**APACH KAFKA**

Table des matières

[Chapitre 1 INTRODUCTION 3](#_Toc102327204)

[Chapitre 2 CONCEPTS CLES : 3](#_Toc102327205)

[Chapitre 3 PIPELINE D’ECRITURE DE MESAGES : 4](#_Toc102327206)

[Chapitre 4 PARAMETRES DE PRODUCERS KAFKA 5](#_Toc102327207)

[Chapitre 5 MODES DE PRODUCTION : 6](#_Toc102327208)

[a-Production bloquante : 6](#_Toc102327209)

[b-production non bloquante : 6](#_Toc102327210)

[Chapitre 6 CONSUMERS KAFKA : 6](#_Toc102327211)

[Chapitre 7 COMMIT DES MESSAGES CONSOMMES : 7](#_Toc102327212)

[Chapitre 8 STATELESS VS STATEFULL : 8](#_Toc102327213)

[a- Stateless : 8](#_Toc102327214)

[b- Statefull : 8](#_Toc102327215)

[Chapitre 9 MESSAGE DELIVERY : 8](#_Toc102327216)

[a- At Least Once : 8](#_Toc102327217)

[b- At Most Once : 9](#_Toc102327218)

[c- Exactly Once : Producer Idempotence: 9](#_Toc102327219)

[Chapitre 10 KAFKA Streams : 10](#_Toc102327220)

[1-Architecture de Kafka STREAMS : 11](#_Toc102327221)

[12](#_Toc102327222)

[2-Configuration de KAFKA STREAMS : 12](#_Toc102327223)

[Chapitre 11 KAFKA CONNECT: 12](#_Toc102327224)

[1-Contexte de Kafka Connect & définition : 12](#_Toc102327225)

[2-Architecture : 14](#_Toc102327226)

[3-Fonctionnement de Kafka Connect : 15](#_Toc102327227)

# 

# INTRODUCTION

Brocker = serveur = intermédiaire,

Kafka sert à envoyer des messages entre deux ou plusieurs services (applications) d’une entreprise. Kafka est comme une boite aux lettres il nous met à disposition un message et attend qu’on le consomme.

Kafka peut être un service indépendant de hadoop on parle à ce moment-là de confluente. Kafka ne stocke que du binaire donc il sérialise chaque objet.

Kafka est très convoité pour ses performances surtout dans le big data quand on parle de l’évènementiel (temps réel). Stockés des milliers voire des millions de messages en temps réel (exemple de LinkedIn en 2014 avec un pic de 200 millions de messages en seconde).

# CONCEPTS CLES :

Kafka cluster :1 ou plusieurs brockers coordonnés via zookeeper.

Brocker : 1 instance de kafka (1 seule machine).

Les producteurs (producers) : écrivent des messages sur des brockers.

Les consommateurs (consumers) : lisent des messages sur des brockers.

Les messages sont écrits dans des topics qui sont divisés en partition.

Topic = une ligne de messages qui sont de même nature. (Nom d’un flux sur lequel les producers écrivent).

Une partition est répliquée sur plusieurs brockers.

Un groupe de consommateurs (consumers group = plusieurs consommateurs) s’occupent 1 seul topic et chaque consumer s’occupe de d’1 seule partition.

***Consumers group***

Consom 1

Consom 2

Consom 3

Consom 4

**TOPIC**

producer

producer

***Producer***: les messages sont publiés à la fin d’une partition.

La position d’un message dans un topic appelé ***offset***.

***Consumer :*** Consomme les messages en conséquence.

***Conception :*** Le topic n’existe qu’à travers les messages. Chaque message écrit sur une partition sera indexé par un offset.

Une partition est une séquence ordonnée et immutable de messages (une liste de messages infinis).

Les messages sont ajoutés uniquement à la fin d’une partition.

Le nombre de partitions d’un topic est configurable

Le nombre de partitions détermine le degré de parallélisme d’un topic.

2 consumers du même groupe s’occupent de deux partitions.

***Réplication :***

**Topic1**

Part 1

T**opic1**

**Topic1**

Part 2

Part 3

Brocker 1 brocker 2 brocker 3

***Un réplica*** dans kafka est un ***follower*** car ***le leader*** reçoit continuellement des messages (en temps réel) donc les followers ne font que suivre et se mettent à jour au fur et à mesure (les followers deviennent alors des consommateurs kafka du leader)

Si un brocker tombe en panne, on élira alors le brocker qui est le plus à jour (à l’aide de l’ISR qui est un paramètre de l’évaluation de degré de mise à jour de chaque réplica dans un brocker).

Le nombre de réplica < au nombre de brockers (si on a 4 brockers dans l’idéal est d’avoir 3 réplica

# PIPELINE D’ECRITURE DE MESAGES :

Le leader prend une décision en fonction des ISR (ISR : In Synchron Replica : nombre de réplicas qui sont à jour (synchrones) avec le leader).

Il peut être défini par le nombre de messages sur lesquels il est en retard (ou estimation du temps de retard qu’il a avec le leader).

Si par malheur un follower n’arrive pas à suivre et à se mettre à jour (pour cause des latences réseaux par exemple) alors le leader peut nommer un autre brocker pour devenir follower.

# PARAMETRES DE PRODUCERS KAFKA

Les clients de kafka sont les différents langages de programmation : java, python, scala, etc

Les messages sont au format (key, value). {map}

***Key :*** le hash de la clé détermine la partition qui recevra le message : en cas d’absence de clé kafka va faire du **Round-Robing (aléatoirement)**.

