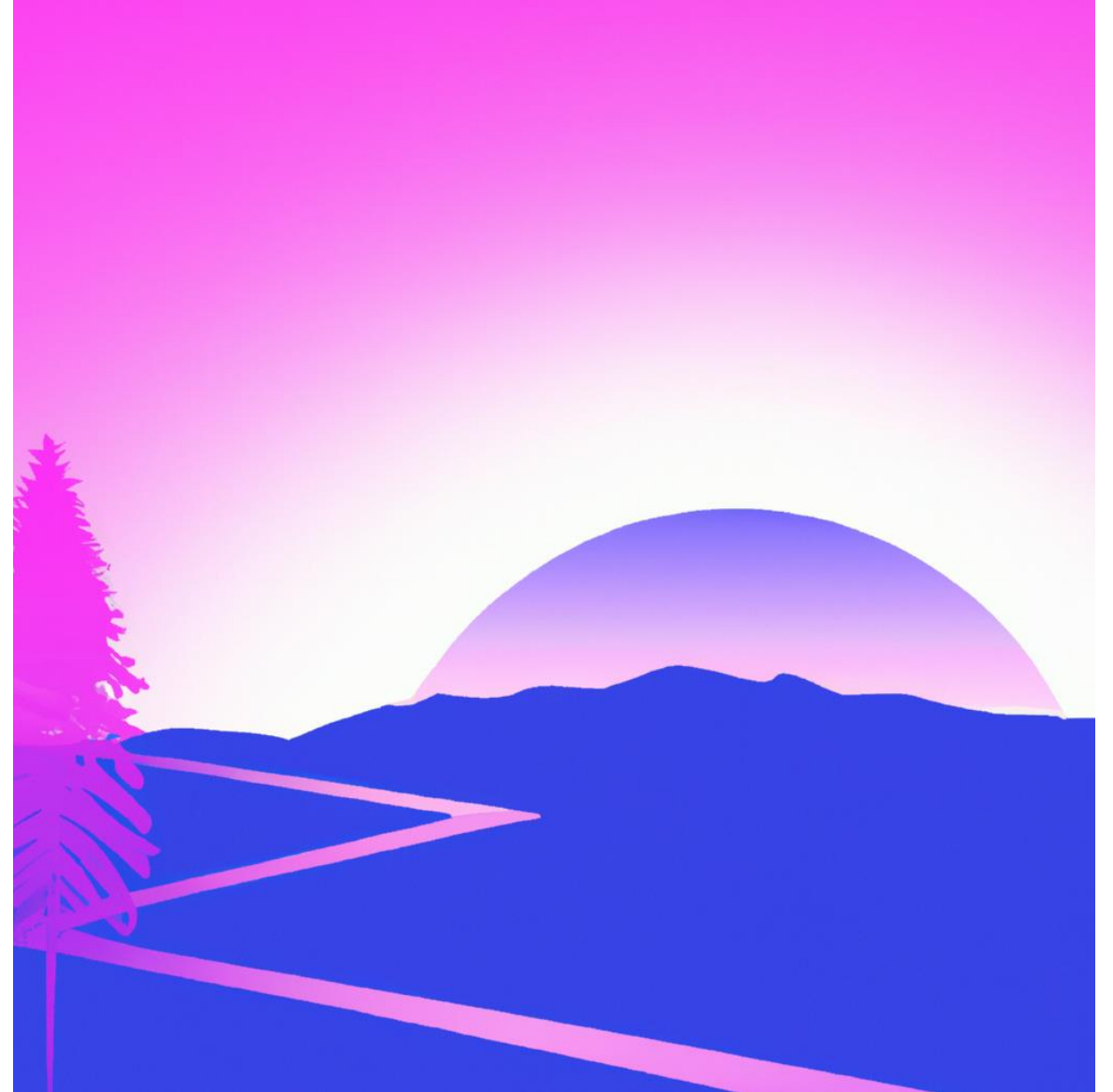


Spark Python

Kafka

Crédit Agricole de Normandie

Complément – Mai 2025



Avant-propos

Ce support a été créé à partir d'excellentes sources, dont le travail d'explication de [Régis Behmo](#), distribué sous licence Creative Commons (<https://creativecommons.org/licenses/by-sa/4.0/>).

En le remerciant pour ce travail qui a inspiré 2 slides, cette licence implique de distribuer le contenu de ces slides (à l'exception de la charte graphique) sous la même licence.

Vous pouvez donc réutiliser l'ensemble du contenu (le fond) dans vos codes, à des fins commerciales, pour donner également des formations, etc.

