

- **Intégration unifiée des systèmes de données en streaming et par lots** : Kafka Connect est une solution idéale pour faire le pont entre les systèmes de données en streaming et par lots. Il permet aux développeurs de construire des pipelines de données intégrant Kafka avec d'autres systèmes de données, tels que les bases de données relationnelles, les bases de données NoSQL, les magasins d'objets et les systèmes de fichiers.
- **Un cadre commun pour les connecteurs Kafka** : Kafka Connect standardise l'intégration d'autres systèmes de données avec Kafka, simplifiant le développement, le déploiement et la gestion des connecteurs.
- **Extensibilité** : Kafka Connect offre une architecture de plugin qui permet aux développeurs d'étendre ses fonctionnalités en construisant des connecteurs personnalisés. Cette extensibilité permet une intégration transparente avec de nouvelles sources ou destinations de données, permettant aux organisations de s'adapter et d'évoluer dans leurs pipelines de données à mesure que leurs besoins changent.
- **Facilité de gestion** : Nous pouvons soumettre et gérer les connecteurs pour le cluster Kafka Connect via une API REST facile à utiliser.
- **Modes distribué et autonome** : Nous pouvons mettre à l'échelle jusqu'à de grands clusters (avec le mode distribué) ou réduire à des déploiements de développement, de test ou de petite production (avec le mode autonome).

Kafka Connect

Kafka Connect prend en charge deux modes d'exécution :

- **Mode autonome (processus unique)**
- **Mode distribué**

Dans le mode autonome, tout le travail est effectué dans un seul processus. Cette configuration est plus simple à configurer et à démarrer et peut être utile dans des situations où un seul travailleur est applicable (par exemple, la collecte de fichiers journaux), mais elle ne bénéficie pas de certaines fonctionnalités de Kafka Connect, telles que la tolérance aux pannes.
Dans le mode distribué, Kafka Connect fonctionne comme un système distribué avec plusieurs instances de travailleurs exécutant sur différents nœuds. Il gère l'équilibrage automatique du travail, nous permet de mettre à l'échelle dynamiquement et offre une tolérance aux pannes à la fois dans les tâches actives et pour la configuration des données de compensation engagées. En mode distribué, Kafka Connect stocke les décalages, les configurations et les statuts des tâches dans des topics Kafka.

Kafka Connect

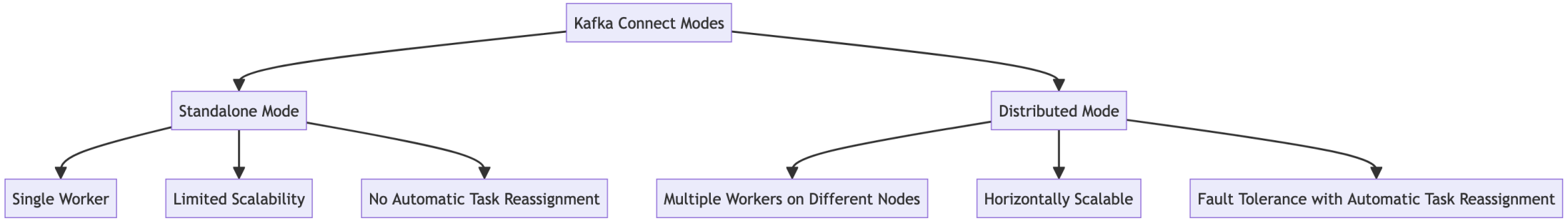
Caractéristique	Mode Distribué	Mode Autonome
Déploiement	Plusieurs travailleurs sur différents nœuds	Un seul travailleur sur une seule machine
Scalabilité	Scalable horizontalement	Scalabilité limitée
Tolérance aux pannes	Tolérant aux pannes avec réaffectation automatique des tâches	Pas de réaffectation automatique des tâches
Équilibrage de charge	Charge de travail automatiquement équilibrée parmi les travailleurs	Pas d'équilibrage de charge
Cas d'utilisation	Déploiements à grande échelle, volumes de données élevés	Développement, tests, déploiements à petite échelle