***Value :*** Présente le message, elle est stockée sous forme d’une série de byte. (Cette valeur peut être de n’importe quel type sérialisable). brocker.one : 9092 =numéro de port de l’écoute de kafka.

1. ***Acks*** : acknowledgement (accusé de réception) :

On le met à un🡺 quand on veut un accusé de réception du leader.

On le met à 0🡺 quand on n’a pas besoin d’un accusé de réception.

On le met à -1🡺 quand on a besoin d’un accusé de réception du leader et de chaque fellower qui le suit

***Use case*** :

On met ***Acks= 0*** c’est le cas des times series (températures par exemple) parceque si on perd une valeur d’une température ce n’est pas grave (la température ce n’est pas une valeur qui change rapidement dans le temps donc si on perd une valeur de température cela n’est pas si grave)

Par contre on le met ***Acks = -1 le*** quand les données sont hyper importantes (ça génère beaucoup de latences réseaux) mais l’importance de la donnée (message) impose que le Producer doit s’assurer de la bonne réception du message de la part du leader et tous ses fellowers.

1. ***Sérialisation des messages :*** key.serializer, value.serializer
2. ***Batching des messages :*** pour contrôler les performances du producer
   * 1. ***Batch.size :*** (methode send)limite le nbre de message à batcher.
     2. ***Buffer.memory :*** limite la taille des messages
     3. ***Linger.ms :*** limite le temps d’attente du producer avant le renvoi du batch.

***NB :*** finalement le producer n’envoie pas directement au brocker mais il passe d’abord par un buffer dont la taille est gérable par la commande suscitée.

1. ***Compression :*** (on peut décider de ne pas compresser et matcher le producer avec consumer) compression.type : les codes supportés sont : gzip, izt, snappy, uncompressed
2. ***L’ordre des messages :*** l’ordre des messages est uniquement dans une partition est uniquement dans une partition pour s’assurer que des messages seront consommés dans leur ordre de production. Il faut qu’ils aient tous la même clé
3. ***Renvoi automatique de message en cas d’echec :***
   * 1. ***retries***: le nombre de fois un message est renvoyé par le producer en cas d’erreur (d’échec).
     2. ***Request.timeout.ms :*** le temps pour timeout une requête.
     3. ***Retry.backoff.ms :*** le temps à attendre avant de renvoyer un message.

Il existe un risque de changement d’ordre des messages si retries >0. Pour garantir l’ordre il faut taper la commande suivante : ***max.inflight.request.per.connection = 1*** (il y a une seule connexion = 1 le deuxième ne passe pas sauf si le message est passé sans échec) par contre cette manipulation crée un goulot d’étranglement (bouchon).

# MODES DE PRODUCTION :

## a-Production bloquante :

quand on met le acsk différent de 0 ( 1 ou -1) : producer.send(record).get

## b-production non bloquante :

producer.send(record,new callback….);

***NB :*** Le nombre de partitions qui rend kafka scale-out. Le nombre d’applications qui produisent et qui consomment des messages et la vitesse de production et de consommation qui définissent le nombre de réplicas qu’il faut pour kafka.

# CONSUMERS KAFKA :

Il existe des consumers dans plusieurs langages de programmation : java, c++, python, ruby etc…

Chaque consumer à chaque fois qu’il consomme un message il commit (soit au broker soit à zookeeper)(le commit = dire à kafka que j’ai consommé un message) :

2 API kafka consumers pour java/scala :

* Old consumer API (zookeeper qui met à jour les offsets de consommation).
* New consumer API : existe pour la version kafka >= 0.9 : les offsets sont gérés par un topic kafka

Si on veut créer un groupe de consumers pour un seul topic donné il suffit de de mettre le paramètre props.put(« group.Id », « test ») (test est le nom du topic qu’on consomme)

Un consumer peut souscrire à consommer plusieurs topics.

* Avanages :
* Les consumers peuvent reprendre la consomamtion à partir de n’importe quel ofset.
* Les consommateurs peuvent choisir les partitions à lire.

***Batcher la consommation :*** lancer le consumer par intervalle de temps

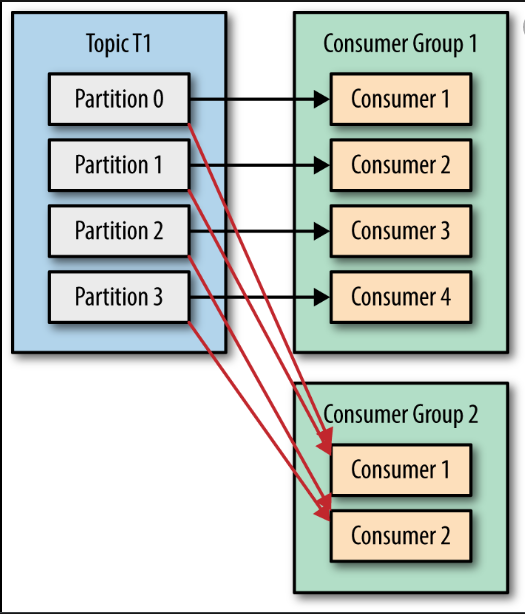
Les consommateur kafka fonctionnement en mode pull

C’est le consommateur qui ramène les messages du broker quand il le souhaite.