Introduction à Kafka

Kafka en deux mots

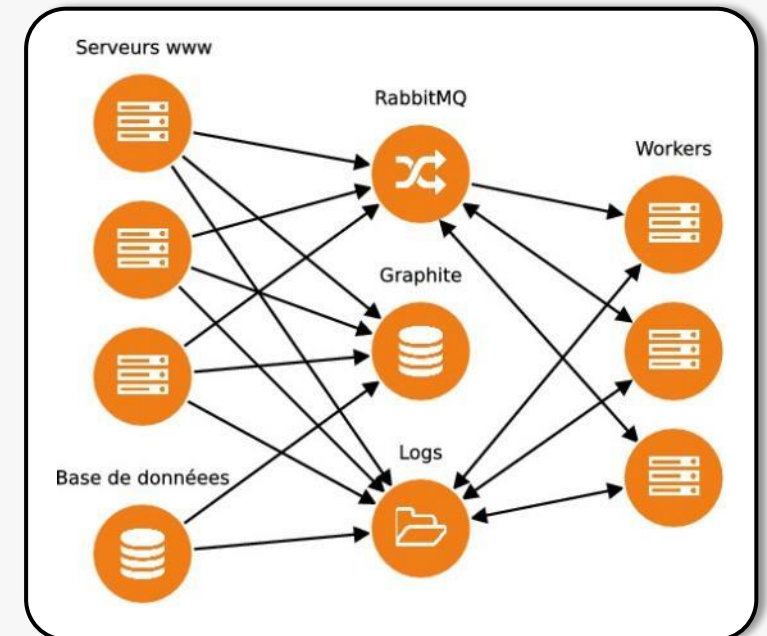
- Kafka : « plateforme de diffusion de données distribuée et évolutive, conçue pour gérer efficacement le flux de données en temps réel ».
- Utilisé pour la gestion des flux de données, la création de pipelines de données en continu, la mise en œuvre de systèmes de messagerie (et d'autres applications liées au traitement de données en temps réel).
- En réalité, Kafka est bien plus qu'une file de messages et peut être utilisé comme une plateforme complète d'échanges de données (Kafka peut agir comme une plateforme distribuée qui centralise tous les messages qui transitent entre différentes applications).

Introduction à Kafka

Intérêt de Kafka

- Pour comprendre l'utilité d'une telle plateforme, imaginons une plateforme web classique composée d'une application, d'une base de données ainsi que de quelques *workers* asynchrones.
- Le rôle des workers est d'exécuter les tâches asynchrones de la plateforme, comme l'e-mailing, et ils sont orchestrés par [RabbitMQ](#).
- Les logs des différents serveurs sont agrégés par [syslog-ng](#).
- Des *workers* supplémentaires analysent périodiquement les logs pour générer des statistiques.
- Enfin, les performances (utilisation CPU, RAM disponible, espace disque) des serveurs sont collectées par [statsd](#) et envoyés vers [Graphite](#).

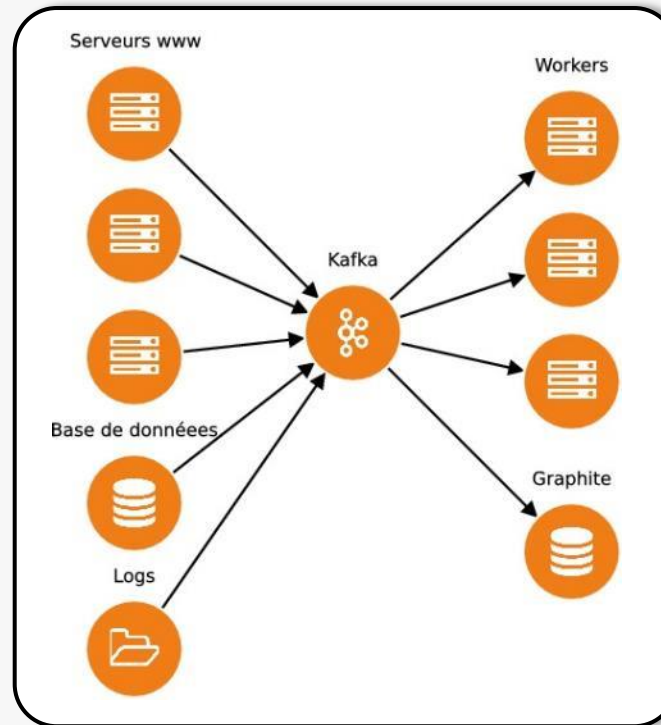
Plusieurs services sont chargés de recevoir des messages et de les transmettre à d'autres services.



Introduction à Kafka

Intérêt de Kafka

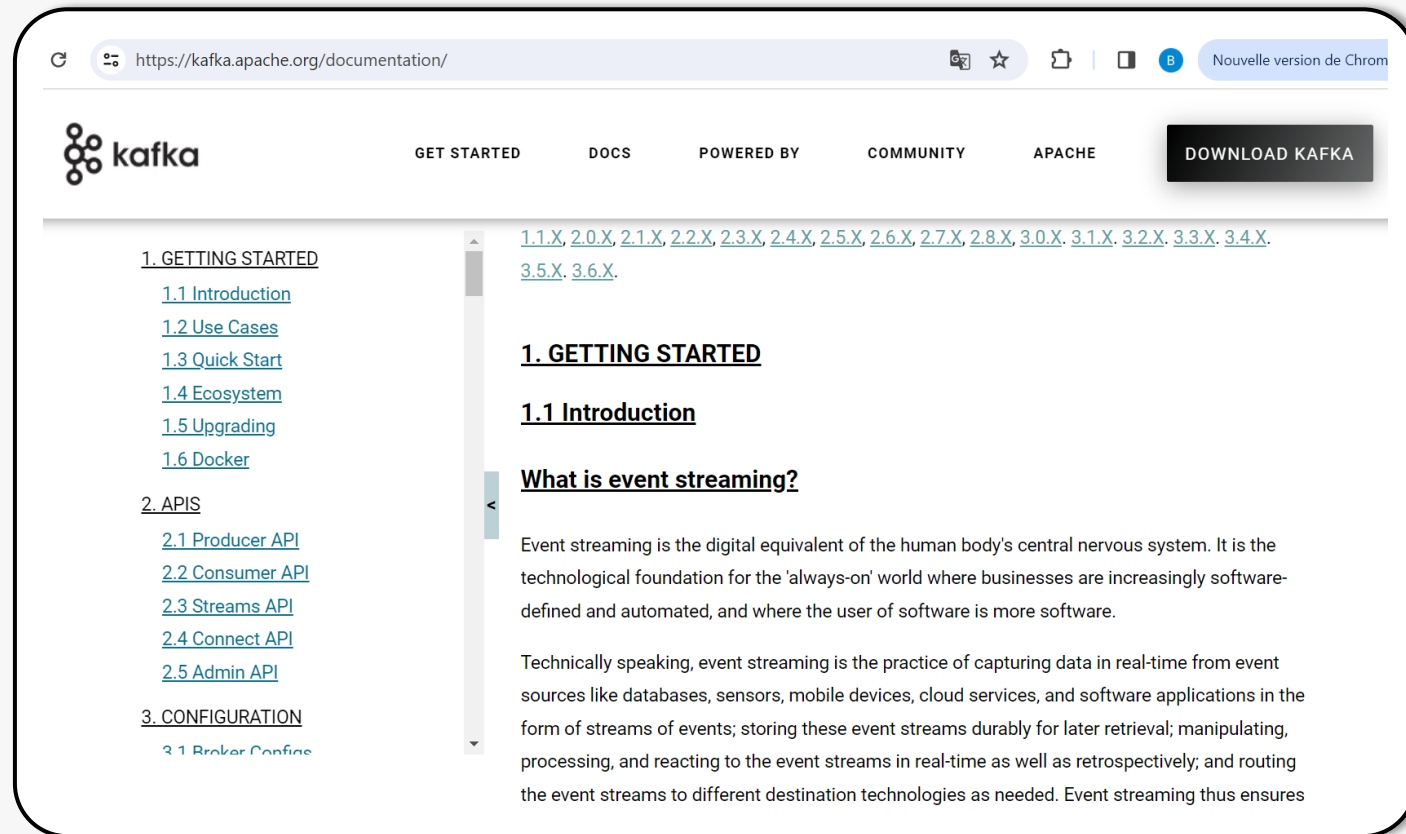
- ➡ Kafka permet de centraliser les messages et de les redistribuer à une variété de services, ce qui simplifie considérablement l'architecture de l'application.