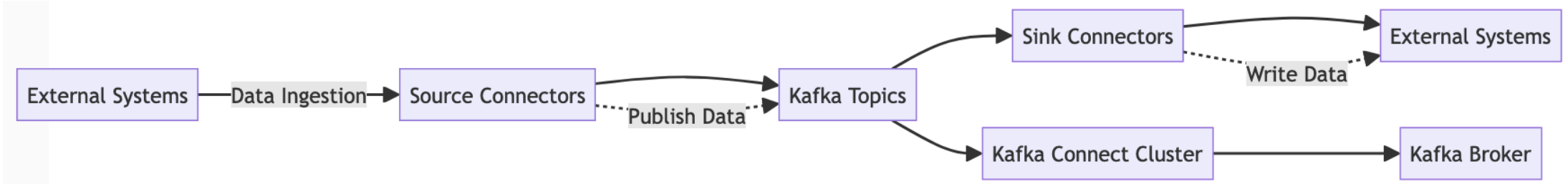
Kafka Connect

Les connecteurs Kafka Connect permettent l'intégration entre Apache Kafka et des systèmes externes pour l'ingestion et la réplication de données. Ce sont des modules enfichables qui définissent la logique et la configuration nécessaires pour lire les données de ou les écrire vers des sources ou des puits de données spécifiques. En utilisant des connecteurs, les développeurs, les ingénieurs de données et les architectes peuvent construire et gérer efficacement des pipelines de données qui interagissent avec diverses sources telles que des bases de données, des files d'attente de messages, des fichiers journaux et des plateformes de médias sociaux.

Les connecteurs gèrent les subtilités de l'extraction des données de ces sources ou de leur livraison, en abstrayant les complexités de l'intégration avec différents systèmes. Cette abstraction permet aux utilisateurs de se concentrer sur la configuration du pipeline et d'exploiter la puissance de la plateforme de messagerie distribuée de Kafka, plutôt que de traiter les spécificités de chaque système externe.

Kafka Connect





Exercice 4: Intégration et Gestion des Connecteurs Kafka dans Spring Boot

Intégrer et gérer les connecteurs Kafka Connect pour synchroniser les données entre les systèmes de notifications et les plateformes d'analyse ou de stockage externe, en utilisant Spring Boot pour le contrôle et l'orchestration.

1. Configuration des Connecteurs Kafka dans Spring Boot

- Configurez des connecteurs source pour extraire les notifications à partir de systèmes de bases de données.
- Configurez des connecteurs sink pour envoyer des notifications traitées vers des systèmes comme Elasticsearch pour l'analyse, ou vers des systèmes de stockage comme Amazon S3.

2. Développement de l'API pour Gérer les Connecteurs

- Créez une API RESTful dans Spring Boot pour ajouter, modifier, et supprimer des configurations de connecteurs en direct.

Custom Connect

Un **connecteur personnalisé** est spécifiquement développé pour répondre à des besoins qui ne sont pas couverts par les connecteurs disponibles dans la communauté ou ceux fournis par des tiers. Il peut être nécessaire de développer un connecteur personnalisé quand :

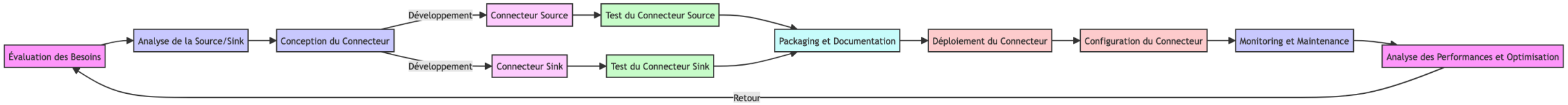
- Aucun connecteur existant ne prend en charge le système source ou sink que vous souhaitez intégrer.
- Les exigences spécifiques de traitement des données, de formatage, ou de performances ne sont pas satisfaites par les connecteurs existants.
- Des intégrations spécifiques d'entreprise ou des considérations de sécurité nécessitent une solution personnalisée.

1. **Évaluation des Besoins** :
    - Identifier la nécessité d'un connecteur personnalisé.
    - Analyser les systèmes source ou sink avec lesquels Kafka doit intégrer.
    - Définir les exigences précises en matière de données, de sécurité, de performance, etc.
2. **Préparation de l'Environnement de Développement** :
    - Installer Java, Kafka et tout autre outil nécessaire.
    - Configurer un environnement de développement avec les outils appropriés (IDE, Maven/Gradle, gestionnaires de version).

Custom Connect

3. **Développement du Connecteur** :
   - **Pour un connecteur source** : Développer une classe qui étend `SourceConnector` pour la configuration initiale et une classe `SourceTask` pour la récupération et le traitement des données.
   - **Pour un connecteur sink** : Développer une classe qui étend `SinkConnector` pour la configuration initiale et une classe `SinkTask` pour le traitement et le chargement des données dans le système cible.
4. **Test du Connecteur** :
   - Écrire des tests unitaires et d'intégration pour s'assurer que le connecteur fonctionne comme prévu dans divers scénarios.
   - Tester le connecteur dans un environnement qui simule la production autant que possible pour détecter les problèmes de performance ou de compatibilité.
5. **Déploiement et Maintenance** :
   - Déployer le connecteur dans un environnement Kafka Connect.
   - Surveiller les performances et la stabilité du connecteur.
   - Mettre à jour le connecteur selon les besoins pour corriger les bugs ou améliorer les fonctionnalités.

Custom Connect



Exercice 5

Développer et déployer deux types de connecteurs Kafka Connect personnalisés :

1. Un **connecteur source** qui lit des données depuis un fichier texte et les publie dans un topic Kafka spécifique.
2. Un **connecteur sink** qui lit des messages depuis un topic Kafka et les écrit dans un fichier texte de sortie.

