ORSYS formation

Elitis CONSULTING

Apache KAFKA

Centraliser les flux en temps réel.

Tour de table des stagiaires.

- Nom & prénom
- Expériences
- Précisez vos attentes pour cette formation

Présentation du formateur - Didier Curvier MALEMBE.

- **Ingénieur en informatique en ESN**
  - > Capgemini Technology Services
    - > *Projet SNCF Connect*
    - > *Projet RM (Revenue Management) , etc…*
- **Consultant Freelance**
  - > Secteur public / TPE-PME
- **Formateur partenaire**
  - > ORSYS Formation
- **Co-founder ELITIS CONSULTING**

Les objectifs pédagogiques

À l’issue de la formation, le participant sera en mesure de :

- Comprendre les principes de communications inter applications
- Appréhender l'architecture de **Kafka**
- Mettre en œuvre **ksqlDB**
- Travailler en sécurité avec **Kafka**

Plan du cours.

I. L’évolution des systèmes informatiques et domaines d’utilisation de **Kafka**
II. Vue d’ensemble de **Kafka**
III. Architecture à haute disponibilité
IV. Vue d'ensemble de **ksqlDB** et écosystème
V. Utiliser **ksqlDB**
VI. **Kafka** Streams
VII. Sécurité avec **Kafka**
VIII. Les outils autour de **Kafka**

Quelques prérequis.

- Bases du Langage de programmation **JAVA**.
- Commandes linux.

L’évolution des systèmes informatiques et domaines d’utilisation de Kafka

Les architectures applicatives en entreprise

Le Cloud

D'après le **NIST** américain *(National Institute of Standards and Technology)* le cloud est : ***« un modèle permettant un accès facile et à la demande, via le réseau, à un pool partagé de ressources informatiques configurables (par exemple, réseaux, serveurs, stockage, applications et services) qui peuvent être rapidement mises à disposition des utilisateurs ou libérées avec un effort minimum d'administration de la part de l'entreprise ou du prestataire de service fournissant les dites ressources ».***

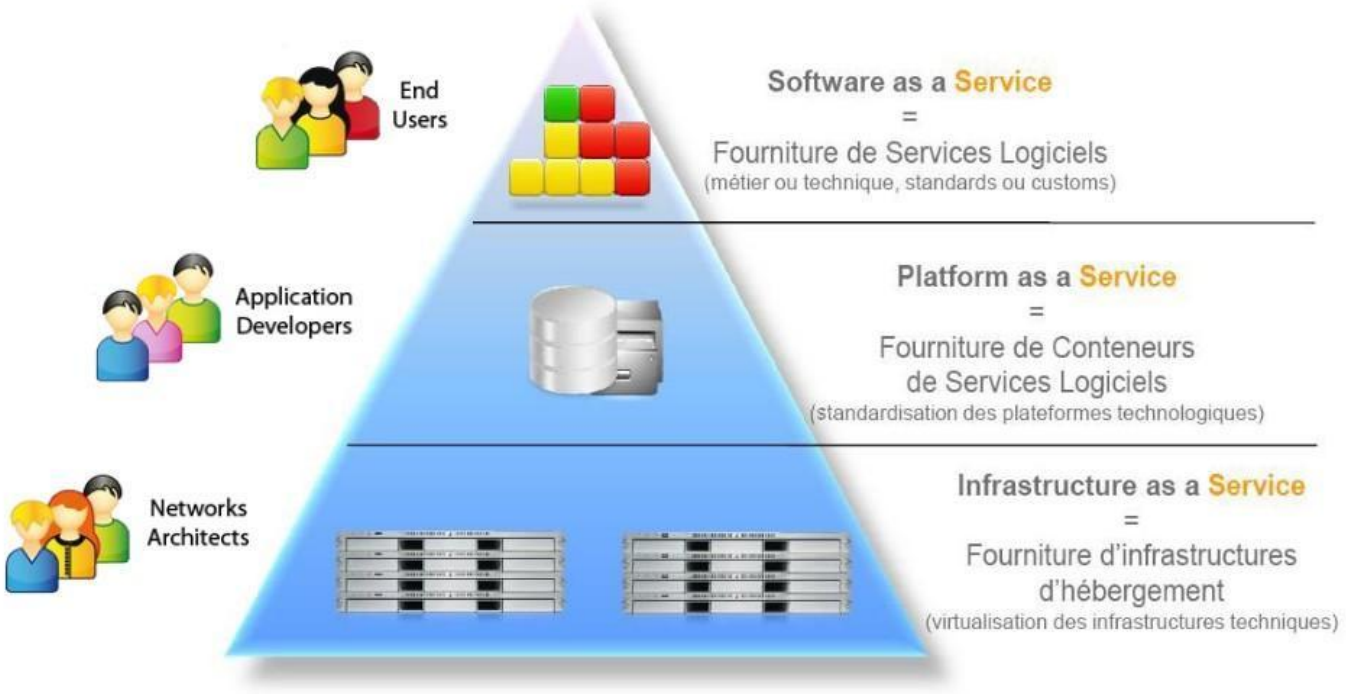
Besoin de l’entreprise	Infrastructure sur site	Cloud Computing
Usage exclusif du datacenter		
chiffrement des données hautement sécurisé		
Matériel personnalisable et systèmes spécialement conçus		
Capacité facile à augmenter et à réduire		

Infrastructure nécessitant des investissements réguliers et importants		
Paiement à l’usage, tarification basée sur l’utilisation		
Visibilité et gestion complètes de données		
Risque de temps d’arrêt quasi nul		
Sauvegardes et restaurations intégrées et automatisées		

05 raisons de choisir le Cloud.

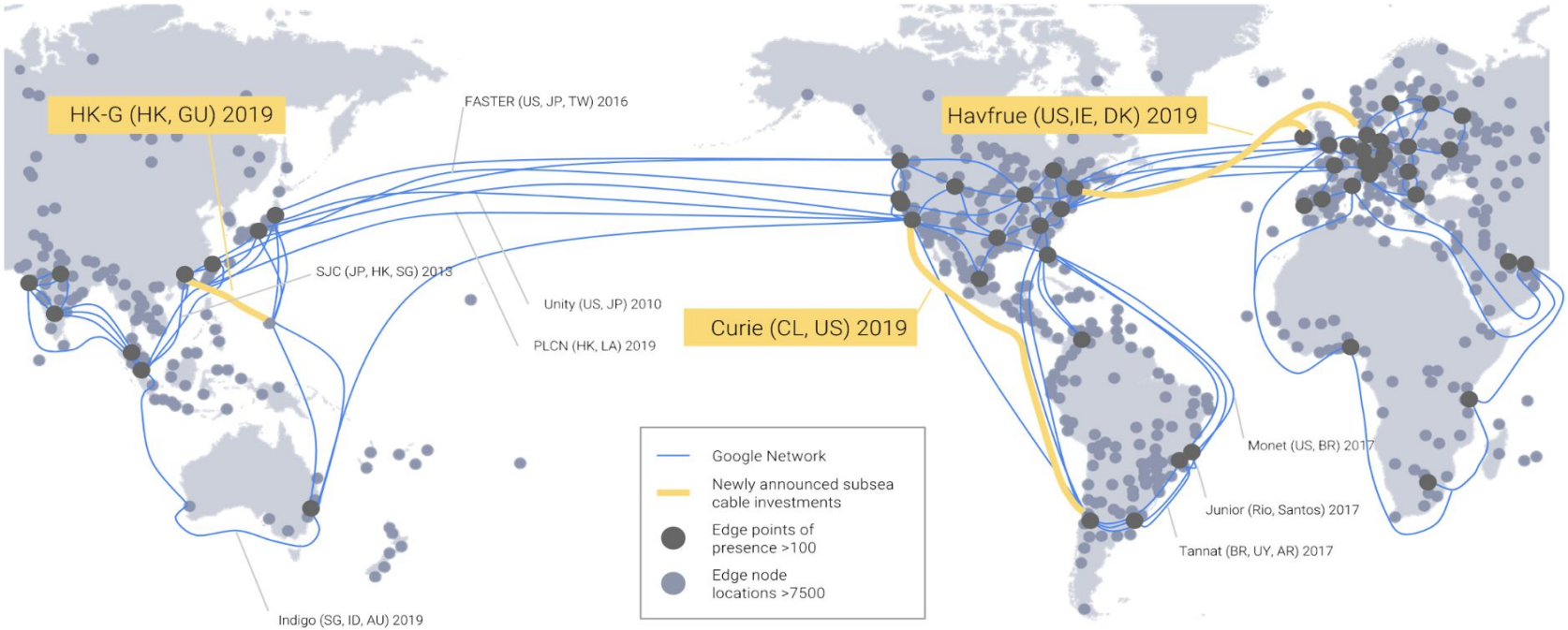
- La puissance de calcul
- Le cloud est moins cher
- La création de valeur
- La scalabilité
- La flexibilité

Les différents services cloud.

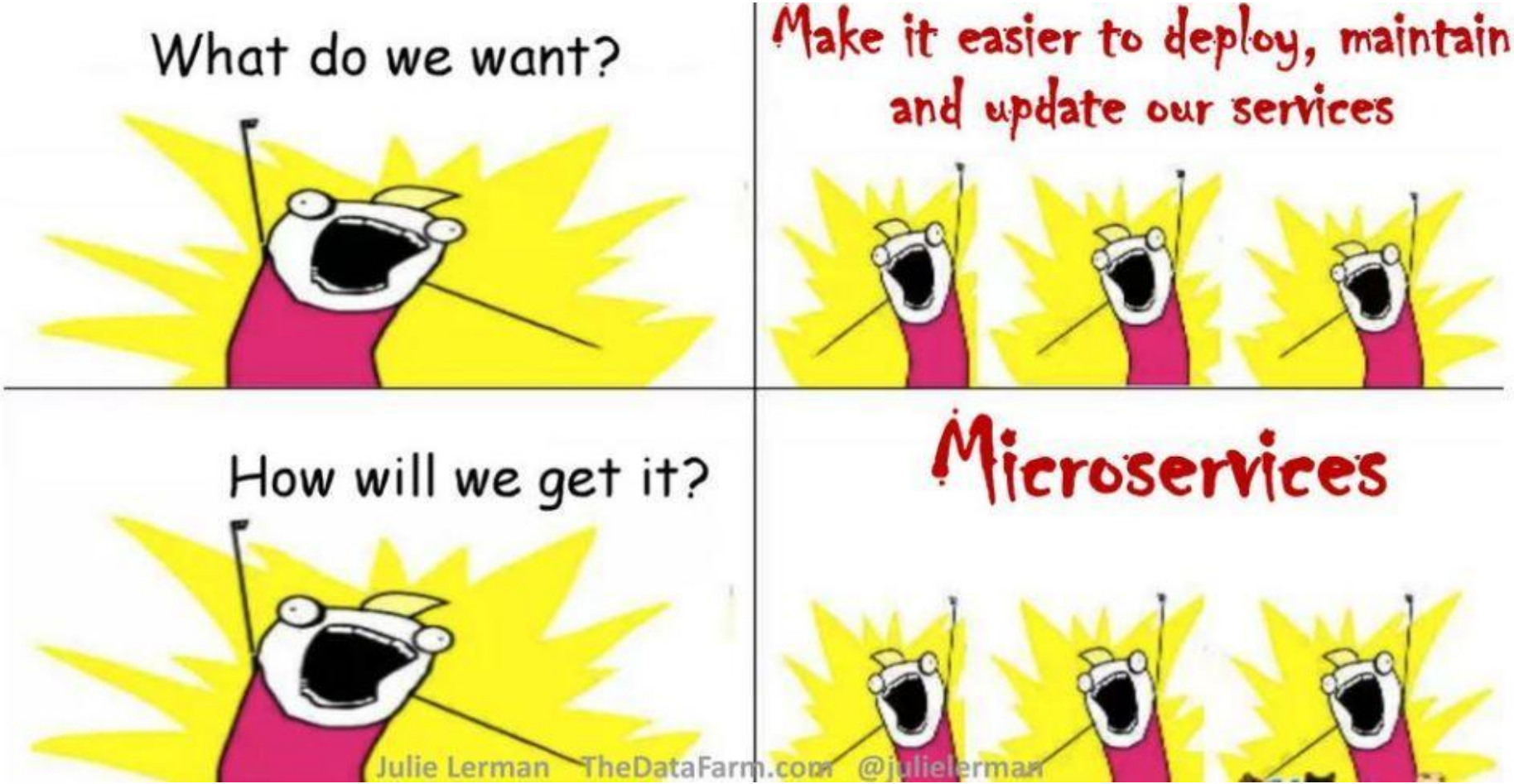


Google Network

The largest cloud network, comprised of more than 100 points of presence



Google Cloud Platform offers services for getting value from data


What do we want?
Make it easier to deploy, maintain and update our services
How will we get it?
Microservices
Julie Lerman TheDataFarm.com @julielerman

Architecture microservices

Définition

- Une architecture de microservices divise une app en une série de services déployables indépendamment qui communiquent via des **API**.