Introduction à Kafka

Intérêt de Kafka

Vous en avez l'habitude, mais référez-vous (vraiment) à la documentation officielle de Kafka au besoin. Elle aussi est excellente.



<https://kafka.apache.org/documentation/>

Introduction à Kafka

Fonctionnalités clefs

- ➡ Publier (écrire) et souscrire (lire) à des flux d'événements, y compris l'importation/exportation continue de vos données depuis d'autres systèmes.
- ➡ Stocker des flux d'événements de manière durable et fiable aussi longtemps que vous le souhaitez.
- ➡ Traiter des flux d'événements au fur et à mesure de leur survenue ou de manière rétrospective.

De manière distribuée, hautement évolutive, élastique, tolérante aux pannes et sécurisée.

Kafka peut être déployé sur du matériel bare metal, des machines virtuelles et des conteneurs, aussi bien sur site que dans le cloud.

Vous pouvez choisir entre la gestion autonome de vos environnements Kafka et l'utilisation de services entièrement gérés proposés par différents fournisseurs.

Introduction à Kafka

Composants

- Kafka est un système distribué composé de serveurs et de clients qui communiquent via un protocole réseau TCP haute performance.
- **Serveurs**
 - Kafka est exécuté sous la forme d'un cluster composé d'un ou plusieurs serveurs pouvant s'étendre sur plusieurs centres de données ou régions cloud.
 - Certains de ces serveurs constituent la couche de stockage, appelée les brokers.
 - D'autres serveurs exécutent Kafka Connect pour importer et exporter continuellement des données sous forme de flux d'événements afin d'intégrer Kafka avec les systèmes existants tels que les bases de données relationnelles, d'autres clusters Kafka...
 - Un cluster Kafka est évolutif et tolérant aux pannes : si l'un de ses serveurs échoue, les autres serveurs prendront le relais pour assurer un fonctionnement continu sans perte de données.

Introduction à Kafka

Composants

- **Brokers** Serveurs responsables du stockage, de la gestion et de la diffusion des messages. Chaque broker est une instance de Kafka exécutée sur un nœud du cluster Kafka. Les brokers sont conçus pour être évolutifs (on peut ajouter ou supprimer des brokers selon les besoins pour ajuster la capacité du cluster).
- Pour garantir la tolérance aux pannes et la résilience du système, les données sur les brokers sont répliquées sur plusieurs brokers dans le cluster. Ainsi, en cas de défaillance d'un broker, les données peuvent être récupérées à partir des répliques disponibles.

Introduction à Kafka

Composants

Clients

Permettent d'écrire des applications distribuées et des microservices qui lisent, écrivent et traitent des flux d'événements en parallèle, à grande échelle (de manière tolérante aux pannes, même en cas de problèmes réseau ou de défaillances de machines, vous connaissez la chanson...).

Kafka est livré avec certains de ces clients inclus, auxquels s'ajoutent des dizaines de clients fournis par la communauté Kafka : des clients sont disponibles pour Java et Scala, y compris la bibliothèque de haut niveau Kafka Streams, pour Go, Python, C/C++, et de nombreux autres langages de programmation ainsi que des APIs REST.

Introduction à Kafka

Composants

Events

Événements

Enregistrent le fait que "quelque chose se soit produit" (dans le monde, dans votre entreprise...).

Également appelé enregistrement ou message dans l'écosystème Kafka.

Lorsqu'on lit ou écrit des données dans Kafka, on le fait sous forme d'événements.

Conceptuellement, un événement a une clé, une valeur, un horodatage et des en-têtes de métadonnées facultatifs.