Vous travaillez pour une entreprise qui gère de grandes quantités de données log générées par différentes applications. Ces logs sont initialement écrits dans des fichiers texte. Votre tâche est de faciliter l'analyse en temps réel de ces logs en les intégrant dans un système centralisé basé sur Kafka. De plus, pour des besoins de conformité, les données traitées doivent également être exportées et archivées sous forme de fichiers texte.

1. Connecteur Source

1. **Création du Connecteur Source** :
   - Développez un connecteur source Kafka Connect qui surveille un fichier texte pour de nouvelles lignes ajoutées.
   - Configurez le connecteur pour lire le fichier ligne par ligne et publier chaque ligne comme un message dans un topic Kafka.

Exercice 5

2. Connecteur Sink

1. **Création du Connecteur Sink** :
   - Développez un connecteur sink Kafka Connect qui abonne à un topic Kafka spécifique.
   - Configurez le connecteur pour écrire chaque message reçu dans un fichier texte de sortie, en ajoutant une nouvelle ligne pour chaque message.

Schema registry

1. **Qu'est-ce que Schema Registry ?**
   - **Définition :** Schema Registry est un outil centralisé pour enregistrer et gérer les schémas des messages Kafka. Il permet aux applications de s'assurer que les messages échangés via Kafka respectent des structures de données définies et compatibles.
   - **Rôles principaux :** Schema Registry sert à enregistrer, versionner et fournir des schémas aux clients, ce qui aide à valider la structure des messages produits et consommés.
2. **Pourquoi utiliser Schema Registry ?**
   - **Validation des messages :** Schema Registry assure que tous les messages publiés sur un topic Kafka sont conformes au schéma enregistré, ce qui aide à éviter les erreurs de données incompatibles.
   - **Gestion de l'évolution des schémas :** Schema Registry supporte plusieurs politiques de compatibilité (BACKWARD, FORWARD, FULL) qui aident à gérer l'évolution des schémas sans interrompre les services existants.
   - **Centralisation :** Tous les schémas sont stockés dans un emplacement centralisé, facilitant l'accès et la réutilisation des schémas par diverses applications qui produisent ou consomment des données via Kafka.

Schema registry

3. **Configuration des Producers avec Avro**:
   - **Sérialisation :** Utiliser `KafkaAvroSerializer` pour sérialiser les données. Cela inclut l'ajout du schéma Avro au Schema Registry si ce n'est pas déjà fait.
4. **Configuration des Consumers avec Avro**:
   - **Désérialisation :** Utiliser `KafkaAvroDeserializer` pour convertir les données sérialisées en objets Java en utilisant le schéma récupéré depuis Schema Registry.
5. **Utilisation de JSON avec Schema Registry**:
   - **Sérialisation et Désérialisation :** Bien que moins courant que Avro, JSON peut aussi être utilisé avec des schémas via `KafkaJsonSchemaSerializer` et `KafkaJsonSchemaDeserializer`.
   - **Avantages de JSON :** Plus lisible par l'homme, mais généralement moins efficace en termes de taille et de performance comparé à Avro.

Exercice 6

Développer une application Kafka pour capturer et traiter des événements de navigation web, où chaque événement contient des informations sur les actions des utilisateurs sur un site web. Vous utiliserez Schema Registry pour assurer la compatibilité des messages au fur et à mesure de l'évolution des schémas.

1. **Définir le Schéma Avro**
   - Créez un schéma Avro pour un message d'événement de navigation web. Le message doit contenir les champs suivants :
     - `eventId`: string (identifiant unique de l'événement)
     - `userId`: int (identifiant de l'utilisateur)
     - `url`: string (URL visitée)
     - `timestamp`: long (horodatage de l'événement, en millisecondes depuis l'époque Unix)
     - `duration`: int (durée passée sur la page en secondes, facultatif)
2. **Configurer Schema Registry**
   - Assurez-vous que Schema Registry est configuré et fonctionnel. Il devrait être en mesure de communiquer avec votre cluster Kafka.

Exercice 6

3. **Développement de l'Application Kafka**
   - **Producteur d'Événements Web**:
     - Intégrez un producteur Kafka dans une application Spring Boot qui capte les événements de navigation à partir d'une source hypothétique (par exemple, une simulation ou une API REST) et les publie sur un topic Kafka.
     - Configurez le producteur pour utiliser `KafkaAvroSerializer` pour sérialiser les données avant de les envoyer à Kafka.
   - **Consommateur d'Événements Web**:
     - Créez un consommateur Kafka dans la même application Spring Boot ou dans un service distinct. Ce consommateur doit s'abonner au topic Kafka et traiter les événements réceptionnés, par exemple en calculant le temps total passé sur des URLs spécifiques ou le nombre de visites par utilisateur.
     - Configurez le consommateur pour utiliser `KafkaAvroDeserializer` pour désérialiser les messages reçus de Kafka.