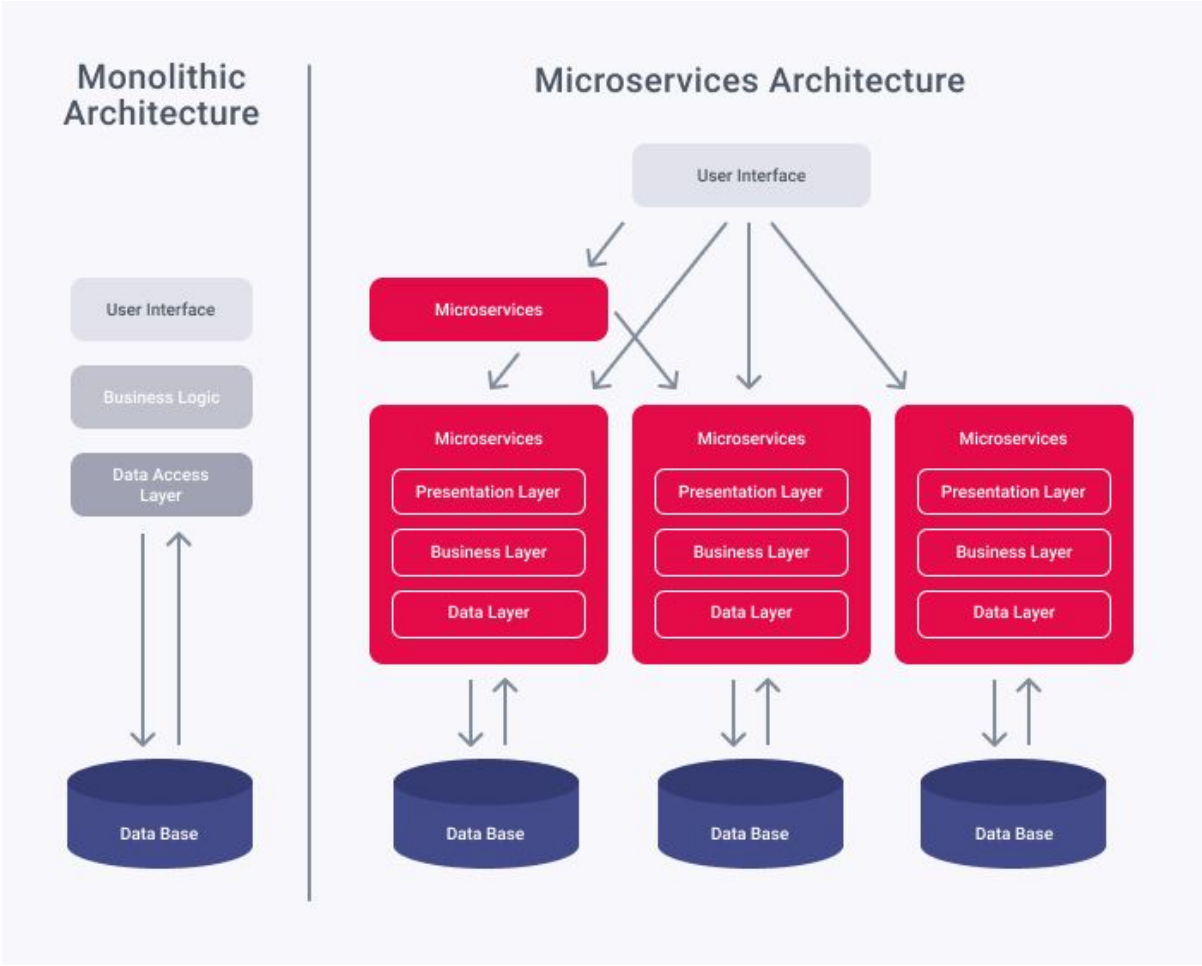
- Chaque service individuel peut être déployé et mis à l'échelle de manière indépendante.

Avantages

- Cycles de développement indépendants
- Builds et livraison simplifiés
- Configuration simplifiée
- Scalabilité horizontal
- Robustesse
- Équipe de travail minimale
- Evolutivité
- Fonctionnalité modulaire - modules indépendants
- Utilisation de conteneurs pour permettre un dev et un déploiement rapide

Inconvénients

- Identification et découpage des services et données
- Communication entre services
- Cohérence des données
- Event driven
- Automatisation
- Besoins de monitoring
- Problème de latence ou l'équilibrage de charge
- Complexité de la gestion d'un grand nombre de services
- Tests complexes sur le déploiement distribué

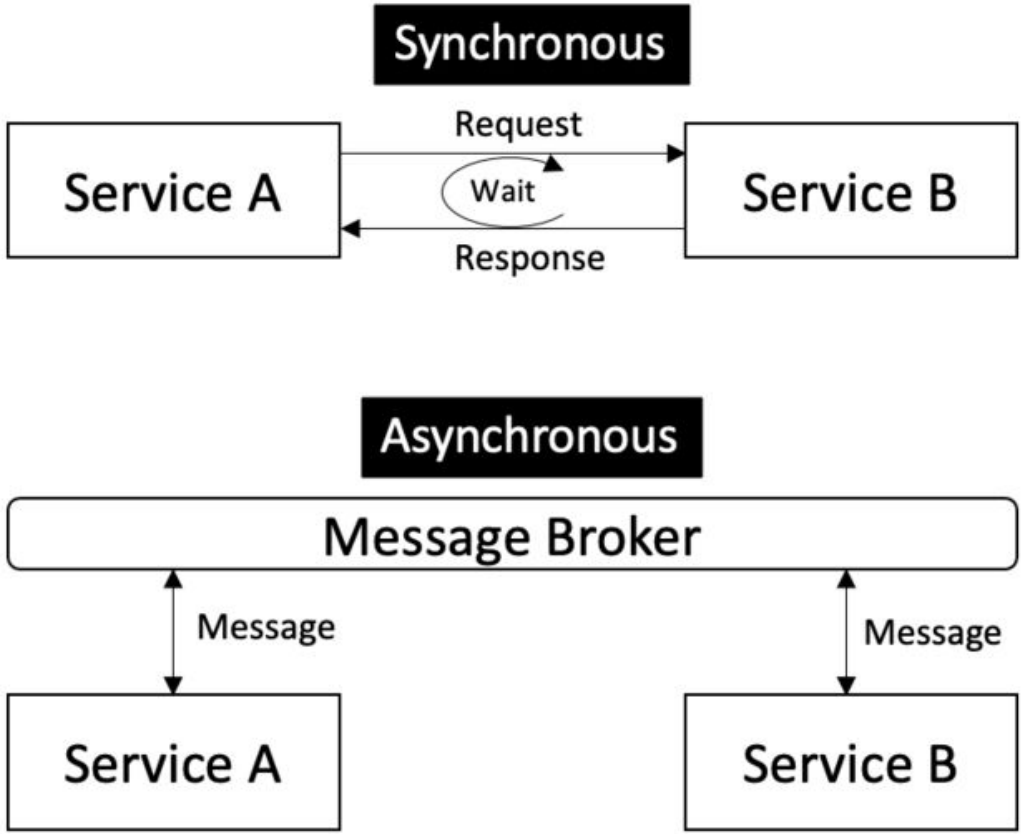

Monolithic Architecture
User Interface
Business Logic
Data Access Layer
Data Base
Microservices Architecture
User Interface
Microservices
Microservices
Presentation Layer
Business Layer
Data Layer
Microservices
Presentation Layer
Business Layer
Data Layer
Microservices
Presentation Layer
Business Layer
Data Layer
Data Base
Data Base
Data Base

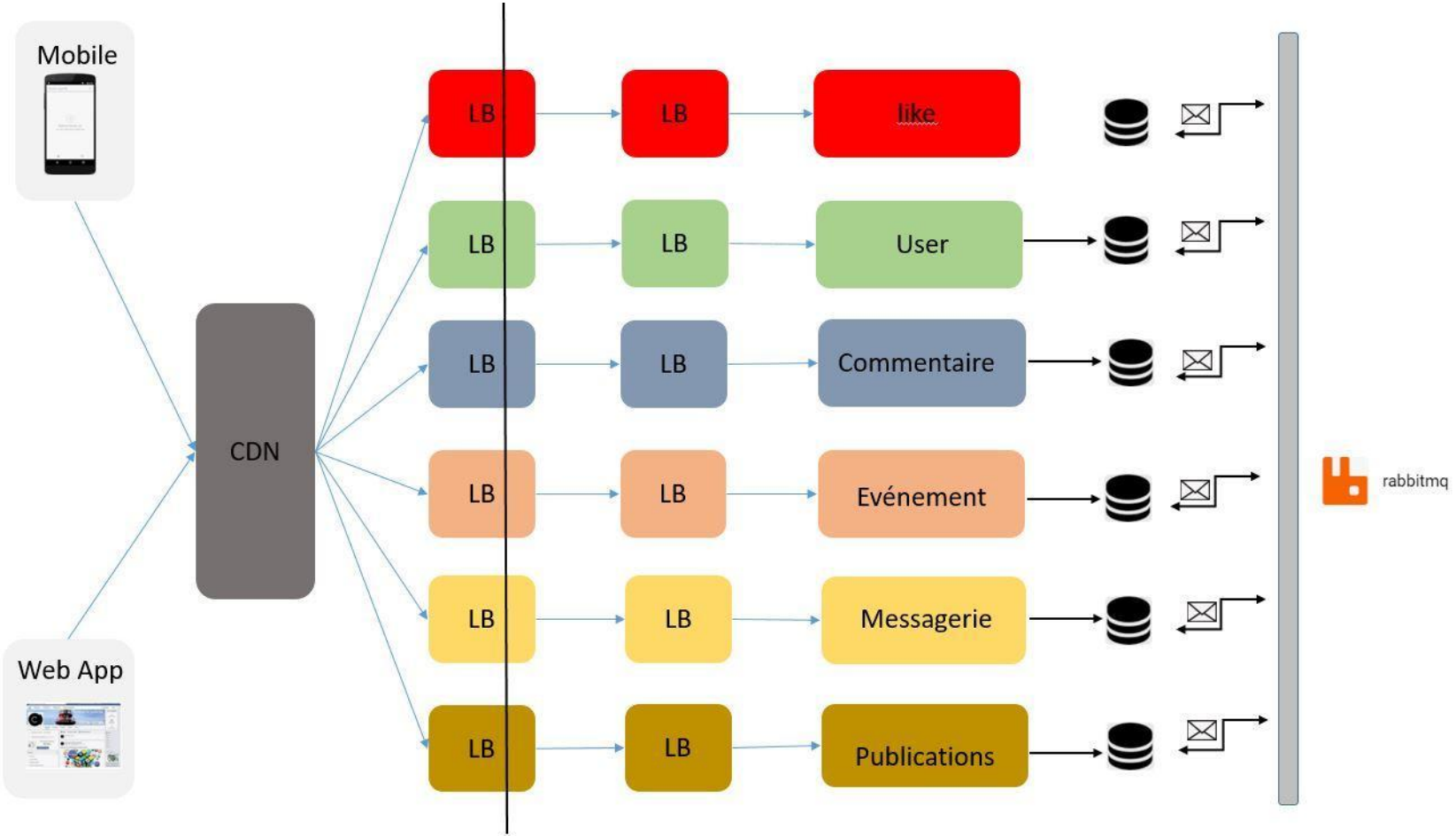
Communication inter-microservice.

La communication inter-microservice est l'un des grands défis d'une architecture orientée microservice. Il s'agit en clair du transfert de données entre microservices.

En effet deux possibilités s'offrent à nous :

- Communication Synchrone **HTTP**
- Communication Asynchrone **AMQP** (cf. **RabbitMQ** sur l'architecture générale).


Synchronous
Request
Service A
Wait
Service B
Response
Asynchronous
Message Broker
Message
Message
Service A
Service B


Mobile
Web App
CDN
LB
LB
like
LB
LB
User
LB
LB
Commentaire
LB
LB
Evénement
LB
LB
Messagerie
LB
LB
Publications
rabbitmq

Comprendre le Big data

De la data au Big data.

Introduction au Big Data

Plusieurs définitions ont été donnés au **Big Data** nous retiendrons dans cet article, celle de Wikipedia qui est la suivante :

« «Big Data, littéralement les grosses données, est une expression anglophone utilisée pour désigner des ensembles de données qui deviennent tellement volumineux qu'ils en deviennent difficiles à travailler avec des outils classiques de gestion de base de données. Dans ces nouveaux ordres de grandeur, la capture, le stockage, la recherche, le partage, l'analyse et la visualisation des données doivent être redéfinis. »

Les 5V du Big data.

Volume : indique la taille et les quantités de Big Data que les entreprises gèrent et analysent.

Valeur : la valeur du Big Data provient généralement de la découverte d'informations et de la reconnaissance de modèles qui conduisent à des opérations plus efficaces, à des relations plus solides avec les clients et à d'autres avantages commerciaux clairs et quantifiables.

Variété : définit la diversité et la gamme des différents types de données, y compris les données non structurées, les données semi-structurées et les données brutes.

Les 5V du Big data.

Vélocité : indique la vitesse à laquelle les entreprises reçoivent, stockent et gèrent les données – comme le nombre spécifique de messages sur les réseaux sociaux ou de requêtes de recherche reçues sur un jour, sur une heure ou sur une autre unité de temps.

Véracité : désigne le degré de « vérité » ou d'exactitude des données et des informations, qui détermine souvent le niveau de confiance des dirigeants.

Système de messaging.

Système de messaging

Le **système de messaging** est responsable du transfert de données d'une application à une autre.

Ce système intermédiaire permet aux applications de d'utiliser les données sans se soucier des processus inhérents à leur collecte ou à leur transmission.

Les messages sont stockés de manière **asynchrone** dans les files d'attente.

Modèle de conception des systèmes de messaging

On peut distinguer :

- **Les systèmes Point à Point**
- **Les systèmes publish-subscribe**

Système de messaging Point à Point

Dans un **système** point à point, les messages sont stockés dans une file.

Un ou plusieurs **consommateurs** peuvent consommer les messages dans la file, mais un message ne peut être consommé que par un seul consommateur à la fois.

Système de messaging Publish-Subscribe.

Le système **pub/sub** est un système de messagerie caractérisé par le fait que :

- **L'expéditeur *(publisher)* d'une donnees *(message)* ne la dirige pas vers un destinataire spécifique (consumer).**
- **L'expéditeur organise *(classe)* les messages suivant une approche bien déterminée. Le destinataire s'abonne à une ou plusieurs classe de messages *(topic)*.**

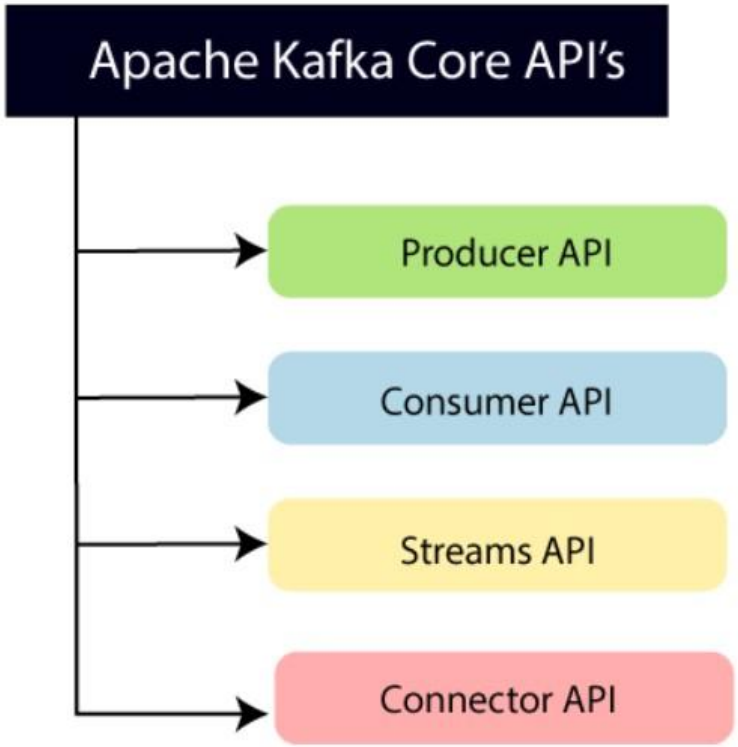
Pour faciliter cette opération, les systèmes **pub/sub** disposent d'un broker : Point central où les messages sont publiés.