Clé de l'événement : "Alice"

Valeur de l'événement : "A effectué un paiement de 200€ à Bob"

Horodatage de l'événement : "8 avril 2024 à 10h43"

Introduction à Kafka

Composants

Producers Applications clientes qui publient (écrivent) des événements dans Kafka.

Consumers Applications qui s'abonnent à (lisent et traitent) ces événements.

Les producteurs et les consommateurs sont entièrement découplés et indépendants les uns des autres (élément clef de conception pour l'évolutivité). Par exemple, les producteurs n'ont jamais besoin d'attendre les consommateurs.

Kafka fournit diverses garanties telles que la capacité à traiter les événements en sémantique exactly-once (nous y reviendrons).

Introduction à Kafka

Composants

Les événements sont organisés et stockés de manière durable dans des **topics**.

De manière simplifiée, un topic est similaire à un dossier dans un système de fichiers, et les événements sont les fichiers dans ce dossier.

Exemple de topic : "paiements".

Les topics dans Kafka sont toujours multi-producteurs et multi-abonnés : un topic peut avoir zéro, un ou plusieurs producteurs qui écrivent des événements dessus, ainsi que zéro, un ou plusieurs consommateurs qui s'abonnent à ces événements.

Les événements dans un topic peuvent être lus aussi souvent que nécessaire - contrairement à de nombreux systèmes de messagerie traditionnels, les événements ne sont pas supprimés après leur consommation. Au lieu de cela, on va définir pendant combien de temps Kafka doit conserver les événements grâce à un paramètre de configuration par topic (après quoi les anciens événements seront supprimés).

Les performances de Kafka sont peu sensibles à la taille des données (stocker des données pendant une longue période est ok).

Introduction à Kafka

Composants

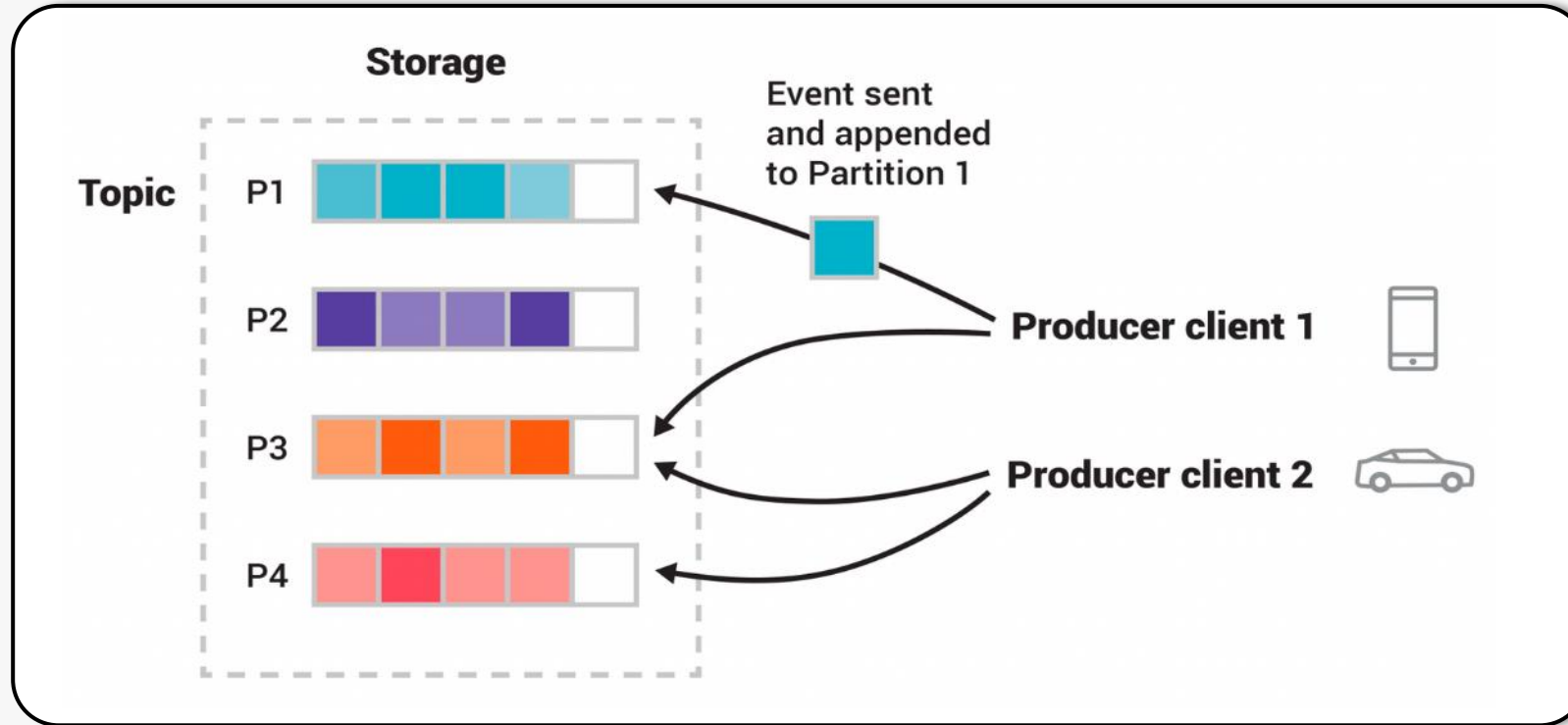
Les topics sont partitionnés (chaque topic est réparti sur un certain nombre de "buckets" situés sur différents brokers Kafka).

Cette répartition distribuée est très importante pour la scalabilité car elle permet aux applications clientes de lire et d'écrire les données à partir/de nombreux brokers en même temps.