Avantages :

* Chaque consumer consomme à son ritjme
* Améliorer la stabilité des applications en aval du consumer.(les applications consommatrices conçues pour 1 charge normale pas pour des pics de consommation.

Chaque consommateur appartient à un groupe de consumers.



# COMMIT DES MESSAGES CONSOMMES :

Selon le use case on peut choisir :

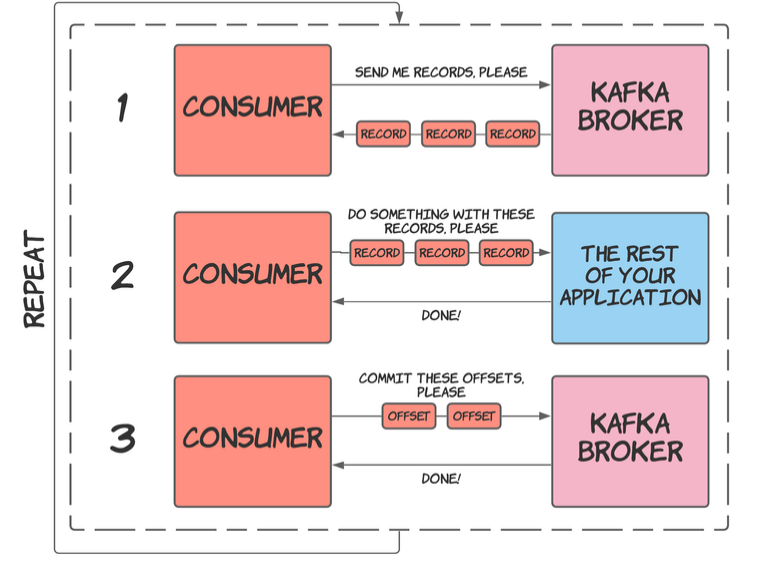
1. Commit automatique (automatic commit) : la gestion du commit est laissée au consommateur.
2. Commit manuel : le commit des message est à la charge du développeur (gérer dans le code) : prop.put (‘’enable.auto.comit’, false) 🡺 manuel.

Avantage des commit automatiques : Simplifie le code

Inconvénient du commit automatique : risque de perte de messages.

Avantage du commit manuel : maitrise totale du cycle de consommateur

Inconvénient : risque de double consommation (suite à des erreurs réseaux.) =>(zookeeper n’a pas l’instruction synchronisée avant le commit)



# STATELESS VS STATEFULL :

(upsert : update et insert au même temps)

## a- Stateless :

Chaque message est atomique : (pas de relation entre les messages)

Exemple : calcul de tva sur chaque produit

Mettre des strings en majuscule

Pas besoin de connaitre les anciens messages pour pouvoir effectuer un traitement

## b- Statefull :

le contraire de stateless : il faut connaître les anciens messages pour pouvoir effectuer un traitement.

Exemple calcul de chiffres d’affaires en faisant la somme de toutes les ventes. (chaque message kafka est une vente)

# MESSAGE DELIVERY :

## a- At Least Once :

Pas de perte de message mais on accepte de dupliquer deux fois.

Cette méthode on l’utilise surtout dans l’insertion dans les bases de données relationnelles. C’est le mode de kafka par défaut.

## b- At Most Once :

On accepte de perdre les messages par contre la duplication de messages n’est pas acceptables et la perte de messages est tolérée.

Kafka : configurer le Producer sans retry (retries =0)

## c- Exactly Once : Producer Idempotence:

le message est reçu par les consumers une et une seule fois. (la duplication et la perte de messages ne sont pas acceptées)

Pour s’y faire il faut avoir une coopération entre les producers et les consumers & de kafka.

Ça nous offre une garantie de traiter chaque message une et une seule fois.

Cette option n’est pas offerte par tous les systèmes de traitement en temps réel.

Pour avoir cette configuration il est conseillé d’utiliser l’API Kafka STREAMS.

La duplication des messages est due :

* la non réception d’ack par le producer.
* Double consommation d’1 message due au crash d’1 consumer après processing du message mais avant de le commiter.
* Duplication de production / process zombies : elles peuvent apporter pour cause de coupure temporairement de réseaux ce qui provoquera le lancement de nouvelles instances. => deux instances font les mêmes taches => duplication du message.

Configiration : enable.idempotence = true.

Avantages :

* Pas de réplication de messages due à un renvoi pour cause d’indisponibilité d’un broker ou d’erreurs de connexion
* Respect d’ordre’ d’envoi des messages.

***NB :*** Les messages sont toujours envoyés d’une façon sérialisée par les producers.

***La serialisation :*** est le processus qui vise à créer et à transformer un objet en mémoire d’une façon optimale (compacte et rendre en série binaire) :

1. Quand on envoie sur le réseau
2. Quand on envoie sur le disk ;

***Transient :*** c’est les attributs exclus de la sérialisation d’un objet (les méthodes d’un objet ne font pas partie car elles décrivent un comportement de l’objet et elle n’est chargée qu’une seule fois en mémoire d’opérations).