Présentation de Kafka.

Apache Kafka

Apache Kafka est une plate-forme logicielle basée sur un processus de streaming distribué.

Il s'agit d'un système de messagerie de **publisher-suscriber** qui permet également l'échange de données entre les applications.



KAFKA

Une plate-forme de streaming possède trois fonctionnalités clés:

- Permettre aux applications clientes **Kafka** de **publier** et **s'abonner** à des flux d'enregistrements, similaires à une file d'attente de messages ou à un système de messagerie d'entreprise comme les Brokers **JMS (ActiveMQ)** ou **AMQP (RabbitMQ)**.

- Permet de stocker les flux d'enregistrements de manière **durable** et **tolérante aux pannes**.
- Permet de traiter les **flux d'enregistrements (stream)** au fur et à mesure qu'ils se produisent **(Real Time Stream Processing)**.

Fonctionnalité	Description
Haut débit	Prend en charge plusieurs millions de messages dans un logiciel modéré.
Scalabilité	Evolutif sans interruption
Réplication	Messages peuvent être repliqués dans le cluster.
Durabilité	Les messages ne sont pas effacés après consommation, ils sont persisté dans le cluster.
Traitement temps réel	Kafka est peut être utilisé pour traiter de gros volumes en temps réel.
Perte de données	Kafka préserve de la perte de données.

Les cas d’utilisation

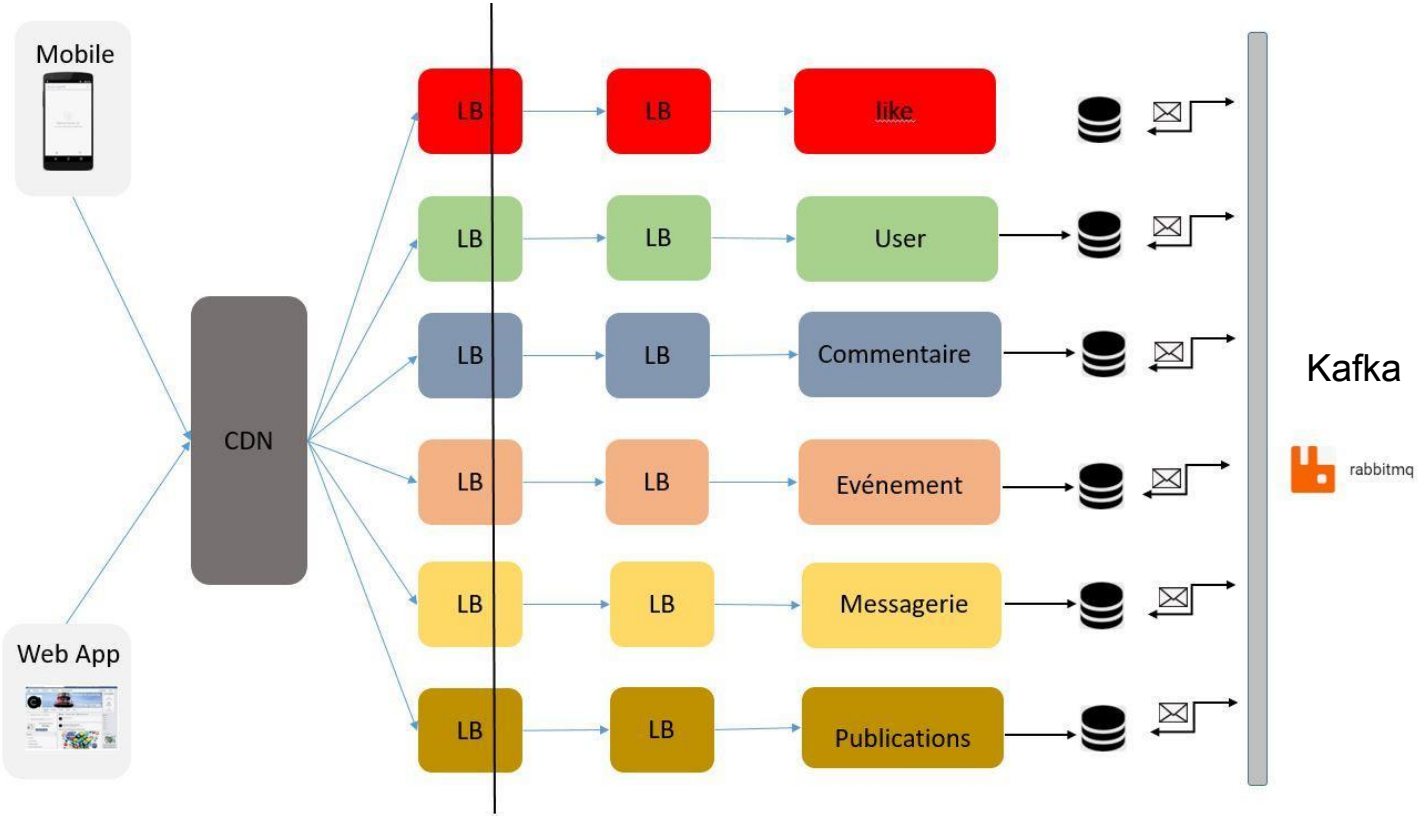
Assurer le tracking d'un site web.

L'un des principaux cas d'utilisation de **Kafka** consiste à assurer le suivi de l'activité d'un site web *(plus généralement d'une application)*.

Les sites web génèrent de grandes quantités de données de différentes natures *(pages web, activité de l'utilisateur…)*.

Kafka permet d'assurer le bon acheminement des données.

Message broker.



Kafka : Vue d’ensemble.

Concepts de base KAFKA.

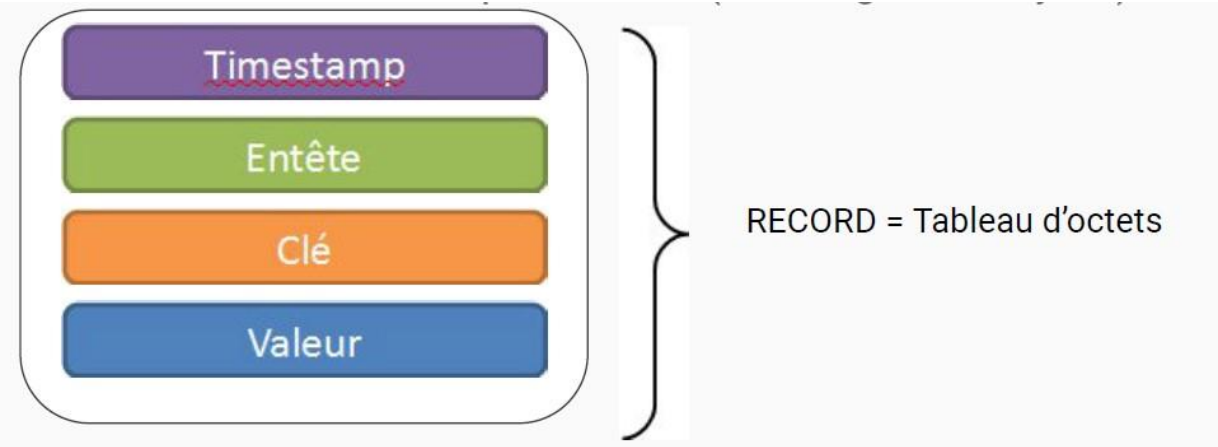
Les messages ou records.

L'unité de données dans Kafka est appelée **message**. Si vous abordez **Kafka** du point de vue d'une base de données, vous pouvez le comparer à une ligne ou à un enregistrement.

- **Un message est simplement un tableau d'octets pour Kafka ;**
- **Les données qu'il contient n'ont donc pas de format ni de signification spécifique pour Kafka.**
- **Un message peut contenir une métadonnée facultative, appelée clé. La clé est également un tableau d'octets et, comme le message, n'a pas de signification.**

Les messages ou records.

- Message ou record est la plus petite unité de données
- Taille maximale de **1Mo** par défaut (message.max.bytes).
- Formats : JSON, String , Avro



KAFKA Topics.

- Un **Topic** fait référence à une catégorie ou à un nom commun utilisé pour stocker et publier un flux particulier de données. La notion de **Topic Kafka** se rapproche de celle de Table dans les **SGBDR** (sans contraintes).
- La notion de **topic** s'assimile aussi à un dossier dans un système de fichiers.
- Un **topic** se définit par son nom.
- Ainsi, dans l'écosystème **Kafka**, un producteur publie les données dans le topic et le consommateur lit les données d'un topic en s'abonnant *(cf système de massaging publisher-subscriber)*.

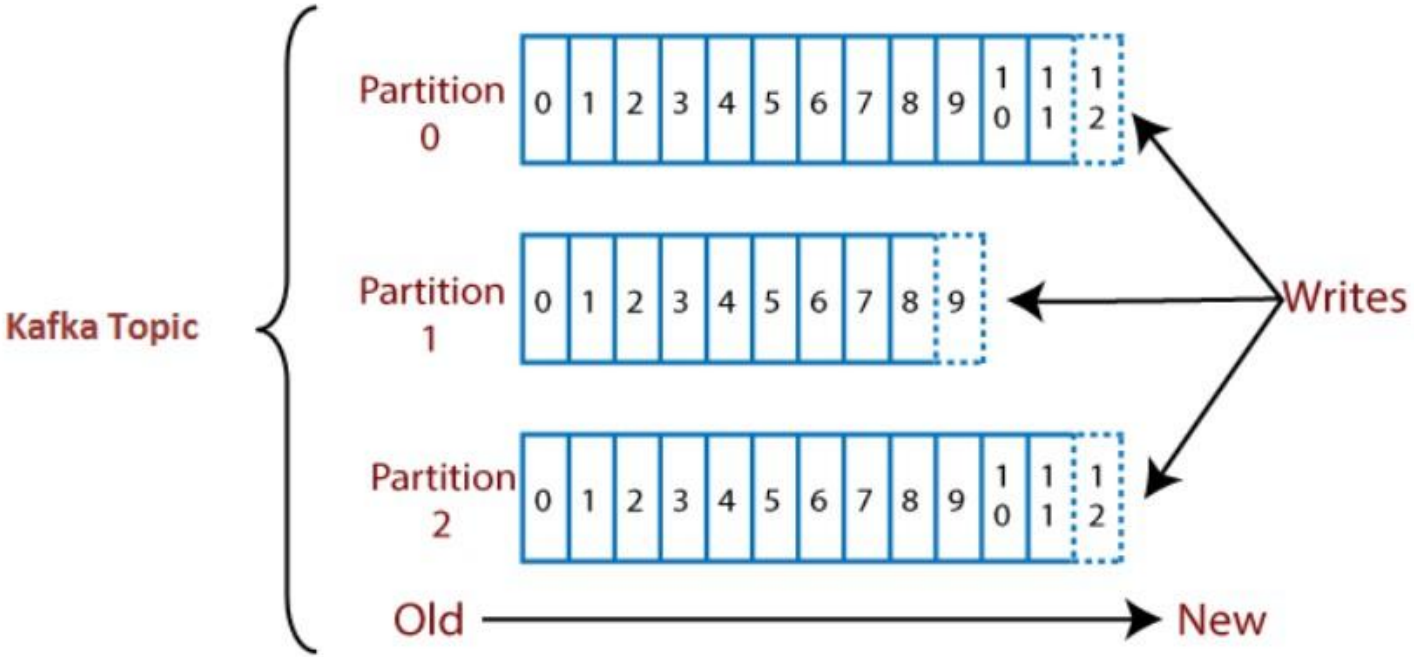
KAFKA Topics.

- Les messages sont regroupés dans des topics.
- Multi-subscriber **(0* producer et/ou consumer)**.
- Découper en partitions - commit log

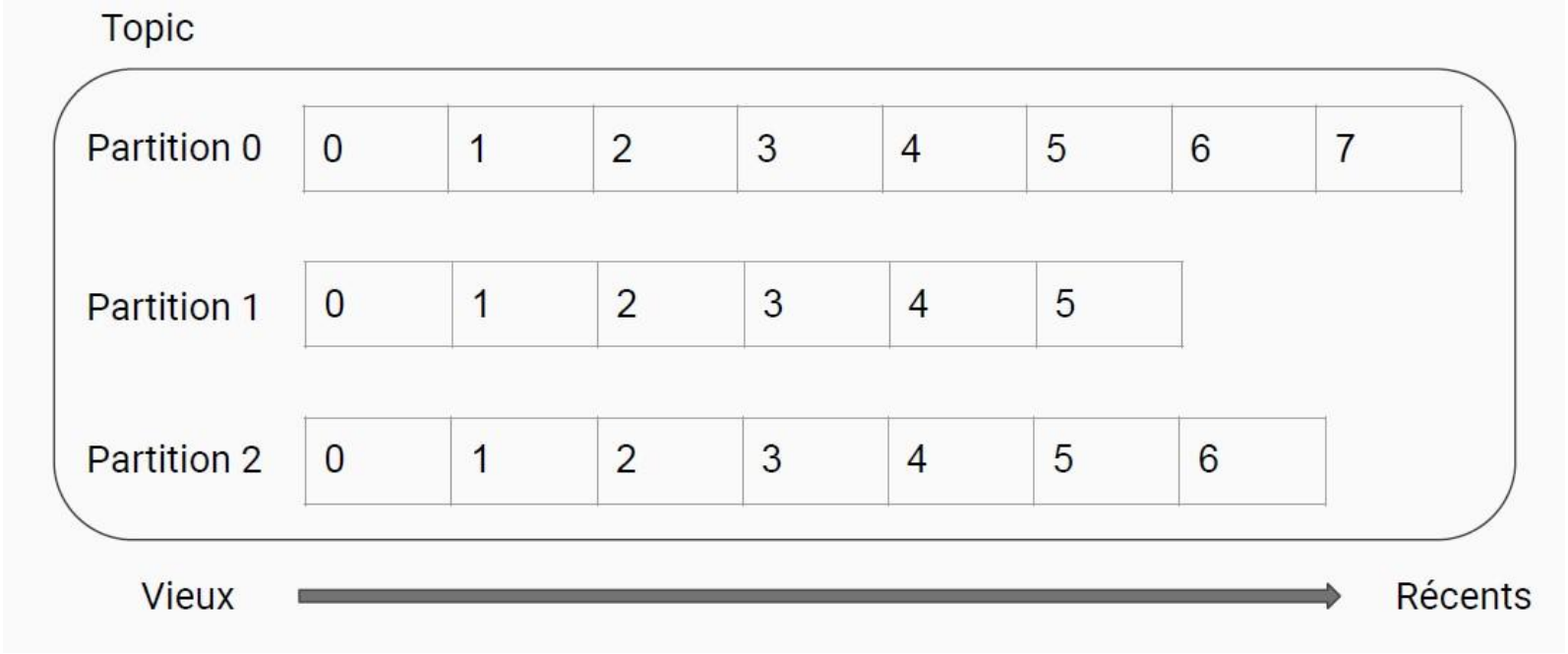
Partitions

- Un topic est divisé en plusieurs parties appelées les **partitions** du topic.
- Chaque **partition** contient des messages dans une **séquence ordonnée**.
- Une donnée écrite sur une partition ne peut jamais être modifiée. On dit que les données sont immuables.
- Les messages sont rajoutés à la fin.
- Chaque **record (message)** reçoit une partition unique.
- Unité de parallélisme
- Les messages sont **persistants** (La durée de rétention des messages est configurable).