Lorsqu'un nouvel événement est publié dans un topic, il est en fait ajouté à l'une des partitions du topic. Les événements avec la même clé d'événement (par exemple, un identifiant de client ou de véhicule) sont écrits dans la même partition, et Kafka garantit que tout consommateur d'une partition donnée d'un topic lira toujours les événements de cette partition dans exactement le même ordre dans lequel ils ont été écrits.

Introduction à Kafka

Composants



Cet exemple de topic comporte quatre partitions P1 à P4.

Deux producteurs différents publient, indépendamment l'un de l'autre, de nouveaux événements dans le topic en écrivant des événements sur le réseau vers les partitions du topic.

Les événements avec la même clé (indiquée par leur couleur dans la figure) sont écrits dans la même partition.

Les deux producteurs peuvent écrire dans la même partition si nécessaire.

Introduction à Kafka

Composants

Réplication

Pour rendre les données tolérantes aux pannes et hautement disponibles, chaque topic peut être répliqué (parfois à travers plusieurs régions géographiques ou centres de données), de sorte qu'il y ait toujours plusieurs brokers qui possèdent une copie des données au cas où des problèmes surviendraient, en cas de maintenance sur les brokers...

Un paramètre de configuration courant en production est un facteur de réplication de 3, c'est-à-dire qu'il y aura toujours trois copies de vos données. Cette réplication est effectuée au niveau des partitions du topic.

Introduction à Kafka

Composants

Offsets

Identifiants associés à chaque message dans un topic-partition. Ils indiquent la position d'un consommateur dans le flux de données.

Chaque message dans un topic-partition a un offset unique qui le distingue des autres messages.

Lorsqu'un consommateur lit un message, son offset est avancé pour indiquer qu'il a traité ce message. Le consommateur peut ensuite utiliser cet offset pour reprendre sa lecture là où il s'était arrêté en cas de reprise après une panne ou un redémarrage.

Les offsets sont stockés dans un sujet interne nommé "__consumer_offsets" et sont gérés par les brokers Kafka. Cela garantit que les offsets sont persistants et disponibles même en cas de panne d'un consommateur ou d'un broker.

Les consommateurs peuvent contrôler manuellement leur position dans le flux de données en manipulant les offsets. Par exemple, ils peuvent réinitialiser l'offset pour relire les messages déjà lus ou avancer l'offset pour ignorer les messages précédents.

Introduction à Kafka

Composants

Consumers groups

Regroupements logiques de consommateurs qui se partagent la lecture des partitions d'un ou plusieurs topics. Les consommateurs au sein d'un même groupe coopèrent pour consommer les messages de manière efficace et parallèle.

Lorsqu'un groupe de consommateurs est créé, Kafka attribue automatiquement à chaque membre du groupe un ensemble de partitions à lire. Chaque partition d'un topic est consommée par un seul membre du groupe, ce qui permet une distribution équilibrée de la charge de travail.

Les consommateurs au sein d'un même groupe peuvent lire les messages de manière parallèle. Cela permet d'augmenter le débit de traitement des messages en répartissant la charge sur plusieurs consommateurs.

Au sein d'un groupe de consommateurs, les offsets sont partagés entre les membres du groupe. Cela signifie que chaque consommateur dans le groupe connaît la position de lecture des autres membres et peut coordonner la répartition des partitions en fonction de l'avancement de la lecture.

Introduction à Kafka

Composants

ZooKeeper

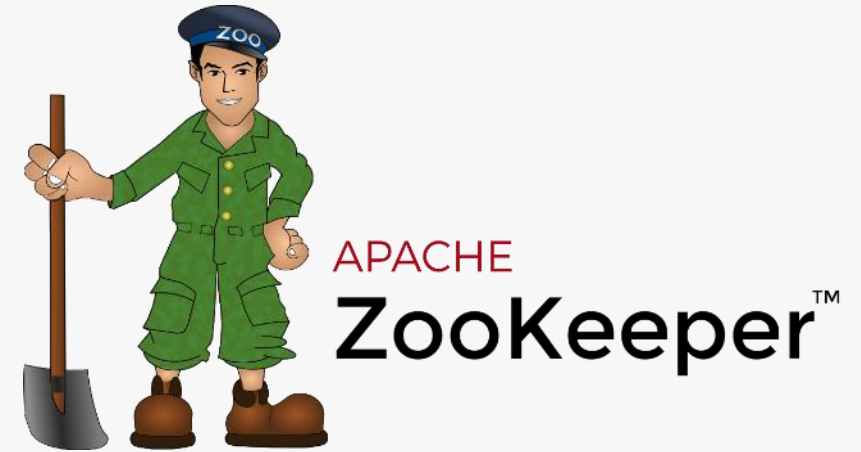
Service de coordination distribué utilisé dans les systèmes distribués pour la gestion des configurations, la synchronisation, la gestion des verrous (locks) et d'autres tâches de coordination.

Fournit un ensemble de primitives de haut niveau qui permettent aux applications de coordonner et de gérer efficacement des tâches distribuées dans un environnement distribué.

➡ ZooKeeper agit comme un service de coordination centralisé pour les applications distribuées. Il fournit des primitives de base telles que les nœuds de données, les verrous distribués, les files d'attente, les séquences, etc., qui permettent aux applications de coordonner leurs actions dans un environnement distribué.

➡ ZooKeeper organise les données sous forme d'une hiérarchie de nœuds semblable à un système de fichiers. Chaque nœud peut contenir des données et être associé à des opérations telles que la lecture, l'écriture, la création et la suppression.