***GCP :*** Pop sub = kafka (équivalent de kafka dans google cloud platform)

# KAFKA Streams :

Le but est de traiter ou d’analyser les données présentes dans Kafka, puis de les faire ressortir dans un nouveau topic Kafka ou dans un service externe (une base de données, par exemple). Disponible depuis la version 0.10 de Kafka

D’un point de vue purement technique, Kafka Streams :

* Est une bibliothèque Java. Rien de plus, rien de moins ! On n’a donc pas besoin d’un cluster s’inspire d’Apache Samza, mais en plus léger
* Est une bibliothèque de transformation et de traitement de données qui tourne sur Kafka
* Est une bibliothèque de traitement de données transactionnelle (traitement unitaire de données avec mécanismes de garantie de sémantique, et de cohérence ACID)

On peut trouver quasiment toujours Kafka Streams en entreprise dans les cas d’usage suivants :

* Traitement des transactions financières et des paiements bancaires en temps réel (CB, bourse, banques, assurances, détection de fraude, etc…)
* Monitoring logistique en temps réel (tracking de véhicule, avions, flottes, parcs automobiles, industries, etc..).
* Traitement des objets connectés et tous les cas d’usage impliquant l’usage des capteurs.
* Monitoring en temps réel de patients dans les hôpitaux.
* Gestion des notifications en temps réel dans les applications mobiles
* Gestion des programmes de fidélité, campagnes marketing, processus opérationnels (réservation commandes en e-commerce et en grande distribution, hôtels, etc.)

Grâce à la combinaison de ce découplage aux fonctionnalités transactionnelles (**ACID**, garanties de sémantique), Kafka possède aujourd’hui toutes les caractéristiques pour être utilisé comme un Hub de données transactionnel (**Transactional Data Hub**), c’est-à-dire un hub dans lequel on peut gérer des cas d’usage streaming opérationnels, et pas seulement des cas d’usage décisionnels.

Il est possible aujourd’hui d’implémenter dans Kafka des cas d’usages où chaque opération doit respecter des contraintes sémantiques et référentielles définies par les business analysts pour être validées par le système. C’était pour cette raison qu’on utilisait et qu’on utilise encore les SGBDR.

Kafka a toutes les caractéristiques aujourd’hui pour ***agir comme un SGBDR à large échelle*** pour des cas d’usage streaming métiers ou opérationnels. D’où l’intérêt de Kafka Streams.

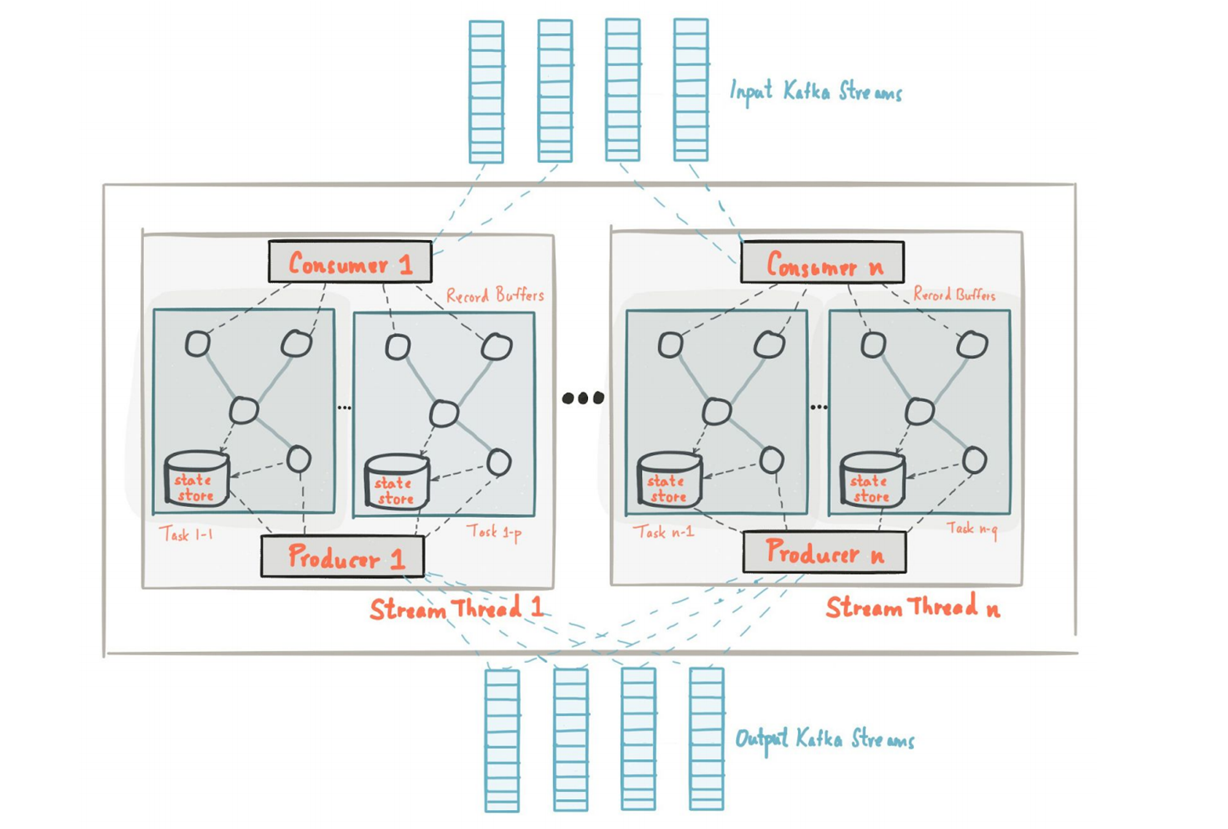
***Kafka Streams est une API streaming transactionnelle***. Elle permet de traiter les données unitairement, une à la fois directement depuis les topics Kafka.