Partitions - Illustration.



Partitions - Illustration.



Offset

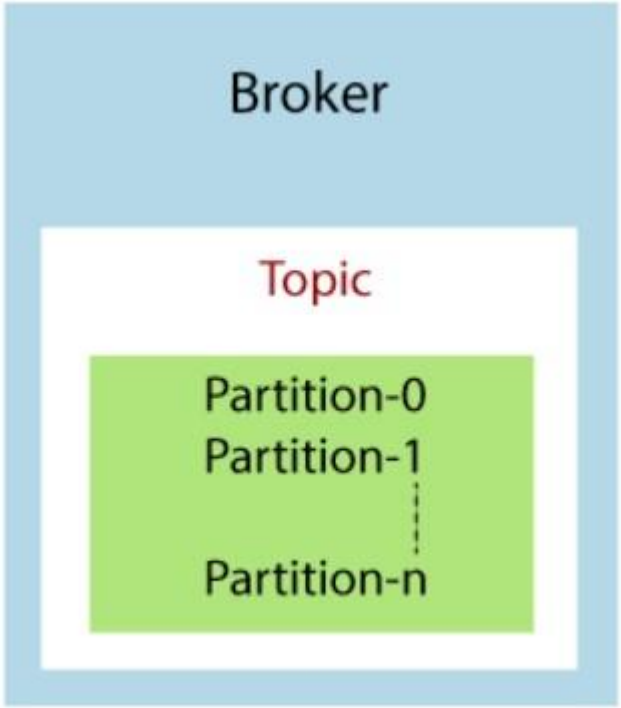
Chaque message est stocké dans des partitions avec un identifiant incrémentiel appelé **offset**.

Une donnée écrite sur une partition ne peut jamais être modifiée. On dit que les données sont immuables.

Brokers.

C’est le rôle joué par **Apache Kafka**.

Un **broker** est un conteneur qui contient plusieurs topics avec leurs multiples partitions.

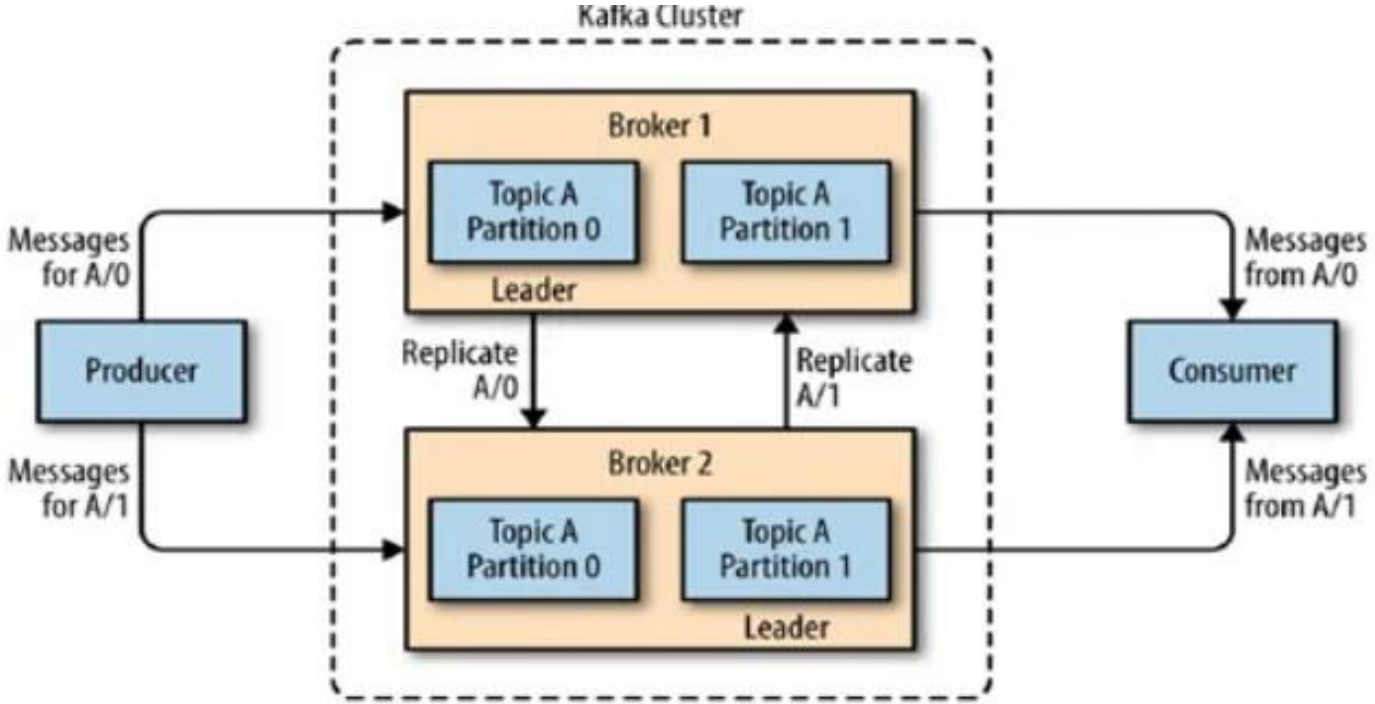


Cluster

- Un cluster Kafka est composé d'un ou plusieurs serveurs appelés **brokers.**

- Les **brokers** du cluster sont identifiés uniquement par un identifiant entier.

Single cluster



Producer - Application émettrice de données.

- Le **producer** (producteur) est tout simplement l'application qui envoie les données.
- Un **producteur** est celui qui publie ou écrit des données sur les topics au sein de différentes partitions.
- Les **messages** sont ajoutés à l'un des topics.
- Les **producers** savent automatiquement (sans mention explicite de l'utilisateur) sur quelle partition et quel broker ajouter le message.

Pour écrire les données dans le cluster, deux approches sont possibles : **Round Robin et Message key.**

L’approche “message keys”

Apache Kafka dispose d’une clé pour favoriser l’envoie de message suivant un ordre spécifique. La clé permet au **producer** d’avoir deux choix :

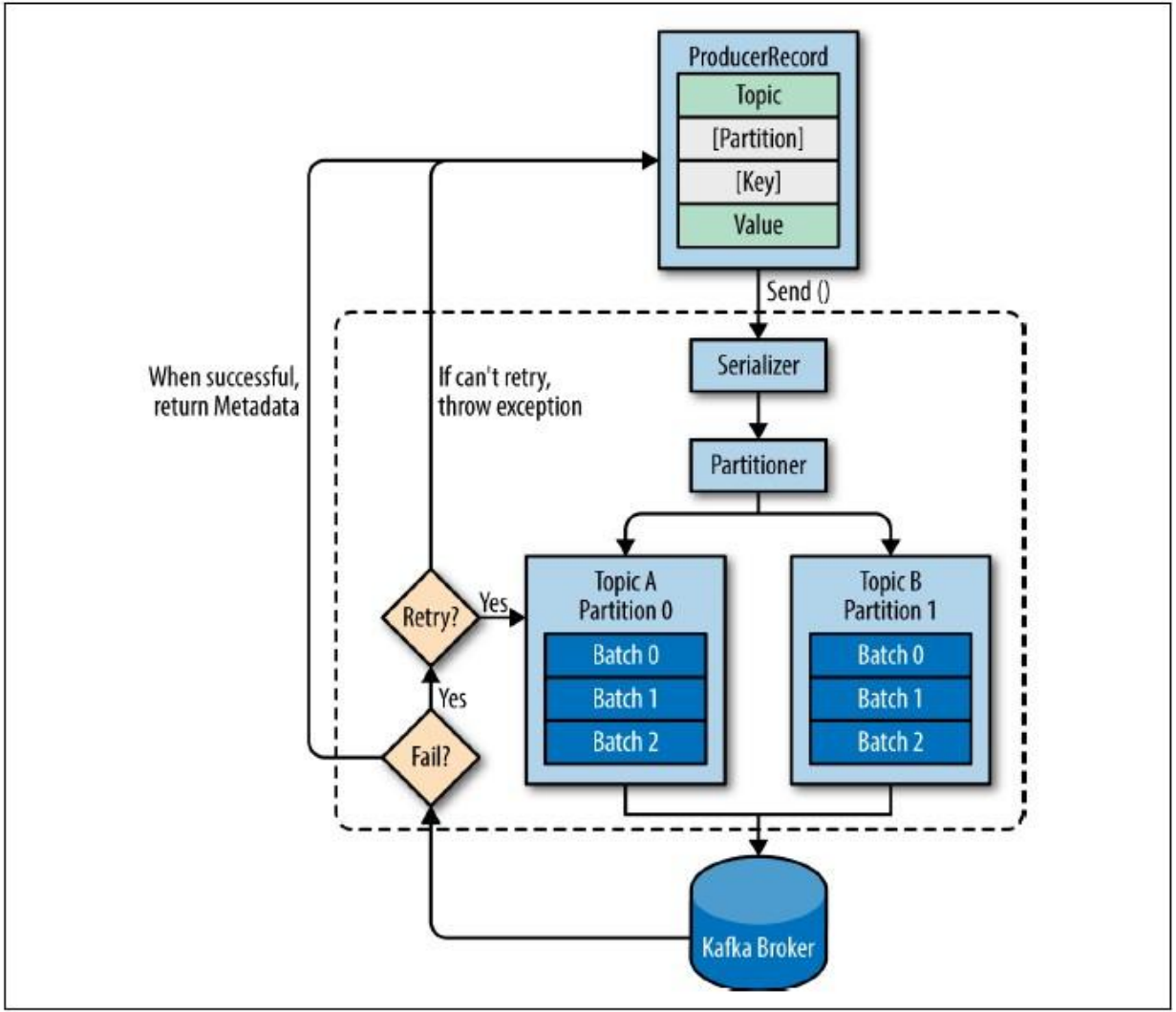
- L’envoie de données à chaque partition
- L’envoie des données à une partition spécifique uniquement.

Par ajout d’une clé aux **messages(données)**, le producer garantit que les messages vont toujours être stockés dans une partition spécifique.

Lorsque la clé n’est pas ajoutée aux messages (données), ils sont envoyés automatiquement suivant un processus de **load balancing**.

La clé peut aussi bien être une chaîne, un nombre… etc .

ProducerRecord



La production de messages dans Kafka.

- Nous commençons à produire des messages vers Kafka en créant un **ProducerRecord**, qui doit inclure le topic auquel nous voulons envoyer l'enregistrement et une **valeur**. En option, nous pouvons également spécifier une **clé et/ou une partition**. Une fois que nous envoyons le **ProducerRecord**, la première chose que le Le producteur fera est de **sérialiser** les objets **clé** et **valeur** dans un tableau d'octets (ByteArrays) afin qu'ils puissent être envoyés sur le réseau.

- Ensuite, les données sont envoyées à un partitionneur. Si une partition est spécifié dans le **ProducerRecord**, le partitionneur ne fait rien et renvoie simplement le record vers la partition spécifiée.

La production de messages dans Kafka.

- Si aucune partition est spécifiée le partitionneur choisira une partition, généralement basée sur sur la clé **ProducerRecord**. Une fois une partition sélectionnée, le producer sait vers quel le topic et quelle partition l'enregistrement ira. L'enregistrement est envoyé au broker Kafka appropriés.

- Lorsque le broker reçoit les messages, il renvoie une réponse. Si les messages ont été écrits avec succès dans **Kafka**, il renverra un objet **RecordMetadata** avec le topic , partition et l'offset de l'enregistrement dans la partition.

- Si le **broker** a échoué l'écriture des messages, il renverra une erreur. Lorsque le producer reçoit une erreur, il peut réessayer d'envoyer le message plusieurs fois avant d'abandonner et de renvoyer une erreur.

La création d’un producer.

La première étape dans l'écriture de messages dans **Kafka** consiste à créer un objet producteur avec les propriétés que vous souhaitez transmettre au producteur. **Un producteur Kafka a trois obligations** :

bootstrap.servers

Liste des paires **hôte:port** de courtiers que le producteur utilisera pour établir les connexion au cluster **Kafka**. Cette liste n’a pas besoin d’inclure tous les courtiers, puisque le producteur obtiendra plus d’informations après la connexion initiale.

La création d’un producer.

key.serializer

Nom d'une classe qui sera utilisée pour sérialiser les clés des enregistrements que nous produirons à **Kafka**. Les brokers **Kafka** attendent des tableaux d'octets comme clés et valeurs des messages.

Cependant, l'interface du producteur autorise, à l'aide de types paramétrés, n'importe quel Java objet à envoyer (comme clé et valeur). Cela donne un code très lisible, mais cela aussi signifie que le producteur doit savoir comment convertir ces objets en tableaux d'octets.

La création d’un producer.

Le producteur utilisera cette classe pour sérialiser l'objet clé dans un tableau d'octets. Le package client **Kafka** inclut **ByteArraySerializer** (qui ne fait pas grand chose), **StringSerializer** et **IntegerSerializer**, donc si vous utilisez des types courants, il il n'est pas nécessaire d'implémenter vos propres sérialiseurs.

Le producteur utilisera cette classe pour sérialiser l'objet clé dans un tableau d'octets. Le package client **Kafka** inclut **ByteArraySerializer** , **StringSerializer** et **IntegerSerializer**, donc si vous utilisez des types courants, il n'est pas nécessaire d'implémenter vos propres sérialiseurs.

Le paramètre **key.serializer** est requis même si vous avez l’intention d’envoyer uniquement des valeurs.

La création d’un producer.

Value.serializer

Nom d'une classe qui sera utilisée pour sérialiser les valeurs des enregistrements que nous produirons à Kafka. De la même manière que vous définissez **key.serializer** sur le nom d'une classe qui sérialisera l'objet clé de message dans un tableau d'octets, vous définissez **value.serializer** sur une classe qui sérialisera l'objet valeur du message.