Streams

Streaming de données

- Le streaming de données fait référence à la transmission continue de données en temps réel, où les données sont produites et consommées simultanément. Dans le contexte de Kafka, cela implique l'envoi de messages (ou enregistrements) en continu à travers un système distribué.
- Les applications de streaming de données utilisent souvent Kafka pour capturer des flux de données en continu provenant de diverses sources (comme des logs de serveurs, des flux de transactions, etc.), et pour les rendre disponibles à des systèmes de traitement en temps réel.

Pipeline de données

- Un pipeline de données dans Kafka est une série d'étapes organisées pour le traitement des données en continu. Ces étapes peuvent inclure la collecte de données, leur transformation (comme l'agrégation ou le filtrage), et finalement leur stockage ou leur visualisation.
- Avec Spring, on utilise souvent Spring Kafka ou Spring Cloud Stream pour gérer ces pipelines. Ces bibliothèques fournissent des abstractions et des outils pour intégrer Kafka dans des applications Spring, facilitant la configuration et la gestion des flux de données.

Streams

Comparatifs streams vs topics

- **Topics**
  - Un topic dans Kafka est une catégorie ou un canal où les messages sont publiés. Les topics sont divisés en partitions pour permettre un stockage et un accès distribués et parallélisés des messages.
  - Les topics sont le fondement de stockage des données dans Kafka. Chaque message dans un topic est identifié par un offset unique, et les données sont conservées dans le topic selon une politique de rétention configurable.
- **Streams**
  - Kafka Streams est une bibliothèque de traitement de flux de données qui peut être utilisée pour développer des applications de traitement de données en temps réel. Un "stream" peut être vu comme une séquence ininterrompue d'enregistrements (messages) qui sont continuellement traités.
- Kafka Streams utilise les topics de Kafka pour l'entrée et la sortie des données, mais offre des fonctionnalités supplémentaires pour le traitement complexe des données, comme la gestion des états, les fenêtres temporelles, et les jointures entre flux.

Streams

Différences clés

- **Nature des données**: Un topic est simplement un mécanisme de stockage des données, alors qu'un stream représente un flux continu de données traitées.
- **Traitement des données**: Les topics ne fournissent pas de mécanismes de traitement des données; ils servent seulement à stocker et à diffuser des messages. En revanche, Kafka Streams permet de transformer les données en cours de flux à travers des opérations complexes.
- **Utilisation avec Spring**: Spring Kafka permet d'intégrer Kafka pour la publication et la consommation de messages à partir de topics, tandis que Spring Cloud Stream offre des abstractions plus élevées pour créer des applications orientées streaming qui utilisent Kafka Streams.

Stream

1. **StreamsBuilder**
   - **Description** : Utilisé pour construire la topologie de flux de données.
   - **Méthodes courantes** :
     - `stream(String topic)` : Crée un `KStream` pour lire les données à partir d'un topic Kafka.

```
KStream<String, String> stream = builder.stream("topic-name");
```

     - `table(String topic)` : Crée un `KTable` pour lire les données à partir d'un topic Kafka en tant que table mise à jour continuellement.

```
KTable<String, String> table = builder.table("table-topic");
```

     - `globalTable(String topic)` : Crée un `GlobalKTable` pour lire les données d'un topic Kafka en tant que table globale.

```
GlobalKTable<String, String> globalTable = builder.globalTable("global-topic");
```

Streams

2. **KStream**

- **Description** : Représente un flux de données clé-valeur (key-value) sans état.
- **Méthodes courantes** :
- `filter(Predicate<? super K, ? super V> predicate)` : Filtre les enregistrements en fonction d'un prédicat.

```
KStream<String, String> filteredStream = stream.filter((key, value) -> value.contains("important"));
```

- `map(KeyValueMapper<? super K, ? super V, ? extends KeyValue<? extends K1, ? extends V1>> mapper)` : Applique une fonction de transformation aux paires clé-valeur.

```
KStream<String, Integer> mappedStream = stream.map((key, value) -> new KeyValue<>(key, value.length()));
```

- `flatMap(KeyValueMapper<? super K, ? super V, ? extends Iterable<? extends KeyValue<? extends K1, ? extends V1>>> mapper)` : Applique une fonction de transformation qui retourne un `Iterable` de nouvelles paires clé-valeur.

```
KStream<String, String> flatMappedStream = stream.flatMap((key, value) -> {
    List<KeyValue<String, String>> result = new ArrayList<>();
    for (String word : value.split("\\s+")) {
        result.add(new KeyValue<>(key, word));
    }
    return result;
});
```

Streams

- `groupByKey()` : Regroupe les enregistrements par clé.

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

- `to(String topic)` : Envoie les enregistrements vers un topic Kafka.