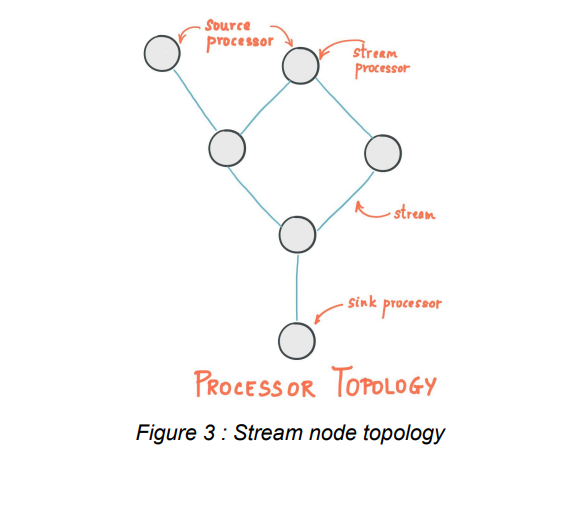
Kafka Streams comble parfaitement les limites d’un ***consumer classique***. Là où un consumer ne sait que lire la donnée sans pouvoir y effectuer des traitements réutilisables (par exemple : récupérer les données d’un topic, tester la validité des données, et publier les valides dans un autre topic, et les non-valides dans un autre), ***avec Kafka Streams, on peut appliquer des règles de validation à chaque donnée***, et même effectuer des opérations complexes tels que les agrégations***, les jointures, ou encore la gestion des cas complexes.***

Kafka Streams comble également les limites des ***« consumers décisionnels*** » tels que Spark Streaming ou Apache Flink. Là où ceux-ci sont obligés d’effectuer des traitements de données sur des ***micro-batch***, avec Kafka Streams on effectue des traitements unitaires par défaut, et on a le choix entre **ce mode de traitement** et le ***traitement groupé***.

Là où la mise en œuvre de Storm demande la mise en place d’une architecture ***lambda complexe***, l’utilisation ***de Kafka Streams nécessite juste un PC***, car elle est une API très légère qui repose sur un cluster Kafka. ***Kafka Streams s’y connecte simplement comme client***.

## 1-Architecture de Kafka STREAMS :





**Figure à gauche** :

* L’avantage d’un processeur de flux c’est qu’il peut être totalement distribué, ou lancer de façon multi-threadé ce qui permet de traiter des données très rapidement.
* Une application stream processor peut être organisée avec différent nœuds. Chaque nœud correspond à une transformation des données et sont donc reliés par des flux de données.
* Le premier noeud ***(source processor***) est celui qui va récupérer le flux provenant des consommateurs et la transmettre aux autres nœuds. Le dernier noeud (***sink processor***) est celui qui va récupérer le flux final et le transmettre au producteur

## 

Afin d’effectuer différentes transformations, l’API Stream propose deux classes : ***KStream et KTable.***

1. La classe KStream représente les données sous forme d’un flux continue sans interruption.

Les données sont sous la forme d’une paire (clé, valeur).

1. La classe KTable est une représentation compactée du flux de donnée qui garde seulement la dernière valeur.

## 2-Configuration de KAFKA STREAMS :

Pour configuration & cas d’usage :

1. <https://blog.ippon.fr/2017/04/18/kafka-streams-101/>
2. <https://docs.confluent.io/platform/current/streams/code-examples.html> (WordCount)

# KAFKA CONNECT:

## 1-Contexte de Kafka Connect & définition :

Le développement d’applications streaming est particulier et dans la majorité des cas, les données [streaming] qu’on traite existeront déjà dans des systèmes de gestion de base de données de l’entreprise (Oracle, SQL Server, IBM DB2, MySQL, Ms Access, Ms Excel, etc), dans ses ERP métiers (Salesforce, SAP), ou encore dans ses systèmes décisionnels (HDFS, Data warehouse, Teradata, Hana, etc) ou même seront hébergées dans les PaaS de ses fournisseurs Cloud.

Cela signifie que qu’on aura rarement à développer un Producer par programmation en partant de rien. On va souvent devoir plutôt définir une source existante comme un Producer Kafka.

Le problème le plus évident qu’on rencontrera pour faire apparaître ces sources de données opérationnelles comme Producer Kafka est que pour la plupart, elles ne sont pas nativement streaming. En d’autres termes, les données qui y sont stockées ne présentent aucune caractéristique streaming et sont stockées comme de simples ‘’faits’’ . De plus, chaque système opérationnel de base de données possède des caractéristiques techniques et technologiques qui lui sont propres.

Ainsi, pour exposer Oracle comme Producer à Kafka, il faut développer un « Producer Oracle » spécifique, pour Salesforces, il faut développer un « Producer Salesforces« , pour Hadoop, il faut développer un « Producer Hadoop« , etc…

Pour résoudre ces 2 problèmes d’un point de vue conceptuel, la solution revient à faire ceci :

1. Développer un ***connecteur générique*** côté Kafka compatible avec le protocole de connexion de données utilisé par les systèmes opérationnels de gestion de données (les SGBDR), notamment le protocole ODBC ou le protocole JDBC, puisque les SGBDR sont quasiment tous compatibles avec l’un de ces 2 protocoles. Ce connecteur, à l’aide de la connexion qu’il établira avec le SGBDR pourra envoyer des appels/requêtes de copie de données vers Kafka, tout comme les applications opérationnelles récupèrent les données de la base de données à travers les requêtes SQL qu’elles envoient au SGBDR.
2. Mettre ***le connecteur générique*** en écoute de la base de données de sorte que Kafka soit informé de tout changement ou modification de données dans la base. Ainsi, toute la base de données est changée de faits en événements (donc en source de données streaming).