Acknowledgement : Acks

- Paramétrage important
- Définie quand un record est bien enregistré dans Kafka
- Valeurs possibles :
  - 0
  - 1
  - all, -1

Acknowledgement : Acks

acks=0

- **Description** : Le **producer** ne demande aucun accusé de réception des brokers. Dès que le message est envoyé, le producer le considère comme livré.
- **Comportement** : Le **broker** ne confirme pas la réception, ce qui peut entraîner une perte de messages si le broker tombe en panne avant de traiter le message.
- **Utilisation** : Cette option est la plus rapide car elle minimise la latence, mais elle est risquée car il n'y a aucune garantie que le message soit réellement écrit dans **Kafka**.
- **Risque** : Risque élevé de perte de messages.

Acknowledgement : Acks

acks=1

- **Description** : Le leader du partitionnement Kafka accuse réception du message une fois qu'il l'a écrit sur son disque, sans attendre que les répliques aient confirmé.
- **Comportement** : Le producer considère le message comme livré dès que le leader a écrit le message, mais les répliques ne sont pas encore synchronisées. Si le leader échoue avant que les répliques n'aient pu copier le message, il peut y avoir une perte.
- **Utilisation** : C'est un bon compromis entre performance et fiabilité. Il offre une latence relativement faible, tout en assurant qu'au moins une machine (le leader) a bien reçu le message.
- **Risque** : Risque moyen de perte de messages si le leader échoue avant que la réplication soit terminée.

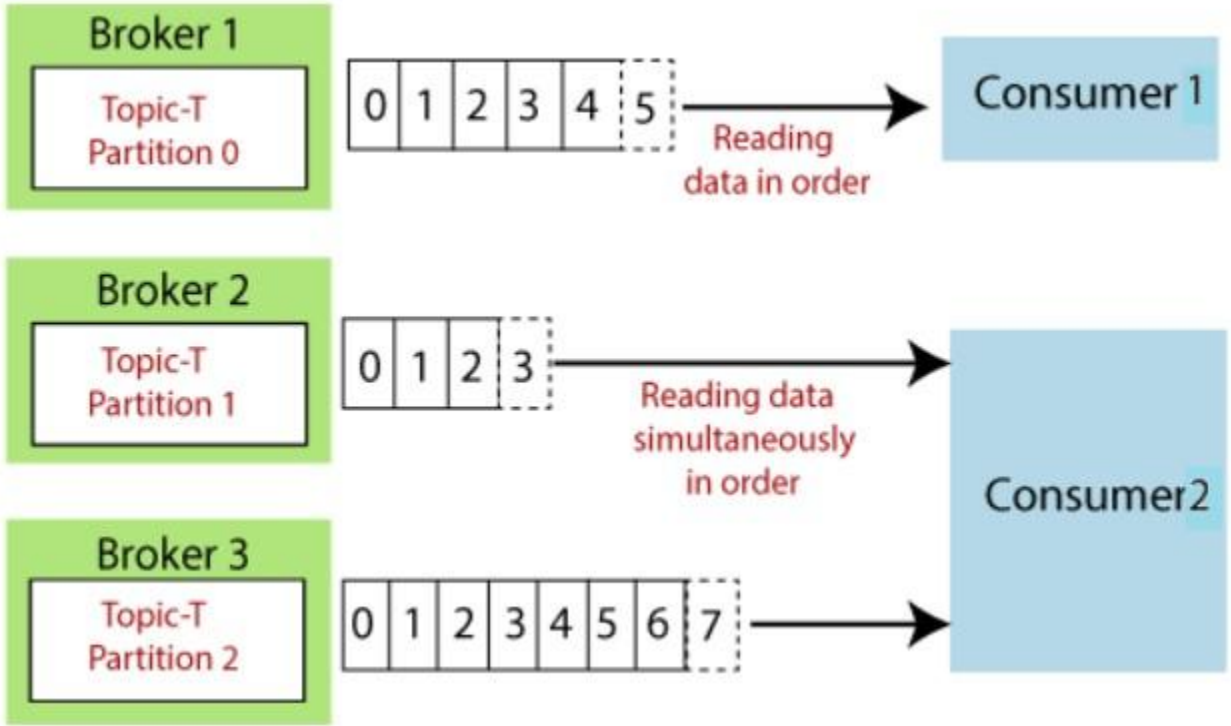
Récapitulatif

Valeur de `acks`	Description	Risque de perte de messages	Latence
0	Pas d'accusé de réception	Élevé	Très faible (rapide)
1	Accusé de réception par le leader	Moyen	Faible
all ou **-1**	Accusé de réception par toutes les répliques	Très faible	Plus élevée

Consumer & Consumer groups

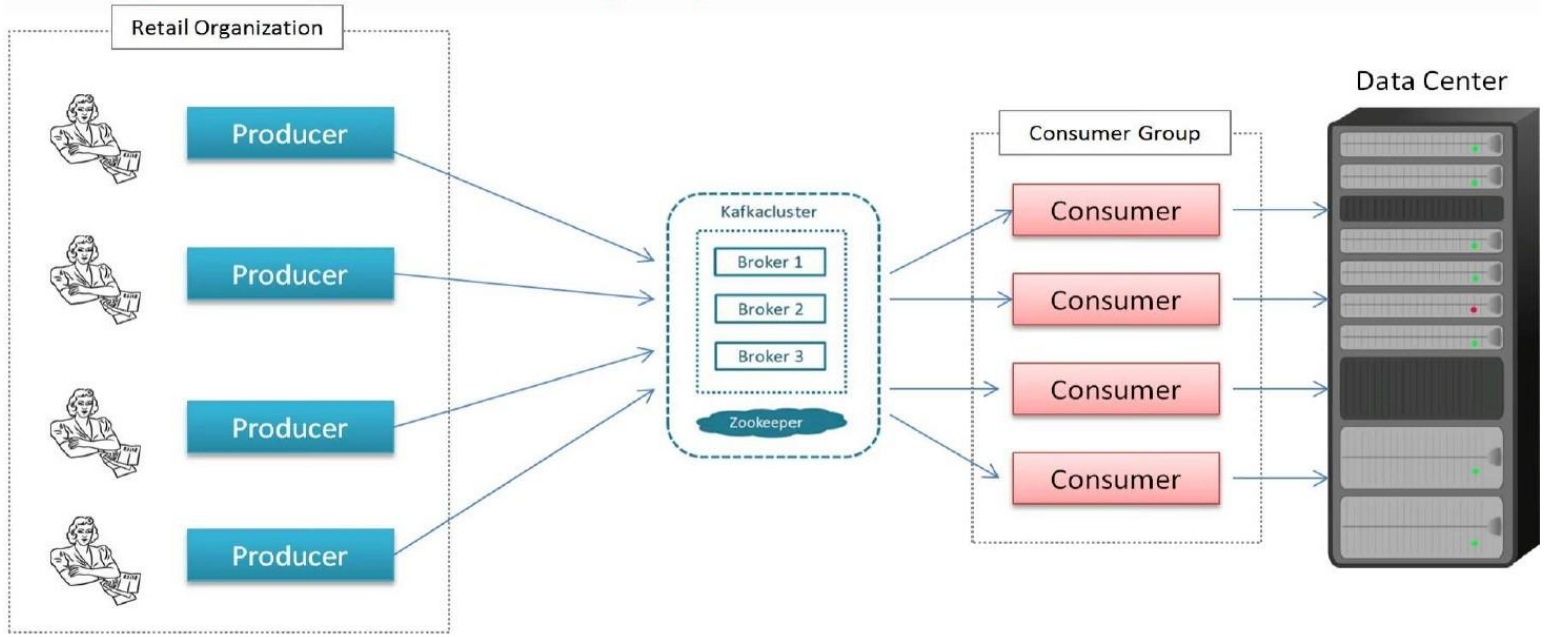
- Un **consumer** est l'application qui lit ou consomme les données d'un cluster **Kafka** via un topic.
- Un **consumer** peut lire les données de plusieurs **brokers** simultanément.
- Un **consumer** lit les données suivant un ordre bien défini.

Illustration

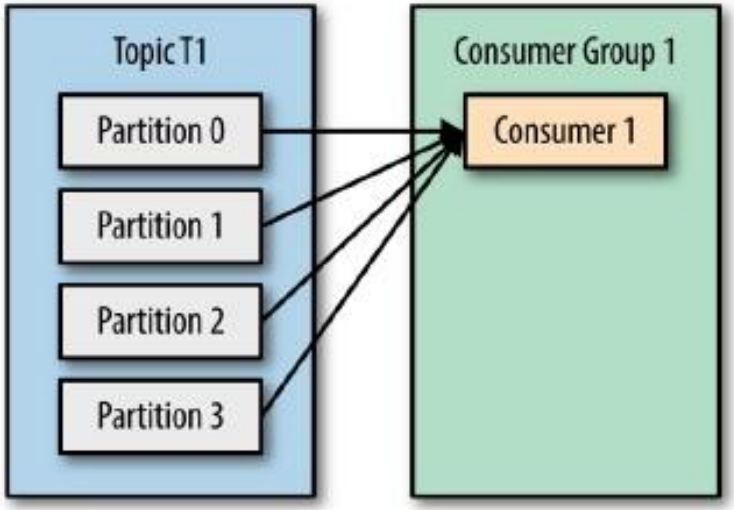


Consumer & Consumer groups

Un **consumer group** désigne un ensemble de consumer qui consomment essentiellement les données d'une application. Chaque consommateur présent dans un groupe lit les données directement depuis les partitions exclusives.



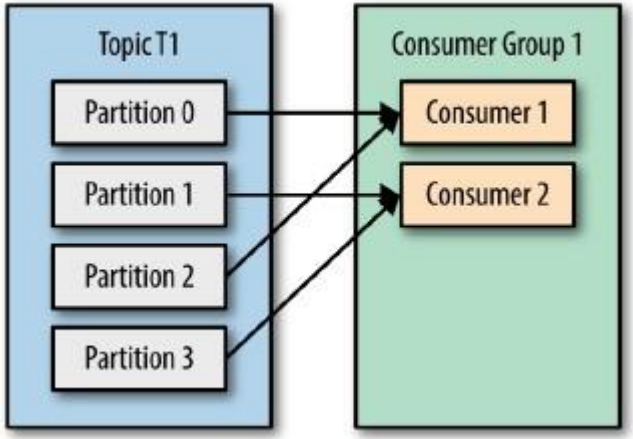
Consumer & Consumer groups



Les consommateurs Kafka font généralement partie d'un **groupe de consommateurs**. Lorsque plusieurs consommateurs sont abonnés à un topic et appartiennent au même groupe de consommateurs, chaque consommateur de le groupe recevra des messages d'un sous-ensemble différent des partitions du topic.

Prenons le sujet **T1** avec quatre partitions. Supposons maintenant que nous créions un nouveau consommateur, **C1**, qui est le seul consommateur du groupe **G1**, et utilisez-le pour vous abonner au topic **T1**. Le consommateur **C1** recevra tous les messages des quatre partitions **t1**.

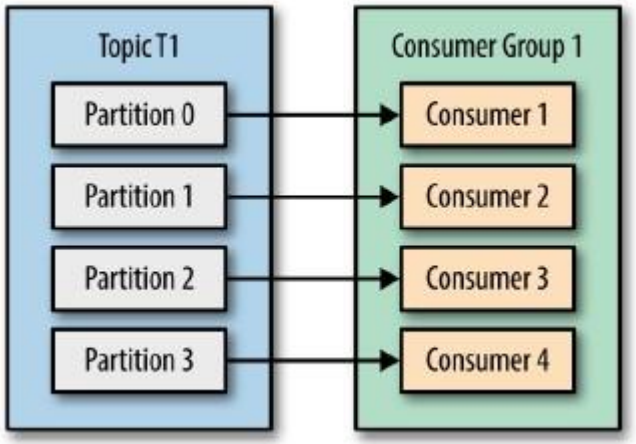
Consumer & Consumer groups



Si nous ajoutons un autre consommateur, **C2**, au groupe **G1**, chaque consommateur ne recevra que des messages à partir de deux partitions.

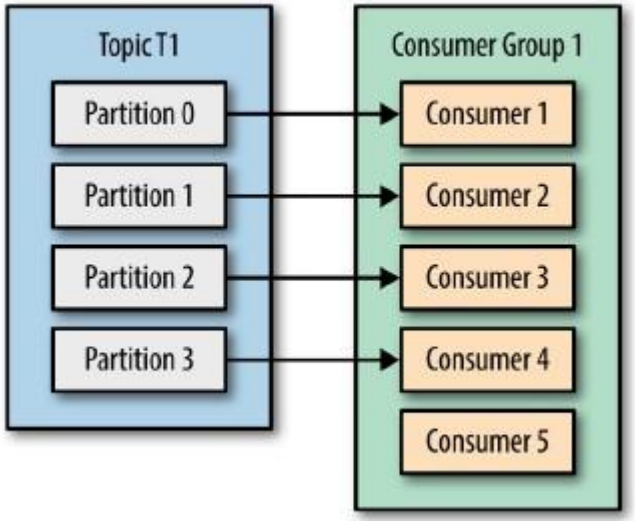
Par exemple, les messages des partitions 0 et 2 vont vers **C1** et les messages des partitions 1 et 3, allez au consommateur **C2**.

Consumer & Consumer groups



Si **G1** a quatre consommateurs, chacun lira les messages d'une seule partition.

Consumer & Consumer groups



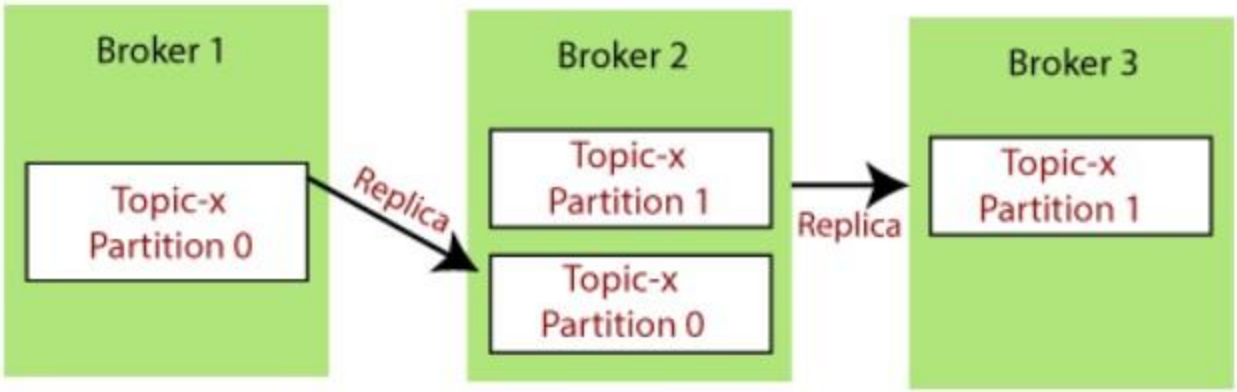
Si nous ajoutons plus de consommateurs à un seul groupe avec un seul sujet que nous n'avons de partitions, certains consommateurs seront inactifs et ne recevront aucun message.