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

3. **KTable**

- **Description** : Représente une table mise à jour continuellement.
- **Méthodes courantes** :
- `toStream()` : Convertit la `KTable` en `KStream`.

```
KStream<String, Long> streamFromTable = table.toStream();
```

- `groupBy(KeyValueMapper<? super K, ? super V, KeyValue<? extends K1, ? extends V1>> selector)` : Regroupe les enregistrements en fonction d'un sélecteur.

```
KGroupedTable<String, String> groupedTable = table.groupBy((key, value) -> new KeyValue<>(value, key));
```

- `count()` : Compte les enregistrements pour chaque clé.

```
KTable<String, Long> countTable = groupedTable.count();
```

Streams

4. **Materialized**

- **Description** : Utilisé pour matérialiser (stocker) les résultats intermédiaires des opérations d'agrégation.
- **Méthodes courantes** :
- `as(String storeName)` : Spécifie le nom du magasin d'état où les résultats seront stockés.

```
Materialized<String, Long, KeyValueStore<Bytes, byte[]>> materialized = Materialized.as("store-name");
```

- `withKeySerde(Serde<K> keySerde)` : Spécifie le serde pour les clés.

```
materialized.withKeySerde(Serdes.String());
```

- `withValueSerde(Serde<V> valueSerde)` : Spécifie le serde pour les valeurs.

```
materialized.withValueSerde(Serdes.Long());
```

Streams

5. **TimeWindows**

- **Description** : Définit les fenêtres de temps utilisées pour les opérations d'agrégation basées sur le temps.
- **Méthodes courantes** :
- `of(Duration size)` : Spécifie la taille de la fenêtre.

```
TimeWindows windows = TimeWindows.of(Duration.ofMinutes(1));
```

- `advanceBy(Duration advance)` : Spécifie l'intervalle d'avancement pour les fenêtres glissantes.

```
TimeWindows slidingWindows = TimeWindows.of(Duration.ofMinutes(1)).advanceBy(Duration.ofSeconds(30));
```

6. **Produced**

- **Description** : Utilisé pour configurer la sérialisation des clés et des valeurs lorsqu'un flux de données est écrit dans un topic Kafka.
- **Méthodes courantes** :
- `with(Serde<K> keySerde, Serde<V> valueSerde)` : Spécifie les serdes pour les clés et les valeurs.

```
Produced<String, Long> produced = Produced.with(Serdes.String(), Serdes.Long());
```

Streams

7. **Serdes**

- **Description** : Fournit des implémentations pour les sérialiseurs/désérialiseurs (serdes) courants.
- **Méthodes courantes** :
- `String()` : Retourne une instance de serde pour les chaînes de caractères.

```
Serdes.StringSerde stringSerde = Serdes.String();
```

- `Long()` : Retourne une instance de serde pour les valeurs de type `Long`.

```
Serdes.LongSerde longSerde = Serdes.Long();
```

8. **KeyValue**

- **Description** : Classe utilitaire représentant une paire clé-valeur.
- **Méthodes courantes** :
- `KeyValue(K key, V value)` : Crée une nouvelle instance de `KeyValue`.

```
KeyValue<String, Integer> keyValue = new KeyValue<>("key", 1);
```

Streams

9. **Windowed**

- **Description** : Encapsule une clé avec des informations sur la fenêtre de temps.
- **Méthodes courantes** :
- `key()` : Retourne la clé encapsulée.

```
K key = windowed.key();
```

- `window()` : Retourne la fenêtre de temps encapsulée.

```
Window window = windowed.window();
```

10. **Duration**

- **Description** : Classe de l'API Java Time qui représente une quantité de temps.
- **Méthodes courantes** :
- `ofMinutes(long minutes)` : Crée une durée représentant un nombre de minutes.

```
Duration duration = Duration.ofMinutes(5);
```

- `ofSeconds(long seconds)` : Crée une durée représentant un nombre de secondes.

```
Duration duration = Duration.ofSeconds(30);
```

Exercice 7

Vous travaillez pour une banque qui veut implémenter un système de détection de fraudes en temps réel. Les transactions bancaires sont envoyées à un topic Kafka et doivent être analysées pour détecter des activités suspectes comme des transactions répétées à partir de la même carte bancaire dans un court laps de temps. Vous allez utiliser Kafka Streams et Spring pour développer cette application.

1. Créer un contrôleur Spring Boot pour simuler l'envoi de transactions bancaires à un topic Kafka.
2. Utiliser Kafka Streams pour analyser les transactions en temps réel.
3. Détecter des fraudes potentielles basées sur des critères spécifiques.
4. Envoyer les alertes de fraude détectée à un autre topic Kafka.

Concepts et architecture de KSQLDB

KSQLDB est une plateforme de traitement de données en streaming qui permet de traiter des flux de données en temps réel avec une syntaxe proche du SQL standard. Elle est conçue pour fonctionner avec Apache Kafka, s'appuyant sur les capacités de Kafka pour gérer des flux de données massifs et continus. Voici une explication détaillée des concepts clés et de l'architecture de KSQLDB.