➡ ZooKeeper maintient un état partagé entre tous les nœuds du cluster, ce qui garantit la cohérence et la fiabilité des données stockées.



Introduction à Kafka

Composants

ZooKeeper

- ➡ ZooKeeper utilise un mécanisme d'élection de leader pour élire un leader parmi les nœuds d'un ensemble. Le leader est chargé de coordonner les opérations de lecture et d'écriture, tandis que les autres nœuds fonctionnent en mode de suiveur.
- ➡ ZooKeeper est conçu pour être hautement disponible et tolérant aux pannes. Il utilise une architecture distribuée et répliquée pour garantir la disponibilité des services même en cas de défaillance d'un ou plusieurs nœuds.
- ➡ Dans le contexte de Kafka, ZooKeeper est utilisé pour le stockage des métadonnées du cluster, la gestion des partitions et la coordination des brokers.

Introduction à Kafka

Composants

Log compaction

Fonctionnalité d'Apache Kafka qui garantit que le journal conserve au moins la dernière valeur connue pour chaque clé de message dans une période de rétention spécifiée (Kafka conservera à la fois la valeur la plus récente pour chaque clé de message et les valeurs plus anciennes nécessaires pour reconstruire l'état des données).

➡ Chaque message dans Kafka peut avoir une paire clé-valeur associée. La **clé** est utilisée pour identifier les messages et permet la compaction des journaux.

➡ Le journal Kafka est segmenté en **segments**, et chaque segment contient un ensemble séquentiel de messages. La compaction des journaux fonctionne sur une base de segment.

➡ Kafka vous permet de configurer différentes **politiques de suppression** pour les segments de journal, telles que la rétention basée sur le temps ou la rétention basée sur la taille. Avec la compaction des journaux, une politique de suppression supplémentaire appelée "compaction" est introduite.

➡ Lorsque la compaction des journaux est activée pour un topic, Kafka surveille continuellement les segments de journal. Il identifie les messages avec des clés en double au sein d'un segment et ne conserve que le message le plus récent pour chaque clé tout en supprimant les doublons plus anciens.

Introduction à Kafka

Composants

Log compaction

La compaction des logs (journaux) garantit que les consommateurs peuvent toujours récupérer la valeur la plus récente pour une clé de message particulière, même si plusieurs mises à jour ont eu lieu. Cela est particulièrement utile pour maintenir l'état actuel d'un ensemble de données ou pour maintenir des vues matérialisées.

Kafka conserve toujours les segments de journal en fonction de la politique de rétention configurée. Pour les topics de journal compactés, Kafka garantit qu'au moins la valeur la plus récente pour chaque clé de message est conservée dans la période de rétention spécifiée.

Introduction à Kafka

APIs

Kafka comprend cinq API principales :

1. ➡ L'API Producer permet aux applications d'envoyer des flux de données vers des topics dans le cluster Kafka.
2. ➡ L'API Consumer permet aux applications de lire des flux de données à partir de topics dans le cluster Kafka.
3. ➡ L'API Streams permet de transformer des flux de données des topics d'entrée vers des topics de sortie.
4. ➡ L'API Connect permet de mettre en œuvre des connecteurs qui extraient en continu des données à partir d'un système source ou d'une application vers Kafka, ou qui poussent des données depuis Kafka vers un système cible ou une application.
5. ➡ L'API Admin permet de gérer et d'inspecter les topics, les brokers et autres objets Kafka.

Kafka expose toutes ses fonctionnalités via un protocole indépendant du langage qui possède des clients disponibles dans de nombreux langages de programmation. Cependant, seuls les clients Java sont maintenus en tant que partie intégrante du projet principal Kafka, les autres étant disponibles en tant que projets open source indépendants. Une liste des clients non-Java est disponible [ici](#).

Introduction à Kafka

Utilisation

Messagerie

Kafka peut fonctionner en tant que remplacement d'un broker de messages plus traditionnel (pour découpler le traitement des producteurs de données, pour mettre en mémoire tampon les messages non traités...).

En comparaison avec la plupart des systèmes de messagerie, Kafka offre une meilleure capacité de traitement, ainsi que les aspects que nous avons décrits : partition intégrée, réplication, une tolérance aux pannes, ce qui en fait une bonne solution pour les applications de traitement de messages à grande échelle.

Dans ce domaine, Kafka est comparable à des systèmes de messagerie traditionnels tels que ActiveMQ ou RabbitMQ.

Introduction à Kafka

Utilisation

Suivi de l'activité sur le site web

Mais le cas d'utilisation initial de Kafka était de pouvoir reconstruire un pipeline de suivi de l'activité utilisateur sous forme de flux de publication-abonnement en temps réel.

L'activité du site (pages vues, recherches ou autres actions que les utilisateurs peuvent effectuer) est alors publiée dans des topics centraux avec un topic par type d'activité.

Ces flux sont disponibles pour abonnement pour une gamme de cas d'utilisation, y compris le traitement en temps réel, la surveillance en temps réel et le chargement dans Hadoop ou les systèmes de data warehousing hors ligne pour un traitement et un reporting hors ligne.

Le suivi de l'activité est souvent très volumineux car de nombreux messages d'activité sont générés pour chaque page vue d'utilisateur.

Introduction à Kafka

Utilisation

Métriques

Kafka est souvent utilisé pour les données de surveillance opérationnelle. Cela implique d'agréger des statistiques à partir d'applications distribuées pour produire des flux centralisés de données opérationnelles.

Introduction à Kafka

Utilisation

Agrégation de logs

Kafka est également très utilisé comme solution de remplacement pour l'agrégation de logs.