Réplication

Comme de nombreux systèmes de l'écosystème **Big data**, **Apache Kafka** est un système distribué, ce qui suppose donc une réplication des données pour favoriser la haute disponibilité.

Pour ce faire un **facteur de réplication** est créé pour les topics d'un broker particulier. Le **facteur de réplication** désigne le nombre de copies de données sur différents brokers.

Topic replication Factor



Quel broker est susceptible de répondre à la demande du client ?

Leaders VS Followers.

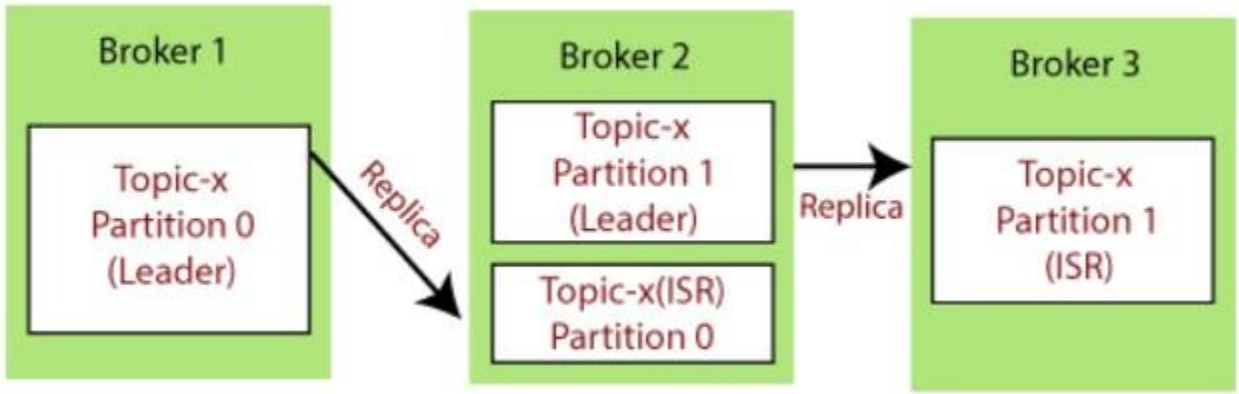
Après réplication des données, pour éviter toute confusion quant au broker qui doit répondre aux sollicitation de consommateurs éventuels, Kafka réalise les action suivantes :

- Une des partition du broker est désignée comme leader et les autres followers.
- En présence du leader aucun follower n’est autorisé à délivrer les données aux clients. Le leader gère toutes les opérations de lecture et d'écriture de données pour les partitions.
- Le **Leader** et ses partisans sont déterminés par le zookeeper

Leaders VS Followers

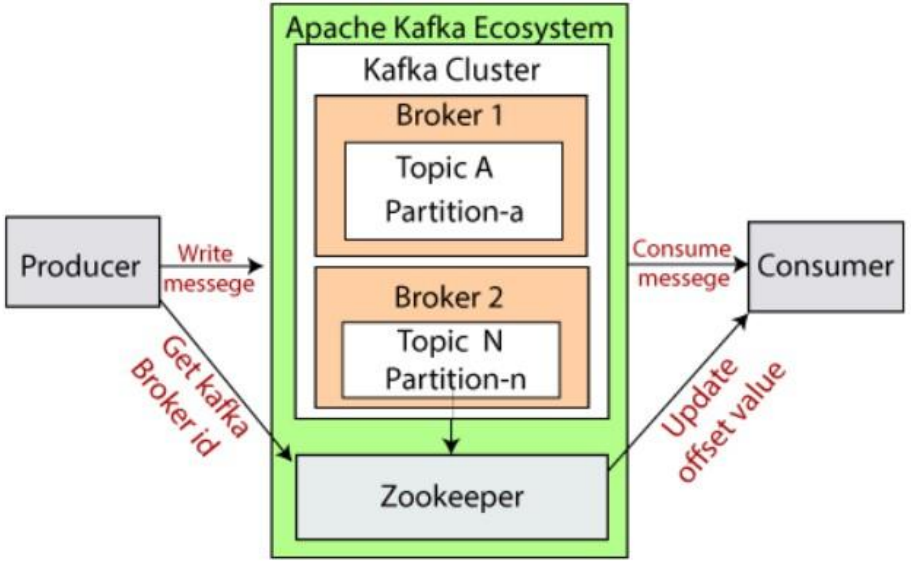
Si le **broker** détenant le rôle de **leader** pour la partition ne parvient pas à servir les données en raison d'une défaillance, l'une de ses répliques *(followers)* respectives prendra le relais.

Ensuite, si le leader précédent revient, il reprend son leadership.



Zookeeper

Un **ZooKeeper** est utilisé pour stocker des informations sur le cluster Kafka et les détails des clients consommateurs.



Apache Kafka Architecture

Zookeeper

- **Kafka broker** utilise **ZooKeeper** pour la gestion et la coordination.

- Il l'utilise également pour informer le producteur et le consommateur de la présence d'un nouveau broker dans le système **Kafka** ou d'une défaillance du broker dans le système **Kafka**.

server.properties.

Server Basics

- **broker.id=0** : Ce paramètre définit **l'ID** unique du broker. Chaque broker dans un cluster Kafka doit avoir un ID unique. Cet ID est utilisé pour identifier le broker au sein du cluster.

Socket Server Settings

- **listeners=PLAINTEXT://:9092** : Cette ligne est commentée, mais elle permet de définir l'adresse sur laquelle le broker Kafka écoute. Ici, le protocole utilisé est **PLAINTEXT** (non chiffré) et le port par défaut est **9092**.

Le format est **listeners = listener_name://host_name:port**

server.properties.

- **advertised.listeners=PLAINTEXT://your.host.name:9092** : L'adresse que le broker va annoncer aux clients Kafka (producers, consumers). Si non définie, Kafka utilisera la même valeur que pour listeners.
- **num.network.threads=3** : Le nombre de threads utilisés par le broker pour recevoir les requêtes réseau et envoyer les réponses aux clients.
- **num.io.threads=8** : Le nombre de threads utilisés pour traiter les requêtes, incluant des opérations d'entrée/sortie (comme l'écriture sur disque).
- **socket.send.buffer.bytes=102400** et **socket.receive.buffer.bytes=102400** : Ces paramètres définissent les tailles des buffers utilisés pour envoyer et recevoir des messages sur le réseau.
- **socket.request.max.bytes=104857600** : La taille maximale d'une requête acceptée par le serveur (100 Mo ici). Cela protège le broker contre des erreurs mémoire (OOM).

server.properties.

Log Basics

- **log.dirs=/tmp/kafka-logs** : Le chemin où les logs **Kafka** (messages) sont stockés. Kafka utilise ce dossier pour sauvegarder les partitions des topics.
- **num.partitions=1** : Le nombre de partitions par défaut pour chaque topic créé.

- **num.recovery.threads.per.data.dir=1** : Le nombre de threads utilisés pour récupérer les données des logs en cas de crash lors du démarrage ou de l'arrêt.

server.properties.

Internal Topic Settings

- **offsets.topic.replication.factor=1** : Le facteur de réplication pour le topic interne ***__consumer_offsets***, qui stocke les offsets de consommation des consumers Kafka. Un facteur supérieur à 1 garantit une haute disponibilité en cas de panne de broker.
- **transaction.state.log.replication.factor=1** : Le facteur de réplication pour le journal d'état des transactions Kafka. Un facteur plus élevé offre une meilleure tolérance aux pannes.
- **transaction.state.log.min.isr=1** : Le nombre minimal de répliques en synchronisation (ISR – In-Sync Replicas) qui doivent confirmer la réception d'un message transactionnel.

server.properties.

Log Flush Policy

- **log.flush.interval.messages=10000 (commenté)** : Le nombre de messages que Kafka accepte avant de forcer l'écriture des logs sur le disque. En ajustant ce paramètre, vous contrôlez la fréquence de vidage des logs sur le disque, ce qui impacte la durabilité et la latence.
- **log.flush.interval.ms=1000 (commenté)** : La période maximale (en millisecondes) pendant laquelle un message peut rester en mémoire avant d'être forcé sur le disque.

server.properties.

Log Retention Policy

- **log.retention.hours=168** : La durée pendant laquelle Kafka conserve les segments de logs. Ici, les segments sont conservés pendant 168 heures (7 jours).
- **log.retention.bytes=1073741824** (commenté) : La taille maximale des logs (1 Go) avant qu'ils ne soient supprimés. Si les logs atteignent cette taille, ils sont supprimés indépendamment de l'âge.
- **log.segment.bytes=1073741824** : La taille maximale d'un segment de log avant qu'un nouveau segment ne soit créé.
- **log.retention.check.interval.ms=300000** : Intervalle (5 minutes) à partir duquel Kafka vérifie si les logs peuvent être supprimés selon les politiques de rétention (basées sur l'âge ou la taille).

server.properties.

Zookeeper

zookeeper.connect=localhost:2181 : La chaîne de connexion à Zookeeper, utilisée par Kafka pour coordonner les brokers. Dans ce cas, Zookeeper est configuré pour s'exécuter sur l'hôte local (localhost) au port 2181.

zookeeper.connection.timeout.ms=18000 : Temps d'attente maximal (18 secondes) pour établir une connexion avec Zookeeper.

server.properties.

Group Coordinator Settings

group.initial.rebalance.delay.ms=0 : Le délai initial avant de commencer le rééquilibrage des consommateurs dans un groupe. Dans cet exemple, le délai est défini à 0 ms, ce qui signifie que le rééquilibrage commence immédiatement. En production, une valeur de 3000 ms *(3 secondes)* est souvent préférable pour éviter des rééquilibrages trop fréquents au démarrage.

Installation & configuration de l’outil.

Procédure

Étape 1 : Vérifier votre version Java.

Étape 2 : Télécharger Apache Kafka sur : https://kafka.apache.org/downloads

Étape 3 : Extraire le fichier compressé.

Étape 4 : Lancer le serveur **Zookeeper** :Naviguer dans le répertoire **Kafka** (de préférence à la racine du disque local) .

Sous Linux :

bin/zookeeper-server-start.sh config/zookeeper.properties

Procédure (Suite)

Sous Windows :

bin/windows/zookeeper-server-start.bat config/zookeeper.properties

Étape 5 : Démarrer le serveur Kafka.

Sous Linux :

bin/kafka-server-start.sh config/server.properties

Sous Windows :

bin\windows\kafka-server-start.bat config\server.properties

Administration Kafka en Ligne de commande.

Création d’un topic.

Nous créons un topic avec les paramètres par défaut :

```
kafka-topics.sh --create --bootstrap-server localhost:9092 --topic topicName
```

Nous créons un sujet avec un **facteur de réplication** et une seule **partition**.

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

Lister et décrire les topics.

Lister tous les topics d’un cluster :

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

Décrire un topic spécifique :

```
kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic topicName
```

Création d’un producer en ligne de commande.

Création d’un producer pour l’envoie de messages sans clé(Key) :

```
kafka-console-producer.sh --bootstrap-server localhost:9092 --topic topicName
```

Création d’un producer pour l’envoie de messages avec clé :

```
kafka-console-producer.sh --bootstrap-server localhost:9092 --topic
topicName --property parse.key=true  --property key.separator=,
```

Chaque ligne est envoyée comme message distinct - Tapez **Ctrl + C** pour quitter.

Création d’un consumer en ligne de commande.

Lancer un **consommateur** :

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic
```

Pour consommer les messages depuis le début :

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic
--from-beginning
```

Pour consommer les messages en affichant les clés et le timestamp de création :

Création d’un consumer en ligne de commande.

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic --from-beginning
--property print.key=true --property print.timestamp=true
```

Stopper le serveur Kafka

```
kafka-server-stop
```

Architecture à haute disponibilité.

Architecture haute disponibilité.

Pour mettre en place une **architecture à haute disponibilité** avec **Apache Kafka**, plusieurs aspects clés doivent être pris en compte, notamment la réplication et la sécurité des données, la gestion des clusters de brokers et de **Zookeeper**, **Kafka Connect**, ainsi que la **gestion des groupes et des charges**.

Réplication & sécurité des données.

Cohérence des Messages : Kafka garantit la cohérence des messages grâce à sa politique de réplication. Les messages sont répliqués sur plusieurs brokers pour assurer la résilience en cas de défaillance.

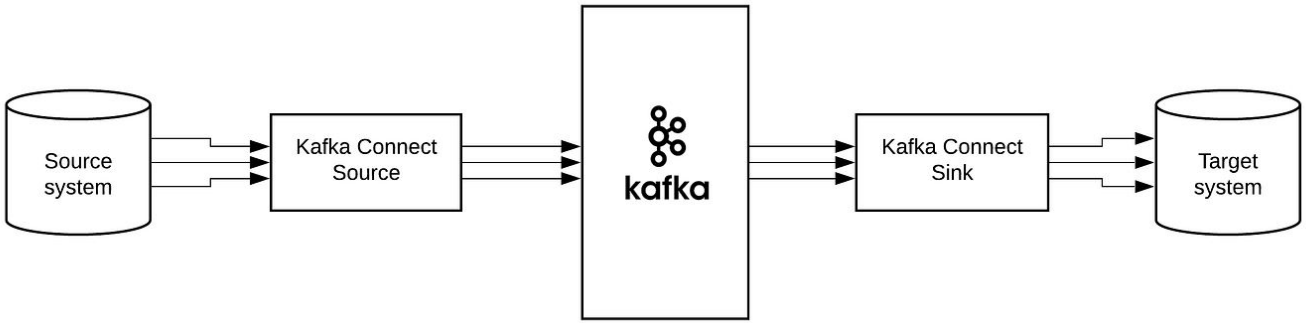
Gestion des Pannes de Brokers : Kafka gère les pannes de brokers en utilisant des leaders et des répliques pour chaque partition. En cas de panne d'un broker leader, une des répliques devient le nouveau leader.

Gestion des Logs : Kafka stocke les messages sous forme de logs. La gestion efficace des logs, y compris leur compaction et leur rétention, est cruciale pour maintenir les performances et la stabilité du système.

Kafka Connect.

Kafka Connect est utilisé pour intégrer **Kafka** avec d'autres systèmes de données.

Il permet la configuration de connecteurs **source** et **sink** pour importer et exporter des données vers et depuis **Kafka**.



Mise en place d’un cluster Kafka.

Étape 1 : Démarrer Zookeeper

- Commencez par démarrer Zookeeper : bin/zookeeper-server-start.sh config/zookeeper.properties

Étape 2 : Configurer et Démarrer les Brokers Kafka

Pour chaque broker Kafka dans votre cluster, vous devrez :

Créer un fichier de configuration distinct pour chaque broker : Copiez server.properties et modifiez-le pour chaque broker.

Par exemple :

Pour le **broker 1** : copiez **server.properties** en **server-1.properties**

Pour le **broker 2** : copiez **server.properties** en **server-2.properties**

Pour le **broker 3** : copiez **server.properties** en **server-3.properties**

Mise en place d’un cluster Kafka.

Modifier les configurations pour chaque broker :

Dans chaque fichier server-x.properties, modifiez les propriétés suivantes :

- **broker.id**: Donnez un ID unique à chaque broker *(0 pour le broker 1, 1 pour le broker 2, 2 pour le broker 3)*.
- **listeners**: Si nécessaire, configurez des ports d'écoute différents pour chaque broker (p*ar exemple, PLAINTEXT://:9092, PLAINTEXT://:9093, PLAINTEXT://:9094)*.
- **log.dirs**: Configurez un répertoire de log unique pour chaque broker (par exemple, /tmp/kafka-logs-1, /tmp/kafka-logs-2, /tmp/kafka-logs-3).

Mise en place d’un cluster Kafka.

Démarrer chaque broker Kafka :

Ouvrez un terminal séparé pour chaque broker et exécutez :