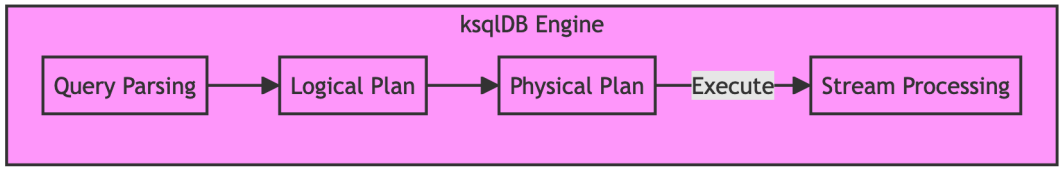
1. **Streams et Tables**:
   - **Streams**: Un stream dans KSQLDB est une abstraction d'un topic Kafka. Il représente un flux de données immuable, où chaque message est considéré comme un enregistrement d'événement. Un stream peut être utilisé pour capturer des événements tels que les clics sur un site web, les transactions financières, ou les mesures de capteurs.
   - **Tables**: Une table dans KSQLDB est une vue mutable des données et représente l'état actuel des enregistrements. Contrairement aux streams, où chaque message est un nouvel événement, les tables maintiennent un enregistrement de l'état le plus récent pour chaque clé.
2. **Queries en Temps Réel**:
   - KSQLDB permet de formuler des requêtes en continu sur les données qui passent à travers les streams et les tables, facilitant ainsi des analyses complexes et des agrégations en temps réel.
3. **Traitement Basé sur le Temps**:
   - KSQLDB supporte des fenêtrages temporels pour les opérations d'agrégation, permettant aux utilisateurs de créer des agrégations sur des fenêtres de temps spécifiques, telles que des fenêtres glissantes ou tumbling.

Concepts et architecture de KSQLDB

KSQLDB fonctionne sur le modèle client-serveur et est composé de plusieurs composants clés:

1. **Serveur KSQLDB**:
   - Le serveur exécute le moteur de traitement de flux et est responsable de l'exécution des requêtes KSQLDB. Il communique avec le cluster Kafka pour lire les messages des topics et pour écrire les résultats des requêtes.
2. **Client KSQLDB**:
   - Les utilisateurs interagissent avec KSQLDB principalement via le client KSQLDB, qui peut être une interface de ligne de commande ou une interface utilisateur graphique. Le client permet aux utilisateurs de soumettre des requêtes, de créer des streams et des tables, et de configurer le système.
3. **API REST**:
   - KSQLDB offre une API REST qui permet aux développeurs d'intégrer la fonctionnalité KSQLDB dans d'autres applications et services. Cette API peut être utilisée pour exécuter des requêtes, gérer des streams/tables, et recevoir des résultats de requêtes.
4. **Moteur de Streaming Kafka Streams**:
   - KSQLDB s'appuie sur Kafka Streams pour le traitement des données en streaming. Kafka Streams est un client de la bibliothèque de traitement de streams pour Kafka qui permet de construire des applications de traitement de streams robustes et facilement scalable.

Concepts et architecture de KSQLDB



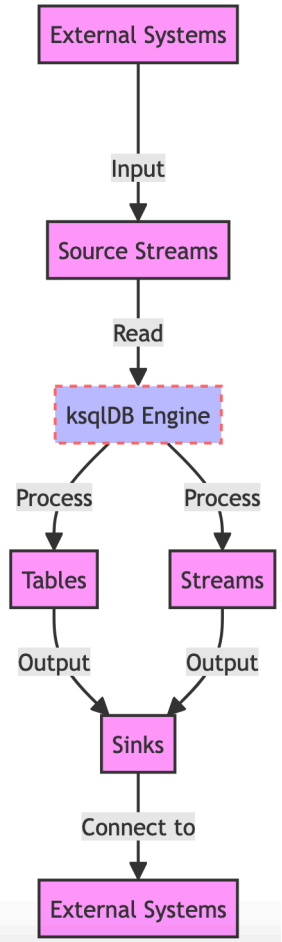


Table et Stream - Stream

Un **stream** dans KSQLDB est une abstraction sur un topic Kafka qui représente un flux de données immuables. Les streams permettent de traiter des données en continu, comme ils arrivent.

```
CREATE STREAM stream_name (
    column_name data_type,
) WITH (kafka_topic='topic_name', value_format='format', ...);
```

- **`kafka_topic`**: Nom du topic Kafka sous-jacent.
- **`value_format`**: Format des données dans le topic (par exemple, `JSON`, `AVRO`, `PROTOBUF`, `DELIMITED`).
- **`key_format`**: Format de la clé dans le topic Kafka.
- **`timestamp`**: Colonne à utiliser comme timestamp. Si non spécifié, le timestamp du message Kafka est utilisé.
- **`timestamp_format`**: Format utilisé pour parser la colonne de timestamp si elle est en format de chaîne.
- **`partitions`** et **`replicas`**: Ces options permettent de spécifier le nombre de partitions et de réplicas pour le topic Kafka lors de la création du stream si le topic n'existe pas déjà.

```
CREATE STREAM users_stream (
    user_id INT KEY,
    first_name STRING,
    last_name STRING,
    email STRING
) WITH (
    kafka_topic='users',
    value_format='JSON',
    partitions=4
);
```

Table et Stream - Table

Une **table** dans KSQLDB représente un état mutable et est généralement utilisée pour modéliser des données qui changent dans le temps, où chaque clé a la dernière valeur connue.