L'agrégation de logs collecte généralement les fichiers logs physiques des serveurs et les place dans un endroit centralisé (un serveur de fichiers ou HDFS par exemple) pour traitement.

Kafka abstrait les détails des fichiers et offre une abstraction plus propre des données de journal ou d'événements sous forme de flux de messages.

Cela permet un traitement à latence plus faible et un support plus facile pour de multiples sources de données et une consommation de données distribuée.

En comparaison avec des systèmes centrés sur les journaux comme Scribe ou Flume, Kafka offre des performances similaires, des garanties de durabilité un peu plus solides en raison de la réplication, et une latence de bout en bout assez significativement inférieure.

Introduction à Kafka

Utilisation

Event sourcing

Style de conception d'application où les changements d'état sont enregistrés sous forme d'une séquence d'enregistrements (records) ordonnés dans le temps. Le support de Kafka pour les journaux de données stockés de manière très volumineuse en fait un excellent backend pour une application construite dans ce style.

Introduction à Kafka

Utilisation

Commit log

Kafka peut servir de type de journal de validation externe pour un système distribué.

Le journal aide à répliquer les données entre les nœuds et agit comme un mécanisme de resynchronisation pour les nœuds défectueux afin de restaurer leurs données.

La fonctionnalité de compaction des journaux dans Kafka aide à prendre en charge cette utilisation.

Dans cette utilisation, Kafka est similaire au projet Apache BookKeeper.

Introduction à Kafka

Utilisation

Traitement de flux

De nombreux utilisateurs de Kafka traitent des données dans des pipelines de traitement composés de plusieurs étapes, où les données d'entrée brutes sont consommées à partir de topics Kafka puis agrégées, enrichies ou autrement transformées en nouveaux topics pour une consommation ultérieure ou un traitement supplémentaire.

C'est probablement un cas d'utilisation qui nous intéresse beaucoup puisque nous sommes intéressés par Spark !

Par exemple, un pipeline de traitement pour recommander des articles de presse pourrait extraire le contenu des articles à partir de flux RSS et le publier dans un topic "articles", un traitement ultérieur pourrait normaliser ou dédupliquer ce contenu et publier le contenu d'article nettoyé dans un nouveau topic, une étape de traitement finale pourrait tenter de recommander ce contenu aux utilisateurs. De tels pipelines de traitement créent des graphiques de flux de données en temps réel basés sur les topics individuels.

À partir de la version 0.10.0.0, une bibliothèque de traitement de flux légère mais puissante appelée Kafka Streams est disponible dans Apache Kafka pour effectuer ce type de traitement des données comme décrit ci-dessus.

D'autres outils de traitement de flux open source alternatifs : Apache Storm, Apache Samza.

Introduction à Kafka

Conclusion : Kafka et Spark Structured Streaming

- Kafka et Structured Streaming de Spark sont à l'origine **deux technologies distinctes avec des fonctionnalités différentes**, qui peuvent être utilisées ensemble dans un pipeline de traitement de données.
- Kafka est un système de messagerie distribué qui permet de stocker, de publier et de consommer des flux de messages de manière distribuée et hautement évolutive. Il est souvent utilisé comme source de données dans les pipelines de traitement de données en streaming.
- Structured Streaming de Spark est un moteur de traitement de données en streaming intégré à Apache Spark, qui fournit une API haut niveau pour le traitement des données en continu avec des concepts de programmation de lot structuré. Il permet de traiter des flux de données structurés à partir de diverses sources telles que Kafka, des files d'attente, des bases de données, etc., et d'exécuter des opérations de traitement sur ces flux de manière similaire au traitement de données par lots.
- Bien que Kafka et Structured Streaming de Spark puissent être utilisés ensemble, ils ne remplacent pas directement les fonctionnalités de l'autre. Kafka fournit la capacité de stocker et de diffuser des flux de messages, tandis que Structured Streaming de Spark fournit un moteur de traitement de données en streaming avec des fonctionnalités de transformation et d'analyse avancées.

Introduction à Kafka

Conclusion : Kafka et Spark Structured Streaming

Kafka Streams et Structured Streaming de Spark sont par contre deux technologies qui fournissent des fonctionnalités similaires (mais avec des approches différentes) pour le traitement de données en streaming : SPE.


Introduction à Kafka




Ouverture

LinkedIn Engineering

Home Blog Data Open Source Trust Infrastructure

The Log: What every software engineer should know about real-time data's unifying abstraction

 Jay Kreps December 16, 2013

 Share  Post  Share

I joined LinkedIn about six years ago at a particularly interesting time. We were just beginning to run up against the limits of our monolithic, centralized database and needed to start the transition to a portfolio of specialized distributed systems. This has been an interesting experience: we built, deployed, and run to this day a distributed graph database, a distributed search backend, a Hadoop installation, and a first and second generation key-value store.

One of the most useful things I learned in all this was that many of the things we were building had a very simple concept at their heart: the log. Sometimes called write-

<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>

Introduction à Kafka

Installation

Récupérer l'archive Kafka (si vous consultez ces slides après la formation, pensez à vérifier s'il n'existe pas un mirror link vers une version plus récente) :

```
$ curl https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz -o  
    <dossier_de_votre_choix>/kafka.tgz
```

Créer un dossier dans lequel extraire cette archive :

```
$ mkdir kafka  
$ cd kafka
```

Y extraire l'archive :

```
$ tar -xzf ../<dossier_de_votre_choix>/kafka.tgz
```

Introduction à Kafka

Prise en main

Nous allons lancer Kafka avec ZooKeeper (KRaft est une alternative).