```
# Pour le broker 1
bin/kafka-server-start.sh config/server-1.properties
# Pour le broker 2
bin/kafka-server-start.sh config/server-2.properties
# Pour le broker 3
bin/kafka-server-start.sh config/server-3.properties
```

Mise en place d’un cluster Kafka.

Étape 3 : Vérifier l'État du Cluster

Après avoir démarré tous les brokers, vous pouvez vérifier l'état du cluster en listant les topics ou en vérifiant l'état des brokers.

Vue d’ensemble de ksqlDB et écosystème.

Pourquoi ksqlDB ?

ksqlDB est un système de gestion de bases de données orienté vers le **streaming**,conçu pour permettre aux utilisateurs de construire des applications orientées flux de données en temps réel.

Utiliser **ksqlDB** présente plusieurs avantages, en particulier pour ceux qui travaillent avec des données en temps réel :

- **Traitement en Temps Réel** : **ksqlDB** permet le traitement et l'analyse des flux de données en temps réel, ce qui est crucial pour les applications nécessitant des réponses rapides, comme la détection de fraude ou le monitoring en temps réel.

Pourquoi ksqlDB ?

- **Intégration avec Kafka : ksqlDB** est conçu pour fonctionner de manière transparente avec **Apache Kafka**, ce qui le rend idéal pour les environnements déjà utilisant Kafka pour le traitement des flux de données.
- **Simplicité de Développement** : Il simplifie le développement d'applications orientées données en temps réel, en réduisant la complexité technique associée à ce type de traitement.

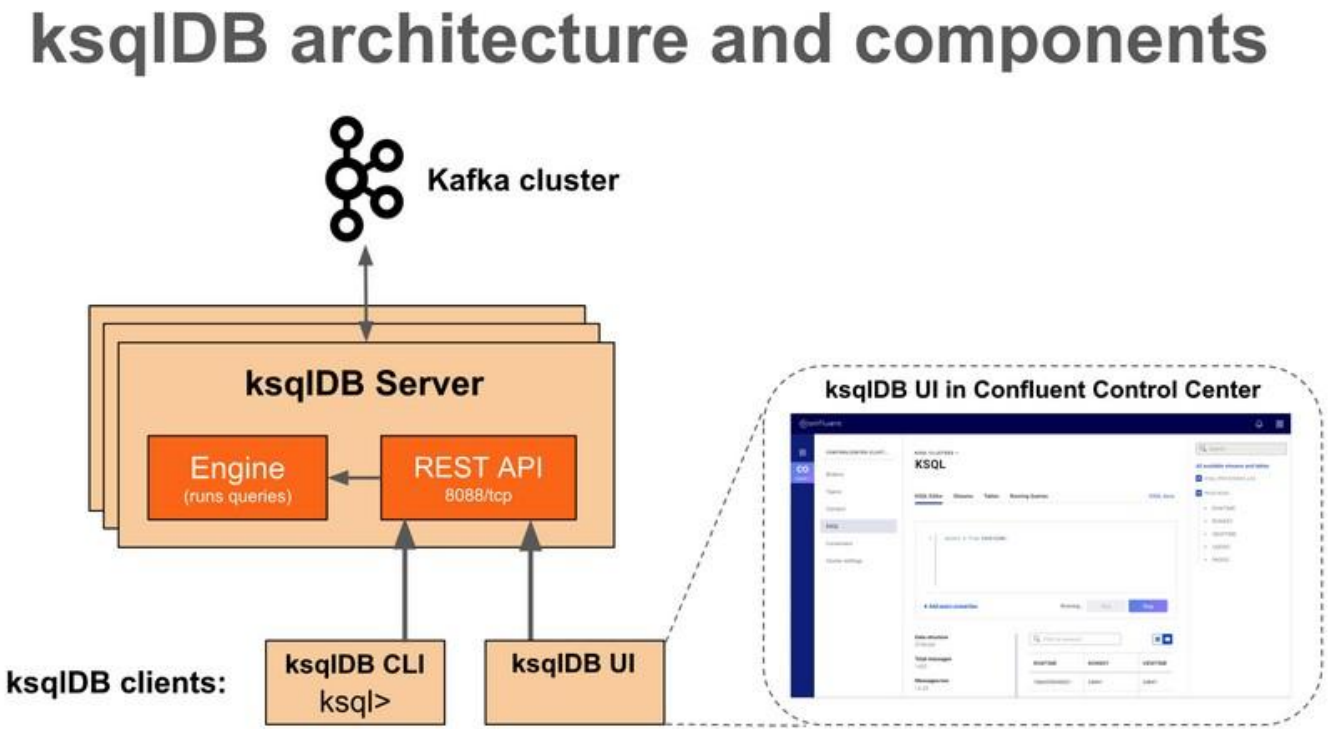
KsqlDB

Développé par Confluent : **ksqlDB** a été développé par Confluent, la société derrière **Apache Kafka.**

Basé sur Kafka : Il est étroitement intégré avec Kafka, utilisant ce dernier pour le stockage des données et pour les flux de données.

Traitement de flux SQL : Permet de traiter et de questionner des flux de données en utilisant le langage SQL, rendant ainsi les opérations sur les données en temps réel plus accessibles.

KsqlDB Architecture.



Composants de l’architecture

ksqlDB a ces composants principaux :

Moteur ksqlDB -- traite les instructions et requêtes **SQL**

Interface REST : permet au client d'accéder au moteur

ksqlDB CLI -- console qui fournit une interface de ligne de commande (CLI) au moteur

Interface utilisateur ksqlDB : permet de développer des applications **ksqlDB** dans Confluent Control Center et Confluent Cloud.

KsqlDB CLI

ksqlDB CLI est une interface en ligne de commande pour interagir avec ksqlDB, qui est un système de traitement de flux de données pour Apache Kafka.

```
$ ./bin/ksql
```


```
Copyright 2017-2022 Confluent Inc.

CLI v0.27.2, Server v0.27.2 located at http://localhost:8088
Server Status: RUNNING

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

KsqlDB CLI

Accès et Configuration : Pour utiliser **ksqlDB CLI**, il faut lancer une instance de **ksqlDB Server**. Une fois le serveur en marche, vous pouvez vous connecter à l'aide de **ksqlDB CLI** en spécifiant l'adresse du serveur.

Exécution de Requêtes SQL : ksqlDB CLI permet d'exécuter des requêtes SQL en temps réel sur des flux de données. Cela inclut la création de flux et de tables, l'exécution de requêtes de sélection, et la manipulation de données en temps réel.

Création de Flux et de Tables : Vous pouvez créer des flux (pour des données en mouvement) et des tables (pour des données agrégées ou statiques) en utilisant une syntaxe SQL standard.

Manipulation des Données en Temps Réel : ksqlDB permet de transformer, filtrer, agréger et joindre des flux de données en temps réel.

KsqlDB CLI

Scripts et Automation : Vous pouvez également utiliser **ksqlDB CLI** pour exécuter des scripts **SQL**, ce qui est utile pour l'automatisation et la configuration initiale des flux de données.

Débogage et Monitoring : La **CLI** fournit des commandes pour déboguer et surveiller l'état de vos requêtes et flux de données.

Les modes serveur de ksqlDB.

Les modes serveurs de **ksqlDB** se réfèrent aux différentes manières dont le système peut être configuré et déployé. **ksqlDB** supporte principalement deux modes de serveur :

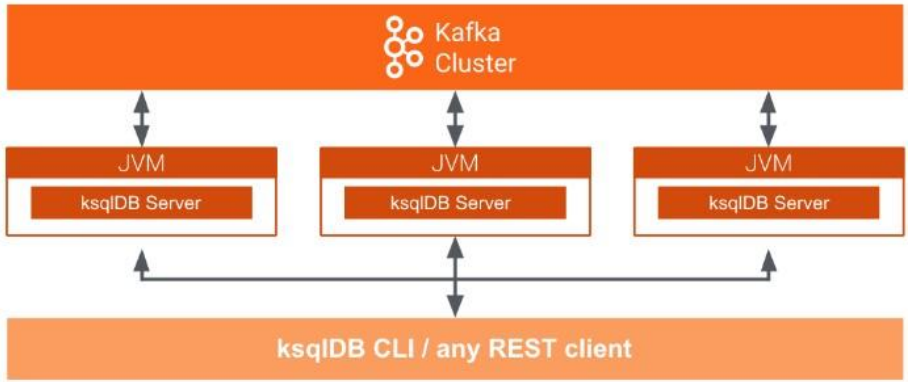
Mode Serveur Interactif : Dans ce mode, **ksqlDB** fonctionne comme un service interactif permettant aux utilisateurs de soumettre des requêtes ***SQL*** en direct pour le traitement des flux de données. Ce mode est souvent utilisé pour des requêtes ad hoc et pour le développement

Les modes serveur de ksqlDB.

Mode Serveur Interactif :

En mode interactif, vous pouvez :

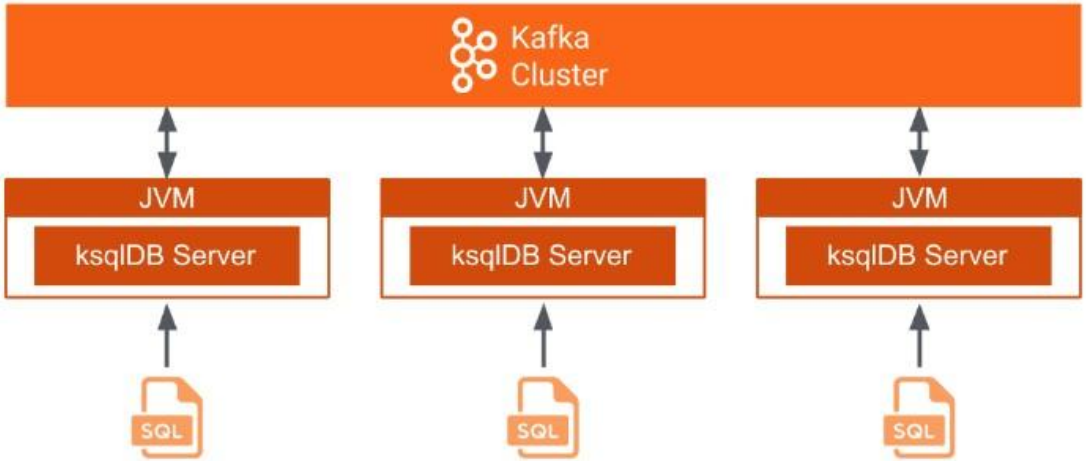
- Écrire des déclarations et des requêtes à la volée
- Démarrer dynamiquement un nombre illimité de nœuds de serveur.
- Démarrer une ou plusieurs CLI ou clients REST



Les modes serveur de ksqlDB.

Mode Serveur Headless (sans tête) :

Dans ce mode, **ksqlDB** exécute un ensemble prédéfini de requêtes SQL à partir d'un fichier de configuration. Ce mode est idéal pour les déploiements en production où les requêtes sont définies à l'avance et ne nécessitent pas d'interaction en temps réel.



Les modes serveur de ksqlDB.

Mode Serveur Headless (sans tête) :

En mode sans tête, vous pouvez :

- Démarrez n'importe quel nombre de nœuds de serveur
- Transmettez un fichier SQL avec les instructions SQL à exécuter.
- Assurer l’isolement des ressources

Utiliser ksqlDB.

Streams et tables.

Streams :

- Un stream dans **ksqlDB** représente un flux de données en temps réel. Il est analogue à un topic Kafka où chaque message est un enregistrement d'un événement.
- Les streams sont par nature immuables ; vous ne pouvez pas modifier les données une fois qu'elles sont dans le stream.

Tables :

- Une table dans **ksqlDB** représente un état agrégé, similaire à une table de base de données traditionnelle.
- Les données dans une table peuvent être modifiées ; elles reflètent l'état le plus récent pour une clé donnée.
- Les tables sont mises à jour de manière incrémentielle à mesure que de nouveaux messages arrivent.

Agrégations fenêtrées (Windowed operation).

Les fonctions d'agrégation fenêtrées dans **ksqlDB**, telles que **TUMBLING**, **HOPPING**, et **SESSION**, sont utilisées pour effectuer des calculs agrégés sur des données qui sont groupées en fonction de périodes de temps définies.

Ces fonctions permettent de traiter des flux de données en temps réel en les segmentant en fenêtres temporelles, offrant ainsi une vue dynamique et temporelle des données.

TUMBLING Windows.

Les fenêtres **TUMBLING** sont des fenêtres de temps fixes qui ne se chevauchent pas. Elles sont utiles pour agréger des données sur des intervalles de temps réguliers.

Utilisation: Utilisez **TUMBLING** lorsque vous voulez des résumés pour des intervalles de temps distincts et non chevauchants.