Syntaxe de Base pour Créer une Table

```
CREATE TABLE table_name (
    column_name data_type PRIMARY KEY,
    ...
) WITH (kafka_topic='topic_name', value_format='format', ...);
```

Options Clés de Configuration

- **`kafka_topic`**: Similaire aux streams, c'est le topic Kafka sous-jacent.
- **`value_format`**: Format des données (par exemple, `JSON`, `AVRO`).
- **`key_format`**: Format de la clé, important pour les tables puisque les opérations de mise à jour s'appuient sur la clé.
- **`partitions`** et **`replicas`**: Spécifiez le nombre de partitions et de réplicas pour le topic.

Exemple de Création de Table

```
CREATE TABLE orders_table (
    order_id INT PRIMARY KEY,user_id INT,
    order_total DECIMAL(5, 2),
    order_date DATE) WITH (
    kafka_topic='orders',
    value_format='JSON',partitions=3
);
```

Table et Stream - Table

Distinctions Importantes

- **Streams vs Tables**: Un stream est une séquence de données immuables, où chaque record est traité indépendamment. Une table est une vue mutable de données, où chaque clé a la dernière valeur mise à jour.
- **Requêtes sur Streams et Tables**: Les streams sont généralement utilisés pour les requêtes qui nécessitent une analyse des événements en temps réel au fur et à mesure de leur arrivée, tandis que les tables sont utilisées pour des requêtes qui nécessitent un état actuel ou une vue agrégée des données.

Jointures, agrégations et fenêtres de temps et de taille

1. Jointures dans ksqlDB

Les jointures en ksqlDB permettent de combiner des données provenant de plusieurs flux (streams) ou tables basées sur des conditions spécifiques.

- **Inner Join** : Combine les lignes de deux flux ou tables où les conditions de jointure sont remplies.
- **Left Join** : Combine toutes les lignes du flux ou de la table de gauche avec les lignes correspondantes du flux ou de la table de droite. Si aucune correspondance n'est trouvée, les résultats de droite seront NULL.
- **Outer Join** : Combine toutes les lignes de deux flux ou tables, remplissant les résultats avec NULLs quand il n'y a pas de correspondance.

Exemple:

```
SELECT orders.order_id, customers.customer_name
FROM orders
JOIN customers ON orders.customer_id = customers.customer_id;
```

2. Agrégations dans ksqlDB

Les agrégations permettent de calculer des résumés de données, tels que des totaux, des moyennes, des comptages, etc. Les fonctions d'agrégation couramment utilisées incluent `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`, etc.

Exemple:

```
SELECT customer_id, COUNT(*) AS order_count
FROM orders
GROUP BY customer_id;
```

Jointures, agrégations et fenêtres de temps et de taille

3. Fenêtres de temps et de taille (Windowing)

Les fenêtres de temps et de taille permettent d'appliquer des agrégations sur des segments de données basés sur le temps ou sur un nombre spécifique d'événements. Cela est crucial pour l'analyse des données en temps réel.

- **Fenêtre Tumbling** : Segmente les flux en fenêtres de taille fixe, sans chevauchement.

```
SELECT COUNT(*)
FROM orders
WINDOW TUMBLING (SIZE 10 MINUTES)
GROUP BY product_id;
```

- **Fenêtre Hopping** : Fenêtre de taille fixe avec chevauchement, permettant une analyse glissante.

```
SELECT COUNT(*)
FROM orders
WINDOW HOPPING (SIZE 10 MINUTES, ADVANCE BY 5 MINUTES)
GROUP BY product_id;
```

- **Fenêtre Sliding** : Fenêtre glissante qui recalcule l'agrégation chaque fois qu'un nouvel événement arrive.

```
SELECT COUNT(*)
FROM orders
WINDOW SLIDING (SIZE 10 MINUTES)
GROUP BY product_id;
```

Jointures, agrégations et fenêtres de temps et de taille

- **Fenêtre Session** : Fenêtre basée sur l'activité, où une nouvelle fenêtre est créée lorsqu'il y a une période d'inactivité définie.

```
SELECT COUNT(*)
FROM orders
WINDOW SESSION (15 MINUTES)
GROUP BY product_id;
```