C’est exactement cette solution qui a été adoptée par les développeurs de Kafka dès sa version 0.9 avec la sortie de Kafka Connect.

Kafka Connect est une extension de Kafka qui établit une passerelle entre une grande variété de systèmes opérationnels (tels que les SGBDR, les ERP, les data warehouse, les outils de journalisation) et le Log de Kafka afin d’y copier/transférer les données. Cela signifie qu’on peut utiliser Kafka Connect pour établir une connexion avec Salesforce et l’exposer comme Producer, afin de récupérer chaque nouvelle donnée client qui y arrive et l’enregistrer dans Kafka pour un usage immédiat ou différé.

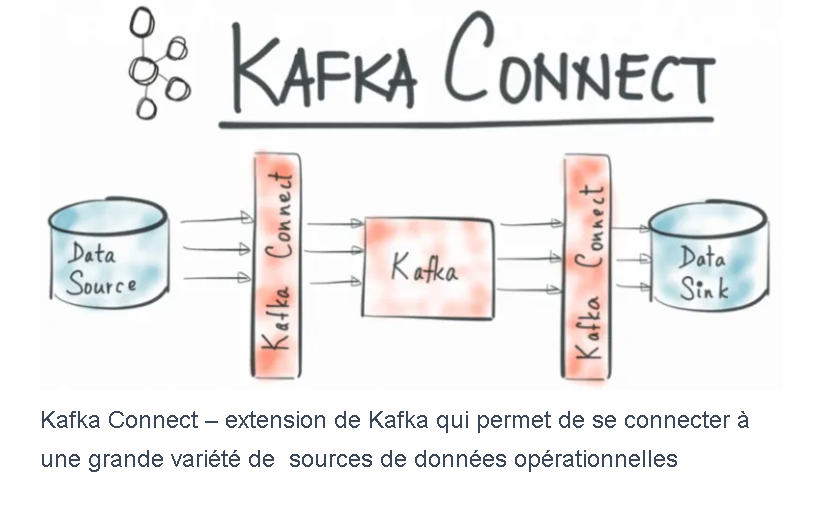
Aussi, on peut utiliser Kafka Connect pour récupérer les données stockées dans Kafka et les pousser vers une destination (un sink) par exemple le data-lake, le data-warehouse de l’entreprise, Hadoop, HDFS, ou encore une autre base de données relationnelle. Kafka Connect transforme toute source de données opérationnelles (les faits) en source de données streaming (des événements), ce qui favorise l’usage de Kafka à un grand nombre de scénarios de données.

Par exemple :

* L’ingestion en temps réel de tout changement dans une base de données relationnelle
* L’ingestion centralisée des données collectées par des agrégateurs de flux tels que Log4J, SysLog pour la détection d’anomalies
* Le reporting en temps réel par l’ingestion des données de systèmes opérationnels vers des Data Hub (Data Lake, Data warehouse)
* Ingestion de données pour la construction d’index d’un moteur de recherche de contenu (elasticsearch, Solr, etc).

En fait, Kafka Connect peut être utilisé pour établir une connexion avec tout système qui est compatible avec le protocole JDBC. De plus, il s’utilise comme un service ***REST***. Cela signifie que la création, la configuration et la suppression d’une connexion avec un système opérationnel se fait à l’aide d’une simple API REST. Nous y reviendrons plus bas.

## 2-Architecture :

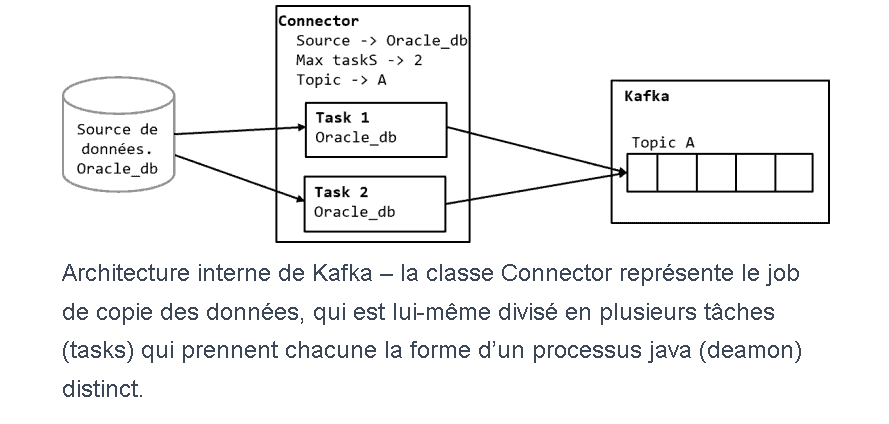


L’architecture de Kafka Connect repose sur 3 grands concepts. En d’autres termes, vous devez apprivoiser uniquement 3 concepts pour comprendre et maîtriser l’utilisation de Kafka Connect : le ***connecteur (connector), la tâche (task) et le worker.***