Syntaxe:

SELECT <champs_agreges>, **WINDOWSTART**, **WINDOWEND**

FROM <stream/table>

WINDOW TUMBLING (SIZE <taille_fenetre>)

GROUP BY <champ_groupement>;

TUMBLING Windows.

Exemple :

```
SELECT COUNT(*), WINDOWSTART, WINDOWEND
FROM mon_stream
WINDOW TUMBLING (SIZE 1 MINUTE)
GROUP BY champ1;
```

Compte le nombre d'événements pour chaque valeur unique de champ1 dans des intervalles d'une minute.

HOPPING Windows.

Les fenêtres **HOPPING** sont similaires aux fenêtres **TUMBLING**, mais elles se chevauchent. Elles ont une durée et un pas (hop).

Utilisation: Utilisez **HOPPING** pour des agrégations où chaque fenêtre a un chevauchement avec la précédente, permettant une vue plus fluide des changements au fil du temps.

Syntaxe:

SELECT <champs_agreges>, **WINDOWSTART**, **WINDOWEND**

FROM <stream/table>

WINDOW HOPPING (SIZE <taille_fenetre>, **ADVANCE BY** <pas_fenetre>)

GROUP BY <champ_groupement>;

HOPPING Windows.

```
SELECT COUNT(*), WINDOWSTART, WINDOWEND
FROM mon_stream
WINDOW HOPPING (SIZE 2 MINUTES, ADVANCE BY 1 MINUTE)
GROUP BY champ1;
```

Compte le nombre d'événements pour chaque champ1 dans des fenêtres de deux minutes, qui avancent d'une minute à chaque fois.

Dimensionnement d’un cluster KAFKA.

Pourquoi dimensionner votre cluster ?

Le dimensionnement d'un cluster Kafka est une étape cruciale pour assurer la performance, la scalabilité et la résilience de votre infrastructure Kafka.

De nombreux facteurs sont à considérer pour un dimensionnement optimal :

- **le volume de données,**
- **le débit des messages,**
- **les SLA,**
- **la tolérance aux pannes**
- **les besoins en matière de réplication.**

Comment déterminer le nombre de brokers ?

Le nombre de brokers (nœuds Kafka) est déterminé par plusieurs facteurs clés :

- **Redondance et haute disponibilité** : Pour assurer une tolérance aux pannes, il est recommandé d'avoir au moins **3 brokers** afin que les répliques des partitions puissent être distribuées entre plusieurs nœuds.
- **Capacité de stockage** : Chaque broker stocke une partie des partitions. Si vous avez des données volumineuses, il est nécessaire d'ajouter plus de brokers pour répartir la charge.
- **Réplication des données** : Plus de brokers sont nécessaires pour permettre des niveaux de réplication plus élevés.

Formule générale :

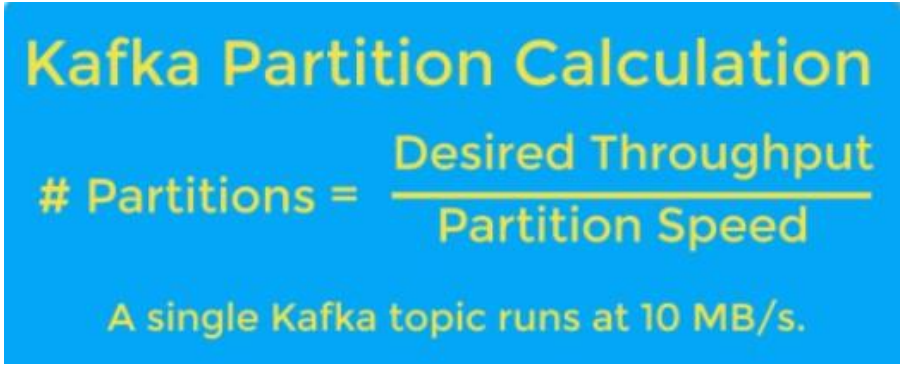
- Si T est la taille totale des données que Kafka doit stocker.
- Si R est le facteur de réplication (exemple, 3 pour la haute disponibilité).
- Si S est la capacité de stockage de chaque broker.

Alors le nombre N de brokers est égal à :

$$N = \frac{T \times R}{S}$$

Nombre de partitions

- Les partitions permettent à **Kafka** de paralléliser la production et la consommation de messages. Le nombre de partitions est un des aspects les plus critiques pour le dimensionnement de **Kafka**.
- En règle générale , **Kafka** recommande de commencer avec **1 000** à **2 000** partitions par broker pour de très grandes installations, mais ce nombre peut varier selon vos besoins.



Nombre de partitions

Par exemple, À titre d'exemple, si le débit souhaité est de **5 To par jour**. Ce chiffre s'élève à environ **58 Mo/s.** En utilisant l'estimation de **10 Mo/s** par partition, cet exemple de mise en œuvre nécessiterait **6 partitions**.

La sécurité avec Kafka

Le chiffrement SSL.

Le chiffrement SSL à pour objectif d'assurer la protection des données en transit entre les clients et les brokers kafka.

Configuration côté serveur (Broker) : Générer un keystore et un trustore pour le broker.

keytool -keystore kafka.server.keystore.jks -alias localhost -validity 365 -genkey

Configurer le Broker Kafka

Dans server.properties, configurez le broker pour utiliser SSL :

Le chiffrement SSL.

listeners=SSL://:9093

ssl.keystore.location=/path/to/kafka.server.keystore.jks

ssl.keystore.password=<keystore-password>

ssl.key.password=<key-password>

ssl.truststore.location=/path/to/kafka.server.truststore.jks

ssl.truststore.password=<truststore-password>

Configurez le client Kafka (producteur ou consommateur) pour utiliser SSL dans ses propriétés:

security.protocol=SSL

ssl.truststore.location=/path/to/kafka.client.truststore.jks

ssl.truststore.password=<truststore-password>

```
sasl.oauthbearer.scope.claim.name = scope
sasl.oauthbearer.sub.claim.name = sub
sasl.oauthbearer.token.endpoint.url = null
security.protocol = PLAINTEXT
security.providers = null
send.buffer.bytes = 131072
session.timeout.ms = 45000
socket.connection.setup.timeout.max.ms = 30000
socket.connection.setup.timeout.ms = 10000
ssl.cipher.suites = null
ssl.enabled.protocols = [TLSv1.2, TLSv1.3]
ssl.endpoint.identification.algorithm = https
ssl.engine.factory.class = null
ssl.key.password = null
ssl.keymanager.algorithm = SunX509
ssl.keystore.certificate.chain = null
ssl.keystore.key = null
ssl.keystore.location = null
ssl.keystore.password = null
ssl.keystore.type = JKS
ssl.protocol = TLSv1.3
ssl.provider = null
ssl.secure.random.implementation = null
ssl.trustmanager.algorithm = PKIX
ssl.truststore.certificates = null
ssl.truststore.location = null
ssl.truststore.password = null
ssl.truststore.type = JKS
value.deserializer = class org.apache.kafka.common.serialization.StringDeserializer
```

Implémentation Java.

Pour configurer un **producteur** ou **consommateur Kafka** en Java pour utiliser SSL, vous devez définir les propriétés SSL dans l'objet Properties utilisé lors de la création de l'instance du producteur ou du consommateur.

Exemple pour un Producteur :

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9093");

props.put("security.protocol", "SSL");

props.put("ssl.truststore.location", "/path/to/kafka.client.truststore.jks");

props.put("ssl.truststore.password", "truststore-password");

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

Implémentation Java.

Exemple pour un Consommateur :

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9093");
props.put("group.id", "test-group");
props.put("security.protocol", "SSL");
props.put("ssl.truststore.location", "/path/to/kafka.client.truststore.jks");
props.put("ssl.truststore.password", "truststore-password");
Consumer<String, String> consumer = new KafkaConsumer<>(props);
```

Sécurisation : L’alternative future à Zookeeper.

Contexte : Kafka utilise **Zookeeper** pour la gestion de cluster et la coordination. Cependant, Kafka travaille à éliminer la dépendance à **Zookeeper** pour simplifier l'architecture et améliorer la sécurité.

KIP-500 : Kafka Improvement Proposal 500 (KIP-500) propose de retirer **Zookeeper** et d'utiliser une architecture interne pour la gestion des métadonnées, réduisant ainsi les vecteurs d'attaque potentiels et simplifiant la configuration de sécurité.

Sécurisation : L’alternative future à Zookeeper.

État Actuel et Future :

Les versions récentes de Kafka incluent des travaux en direction de cette architecture sans **Zookeeper**, mais une migration complète vers **KIP-500** est un processus progressif.

Les administrateurs doivent suivre les recommandations de **Confluent** et de la communauté **Apache Kafka** pour les meilleures pratiques de migration et de sécurisation lors de cette transition.

Les outils autour de Kafka.

Les outils de gestion de Kafka.

Kafka Manager :

Une interface utilisateur web pour gérer les clusters **Kafka**. Permet de visualiser les topics, les partitions, les offsets, et plus.

- Télécharger les sources

  https://github.com/yahoo/CMAK/releases

- Extraire le fichier
- Executer **kafka Manager**

bin/cmak -Dhttp.port=<port> -Dconfig.file=conf/application.conf

Monitoring de Kafka

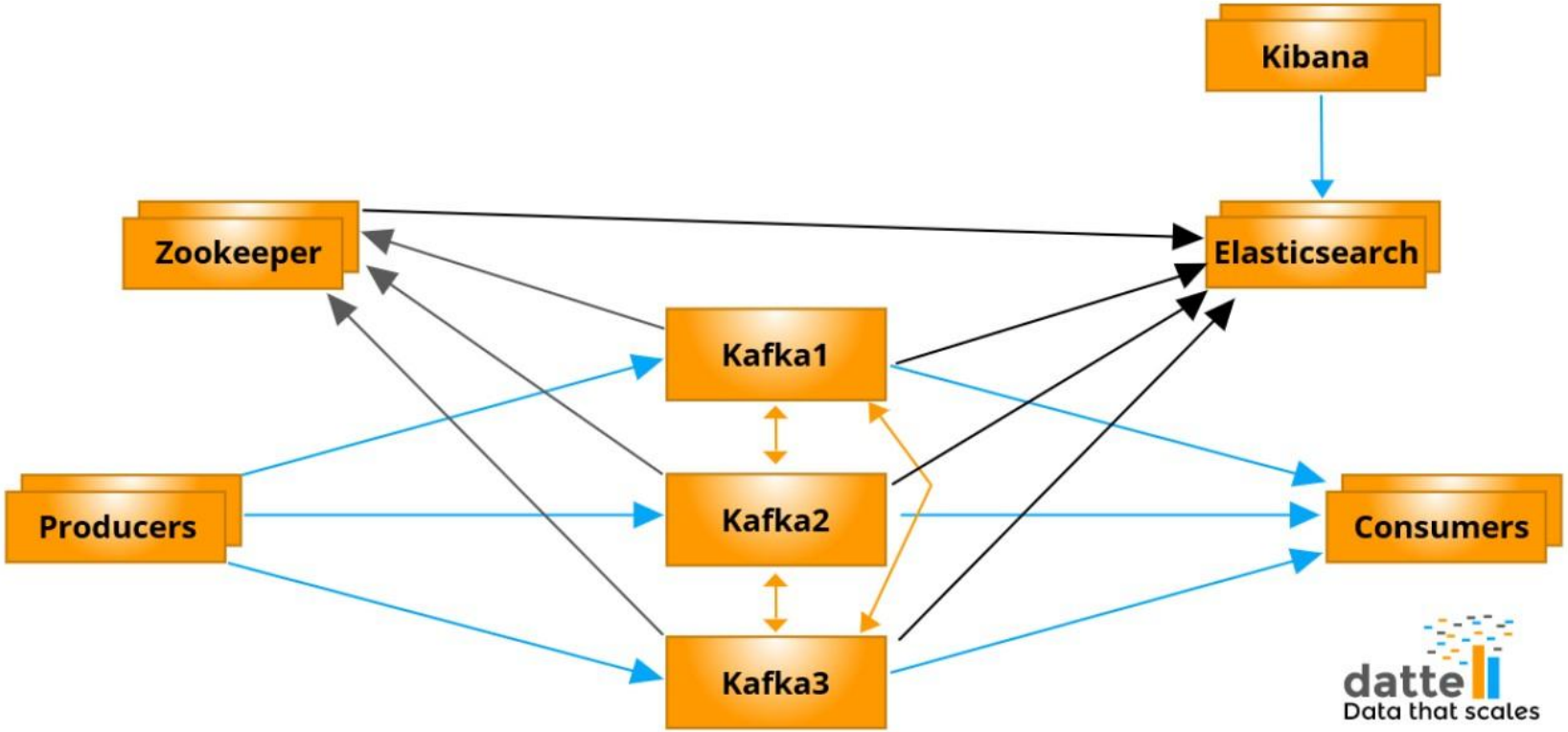
Prometheus et Grafana :

Utilisés ensemble pour monitorer les performances de **Kafka**. **Prometheus** collecte les métriques et **Grafana** les affiche sous forme de tableaux de bord.

JMX (Java Management Extensions) :

Utilisé pour surveiller l'état et la performance des instances **Kafka**. Les métriques **JMX** peuvent être visualisées à l'aide d'outils comme **JConsole** ou intégrées dans d'autres systèmes de monitoring.

Monitoring de Kafka



Apache Avro

Apache Avro :

Un système de sérialisation des données qui est largement utilisé avec Kafka pour la sérialisation des messages. Il prend en charge la définition de schémas riches et est connu pour être compact et rapide.

https://avro.apache.org/

Le “Schéma Registry”

Schéma Registry :

Fournit un service de stockage centralisé pour les schémas **Avro**. Cela aide à assurer que la structure des messages **Kafka** reste cohérente et compatible avec le temps, et facilite l'évolution des schémas.

Présentation de la plateforme Confluent

La plateforme **Confluent** est une distribution de **Kafka** qui étend ses fonctionnalités et offre des outils et services supplémentaires. Les différences avec Kafka open-source incluent :

KSQL et **Stream Processing** :

Confluent propose **KSQL**, un langage de requête de **streaming SQL** pour Kafka, facilitant la création d'applications de traitement de streams.

Présentation de la plateforme Confluent.

Contrôle d'Accès et Sécurité :

Confluent offre des fonctionnalités avancées de sécurité et de contrôle d'accès, telles que le chiffrement des données en transit et au repos, et un contrôle d'accès basé sur les rôles.

Connecteurs Intégrés :

Confluent fournit une large gamme de connecteurs **Kafka** prêts à l'emploi pour intégrer avec divers systèmes et bases de données.

Conclusion.