Exemple de script complet utilisant jointure, agrégation et fenêtres

```
-- Création des streams
CREATE STREAM orders (order_id INT, customer_id INT, product_id INT) WITH (KAFKA_TOPIC='orders_topic', VALUE_FORMAT='JSON');
CREATE STREAM customers (customer_id INT, customer_name STRING) WITH (KAFKA_TOPIC='customers_topic', VALUE_FORMAT='JSON');

-- Jointure des streams
CREATE STREAM enriched_orders AS
SELECT orders.order_id, orders.product_id, customers.customer_name
FROM orders
JOIN customers ON orders.customer_id = customers.customer_id;

-- Agrégation avec fenêtre de temps
CREATE TABLE product_order_count AS
SELECT product_id, COUNT(*) AS order_count
FROM enriched_orders
WINDOW TUMBLING (SIZE 10 MINUTES)
GROUP BY product_id;
```

Exercice 8

Vous travaillez pour une entreprise qui gère un réseau de dispositifs IoT (Internet of Things) déployés dans diverses localisations pour surveiller la température et l'humidité. Vous souhaitez analyser les données en temps réel pour détecter des anomalies et effectuer des agrégations sur les mesures.

1. Création des Streams :

- Créer deux streams : temperature_readings et humidity_readings.
- Insérer des données dans ces streams.

2. Création d'une Jointure :

- Joindre les deux streams pour créer un stream enrichi sensor_data.

3. Effectuer des Agrégations :

- Calculer la moyenne de la température et de l'humidité par zone sur une fenêtre de 5 minutes en utilisant des agrégations.

extension KSQLDB spécifique

Développer une extension ksqlDB spécifique implique la création de fonctions définies par l'utilisateur (UDF - User Defined Functions) ou d'agrégats définis par l'utilisateur (UDAF - User Defined Aggregate Functions) en Java. Ces extensions permettent d'ajouter des fonctionnalités personnalisées pour traiter des données en temps réel. Voici les étapes générales pour développer une extension ksqlDB.

Etude de cas

Contexte :
Une entreprise de logistique, LogiTrack, souhaite moderniser son système de suivi des livraisons. Actuellement, les informations sur les colis en transit sont collectées et traitées de manière asynchrone, ce qui cause des retards dans la mise à jour des statuts de livraison. L'objectif est de mettre en place une solution de streaming de données en temps réel avec Apache Kafka pour suivre les colis depuis leur départ jusqu'à leur livraison, en capturant les événements à chaque étape (scans, chargements, déchargements, etc.).

1. **Implémentation d'un Producteur Kafka**
   - Développez une application SpringBoot qui fonctionne comme un producteur Kafka.
   - Cette application doit envoyer des événements de suivi de colis (par exemple, `id_colis`, `timestamp_scan`, `lieu_scan`, `état_colis`) à un topic Kafka nommé `tracking-events`.
2. **Configuration d'un Cluster Kafka**
   - Configurez un cluster Kafka avec au moins 3 brokers.
   - Créez un topic Kafka nommé `tracking-events` avec 3 partitions et un facteur de réplication de 2.
   - Documentez le processus de configuration et les raisons des choix techniques effectués.
3. **Développement d'un Consommateur Kafka**
   - Développez une application SpringBoot qui fonctionne comme un consommateur Kafka.
   - Cette application doit s'abonner au topic `tracking-events`, lire les événements de suivi, et les stocker dans une base de données PostgreSQL.

Etude de cas

4. **Utilisation de Kafka Connect**
   - Configurez un connecteur Kafka pour intégrer les données de suivi provenant des scanners de codes-barres.
   - Le connecteur doit lire les fichiers CSV générés par les scanners et les envoyer au topic `tracking-events`.
   - Définissez et implémentez les transformations nécessaires pour convertir les données des fichiers CSV en format JSON avant de les envoyer à Kafka.
5. **Gestion des Schémas avec Schema Registry**
   - Implémentez des schémas Avro pour les messages de suivi de colis.
   - Enregistrez ces schémas dans le Schema Registry.
   - Mettez à jour le producteur et le consommateur pour utiliser les schémas Avro lors de l'envoi et de la réception des messages.
6. **Mise en Place d'un Pipeline de Streaming avec KSQLDB**
   - Configurez KSQLDB pour analyser les données de suivi en temps réel.
   - Créez un flux KSQLDB qui calcule et met à jour le statut global de chaque colis en fonction des événements de suivi.
   - Documentez les requêtes KSQLDB utilisées et leur impact sur les performances du système.

Etude de cas

7. **Développement d'un Connecteur Kafka Spécifique**
    - Développez un connecteur Kafka spécifique pour un système de suivi propriétaire utilisé par LogiTrack.
    - Le connecteur doit être capable de lire les données de l'API du système propriétaire et les envoyer au topic `tracking-events`.
    - Implémentez des transformations personnalisées pour adapter le format des données de l'API au format utilisé par Kafka.