1. ***Le Connecteur (Connector)*** : c’est l’instance logique d’un job Kafka Connect qui gère la copie/transfert des données d’un système source (source system) vers un système cible (sink system). En fait, il fait référence à une classe appelée Connector dans laquelle on spécifie la source de données copiée, ainsi que différentes configurations.
2. ***La tâche (Task)*** : chaque connecteur instancie une ou plusieurs tâches pour l’exécution de la copie des données. Les tâches sont les instances logiques qui exécutent la copie/transfert de données du système source vers Kafka ou de kafka vers le système cible. Les tâches sont en réalités l’exécution distribuée de la copie des données (du job Kafka Connect – Connector).
3. ***Les workers*** : le connecteur est le job qui copie, les tâches sont les instances logiques de l’exécution de la copie, et les workers sont en fait les processus informatiques (***deamon***) qui s’exécutent sur le cluster pour copier les données. Les workers ce sont les instances physiques des tâches***. worker = connector + task***

Les processus workers vont s’exécuter en distribué de façon coordonnée. Par contre, il faudra un gestionnaire de ressource pour la gestion et le monitoring de leur exécution (par exemple YARN).

La figure suivante récapitule l’architecture interne de Kafka Connect avec l’exemple d’une source de données, Oracle. L’utilisation de Kafka Connect revient à déclarer/supprimer les Connecteurs (indiquer la source de données à copier, les paramètres de connexion à cette source, le nombre de tâches, le topic de destination, …etc) et les configurer via un fichier de configuration (mode standalone), ou via une API REST.



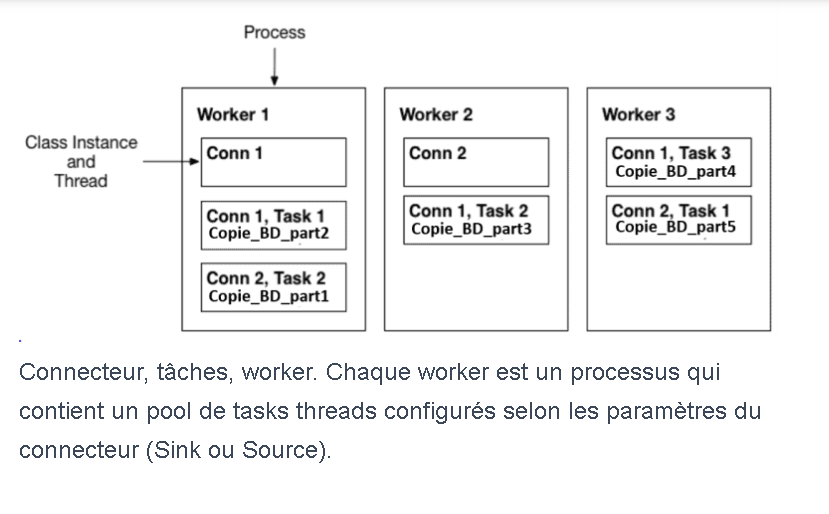
## 3-Fonctionnement de Kafka Connect :

Le fonctionnement de Kafka Connect commence avec le démarrage des processus Kafka Connect sur les machines du cluster Kafka (les workers). Chaque worker tourne sur un noeud différent du cluster.

Chaque instance de worker démarre avec comme configuration minimale : l’adresse d’un agent Kafka, le topic de stockage des données à copier et le ***Kafka\_consumer\_group\_id***. Chaque instance de worker est sans état ***(stateless***) et ne partage aucune information (état) avec les autres instances worker. Par contre, toutes ces instances appartiennent au même ***Kafka\_consumer\_group\_id*** et se coordonnent à l’aide d’un topic Kafka.

Chaque worker instancie ***un Connecteur (source ou cible)*** avec les configurations de ***la tâche*** associée (***tâche source ou tâche cible***). Comme expliqué précédemment, ***la tâche*** est le processus logique de copie des données. Par exemple, une tâche ***HDFSSinkTask*** est un thread qui copie les données d’un topic Kafka pour les déposer dans le HDFS.

Chaque worker instancie ***un pool de tâches*** sur la base de la configuration fournie par le Connecteur et décide en coordination avec les autres workers sur la base du ***topic kafka***, quelles données vont être traitées par chaque tâche.



***Attention !!!*** la coordination distribuée des workers et des différentes tâches est assurée par Kafka. En cas d’arrêt inopiné d’un worker par exemple, le rééquilibrage automatique de charge est pris en charge par Kafka.

Kafka Connect expose ***une API REST*** pour la création, la modification et la suppression des connecteurs. Nous vous montrerons comment l’utiliser plus bas.

Quel que soit le Connecteur utilisé et qu’il soit Source ou Cible, ce qui se passe est qu’à des intervalles réguliers (par exemple toutes les 30 secondes), les données sont copiées du système source par les tâches et écrites dans le système cible. Notez que l’un des avantages avec Kafka Connect est qu’il gère automatiquement les offsets. Il les stocke par défaut dans un topic Kafka. Ainsi, vous n’avez pas à gérer les offsets lorsque vous développez vos connecteurs.

Cela signifie qu’il y’a un commit de l’offset automatiquement après la copie des données et/ou après leur dépôt dans le système cible. Grâce à cette gestion automatique des offset, Kafka Connect sait toujours où il en est dans les copies/transferts de données, et celles-ci sont par ***conséquent idempotents***. En cas d’arrêt d’un worker ou de panne sur le cluster, l’offset est rejoué par Kafka Connect pour récupérer l’état de la copie avant la panne lors de la restauration. Il n’y’a donc jamais une double écriture/copie de la même donnée. Ainsi, les opérations de copie/transfert de données de Kafka Connect sont ***idempotentes et cela garantie*** la sémantique exactement-une-fois.

Pour la copie des données des systèmes opérationnels (SGBDR), Kafka est compatible avec le protocole JDBC (Java Data Base Connectivity). Le JDBC, tout comme l’ODBC (Open Data Base Connectivity), est un protocole générique de données qui permet de communiquer avec des SGBDR. Pratiquement tous les SGBDR (Oracle, SQL Server, Access, DB2, ...etc) sont compatibles avec les protocoles JDBC et ODBC. Ainsi, en étant compatible avec le protocole JDBC, Kafka Connect est capable de se connecter à n’importe quel SGBDR et l’exposer soit comme un producer Kafka (copier les tables de la BD et la transférer vers un topic Kafka), soit alors comme un Consumer Kafka (copier les données d’un topic Kafka pour les stocker dans une BD relationnelle).

Le processus est le même que celui décrit plus haut : les données sont chargées à des intervalles périodiques via l’exécution d’une requête SQL. Par défaut, toutes les tables dans la base de données sont copiées, chacune dans un topic distinct. Kafka se met en écoute de la base de données pour y détecter tout changement (par exemple création de nouvelles tables, suppression d’enregistrements, ajout de nouveaux enregistrements dans une table, etc.) et s’adapter automatiquement (réplication des changements dans les topics correspondants). Chaque copie ou transfert des résultats de la requête SQL entraîne automatiquement un commit de l’offset.

Attention !!! Lorsque on place Kafka en écoute d’une BD relationnelle, on doit indiquer le champ ou la colonne qui doit être écouté (ou dont les changements entraîne les changements dans le topic). C’est cette colonne qui servira de référence pour l’envoie de nouvelles données.

POUR PLUS D’INFOS (le code et exemples :) SITE SUIVANT :

1. <https://www.data-transitionnumerique.com/consommation-donnees-kafka-connect/>
2. <https://blog.octo.com/kafka-streams-encore-un-framework-de-stream-processing/#:~:text=Les%20KTables%20sont%20des%20Streams,si%20la%20cl%C3%A9%20est%20nouvelle>.

***Spring***: En informatique, Spring est un framework open source pour construire et définir l'infrastructure d'une application Java3, dont il facilite le développement et les tests.

***Conteneur léger***

Spring est considéré comme un conteneur dit « léger ». La raison de ce nommage est expliquée par Erik Gollot dans l’introduction du document Introduction au framework Spring5.

« Spring est effectivement un conteneur dit « léger », c’est-à-dire une infrastructure similaire à un serveur d'applications J2EE. Il prend donc en charge la création d’objets et la mise en relation d’objets par l’intermédiaire d’un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d’application est qu’avec Spring, les classes n’ont pas besoin d’implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications J2EE et des EJBs). C’est en ce sens que Spring est qualifié de conteneur « léger ». »

Spring s’appuie principalement sur l’intégration de trois concepts clés :

* L’inversion de contrôle est assurée de deux façons différentes : la recherche de dépendances et l'injection de dépendances ;
* La programmation orientée aspect ;
* Une couche d’abstraction.

***La couche d’abstraction*** permet d’intégrer d’autres frameworks et bibliothèques avec une plus grande facilité. Cela se fait par l’apport ou non de couches d’abstraction spécifiques à des frameworks particuliers. Il est ainsi possible d’intégrer un module d’envoi de mails plus facilement.

***L’inversion de contrôle :***

***La recherche de dépendance :*** consiste pour un objet à interroger le conteneur, afin de trouver ses dépendances avec les autres objets. C’est un cas de fonctionnement similaire aux EJBs ;

L’injection de dépendances : cette injection peut être effectuée de trois manières possibles :

* L’injection de dépendance via le constructeur,
* L’injection de dépendance via les modificateurs (setters),
* L’injection de dépendance via une interface.

Les deux premières sont les plus utilisées par Spring.

Ce framework, grâce à sa couche d’abstraction, ne concurrence pas d’autres frameworks dans une couche spécifique d’un modèle architectural Modèle-Vue-Contrôleur mais s’avère être un framework multi-couches pouvant s’insérer au niveau de toutes les couches ; modèle, vue et contrôleur. Ainsi il permet d’intégrer Hibernate ou iBATIS pour la couche de persistance ou encore Struts et JavaServer Faces pour la couche présentation.

***Le noyau*** de Spring est basé sur :

* Une fabrique générique de composants informatiques, composants nommés ***beans*** (anglais de haricots, et dans le contexte Java de grain de café) ;
* Un conteneur capable de stocker ces beans.

De plus, le noyau de Spring permet de forcer le contrôle de ces composants de leur extérieur, par la technique nommée inversion de contrôle.

Le principal avantage est de composer les beans de façon plus déclarative plutôt que de façon impérative dans le programme. Les beans peuvent être définis par le biais de fichiers de configuration en Java ou XML

***Définition  HUB de données (DATA HUB):***

<https://www.data-transitionnumerique.com/data-hub-definition/>

***Propriétés ACID :*** (Atomicity, Consistency, Isolation, Durability)