Rentrer dans le dossier créé et lancer le service ZooKeeper :

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

Ouvrir un nouveau terminal, se rendre dans le même dossier et lancer le broker Kafka :

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

Introduction à Kafka

Prise en main

Avant d'écrire vos premiers événements, vous devez créer un topic.

Ouvrir un nouveau terminal et exécuter la commande suivante :

```
$ bin/kafka-topics.sh --create --topic vive-le-ca --bootstrap-server localhost:9092
```

Tous les outils en ligne de commande de Kafka disposent d'options supplémentaires : vous pouvez exécuter la commande `kafka-topics.sh` sans aucun argument pour afficher des informations sur l'utilisation. Par exemple, pour afficher le nombre de partitions du nouveau topic :

```
$ bin/kafka-topics.sh --describe --topic vive-le-ca --bootstrap-server localhost:9092
```

Introduction à Kafka

Prise en main

Un client Kafka communique avec les brokers Kafka via le réseau pour écrire (ou lire) des événements. Une fois reçus, les brokers stockent les événements de manière durable et tolérante aux pannes aussi longtemps que vous en avez besoin (voire de manière indéfinie).

Exécuter le client console producer pour écrire quelques événements dans votre topic. Par défaut, chaque ligne que vous saisissez entraînera l'écriture d'un événement distinct dans le sujet.

```
$ bin/kafka-console-producer.sh --topic vive-le-ca --bootstrap-server localhost:9092
```

(Ctrl-C permet d'arrêter le producer, mais ne le faites pas tout de suite.)

Introduction à Kafka

Prise en main

Ouvrez une autre session de terminal et exécutez le client consommateur de console pour lire les événements que vous venez de créer :

```
$ bin/kafka-console-consumer.sh --topic vive-le-ca --from-beginning --bootstrap-server localhost:9092
```

Vous pouvez arrêter le client consommateur avec Ctrl-C à tout moment, mais ne le faites pas tout de suite.

Revenir au terminal du producer et écrire un nouveau message.

Visualiser comment cet événement apparaît dans le terminal du consumer.

Comme les événements sont stockés de manière durable dans Kafka, ils peuvent être lus autant de fois et par autant de consommateurs que vous le souhaitez.

Ouvrir encore un nouveau terminal et réexécuter la commande précédente. Vous pouvez jouer avec différents timings et la présence ou non de l'option `--from-beginning`.

Ensuite vous pouvez arrêter le consumer et le producer (conserver Zookeeper et le broker, ou relancer ces deux serveurs).

Introduction à Kafka

Prise en main

Kafka Connect permet d'ingérer continuellement des données depuis des systèmes externes vers Kafka, et vice versa. C'est un outil extensible qui exécute des connecteurs, lesquels mettent en œuvre la logique personnalisée pour interagir avec un système externe.

Le fichier `connect-file-3.7.0.jar` est une bibliothèque JAR qui contient un connecteur prédéfini pour Kafka Connect. Ce connecteur spécifique est utilisé pour lire des données à partir de fichiers locaux et les écrire dans un topic Kafka, ainsi que pour lire des données à partir d'un topic Kafka et les écrire dans des fichiers locaux.

L'ajout de ce fichier JAR au chemin du plugin dans la configuration de Kafka Connect permet à Kafka Connect de trouver et de charger ce connecteur lors de son exécution. Une fois le connecteur chargé, vous pouvez le configurer pour qu'il lise ou écrive des données selon vos besoins spécifiques.

Introduction à Kafka

Prise en main

Ouvrir le fichier config/connect-standalone.properties et ajouter la ligne suivante :

```
plugin.path=<chemin_vers_dossier_kafka>/kafka_2.13-3.9.0/libs/connect-file-3.9.0.jar
```

(Adapter le chemin en fonction de l'OS utilisé.)

Enregistrer le fichier.

Créer un repertoire input quelque part dans votre espace de travail.

Ouvrir le fichier config/connect-file-source.properties et modifier la ligne *file* de la manière suivante :

```
file=<chemin_vers_input>/test.txt
```

Enfin, ouvrir le fichier config/connect-file-sink.properties et modifier la ligne *file* afin de spécifier là où vous souhaitez obtenir la sortie (sink) :

```
file=<chemin_vers_sink>/test.sink.txt
```

Remarque : c'est une bonne pratique de travailler en chemins absolus plutôt que relatifs.

Introduction à Kafka

Prise en main

Démarrer deux connecteurs en mode autonome (ils s'exécuteront dans un seul processus local dédié), en fournissant 3 fichiers de configuration en tant que paramètres, de la manière suivante :

```
$ bin/connect-standalone.sh config/connect-standalone.properties  
config/connect-file-source.properties config/connect-file-sink.properties
```

Quelques informations sur ces 3 fichiers :

- Le premier est toujours la configuration du processus Kafka Connect, contenant des configurations communes telles que les courtiers Kafka auxquels se connecter et le format de sérialisation des données.
- Les 2 autres fichiers de configuration restants spécifient chacun un connecteur à créer. Ces fichiers comprennent un nom de connecteur unique, la classe de connecteur à instancier, et toute autre configuration requise par le connecteur.

Introduction à Kafka

Prise en main

Ces fichiers de configuration d'exemple, inclus avec Kafka, utilisent la configuration de cluster local par défaut qui a été démarrée précédemment et créent deux connecteurs : le premier est un connecteur source qui lit des lignes à partir d'un fichier d'entrée et les produit dans un topic Kafka, et le deuxième est un connecteur puits qui lit les messages d'un topic Kafka et les écrit chacun dans un fichier de sortie.

Pendant le démarrage, vous verrez plusieurs messages de journalisation, y compris certains indiquant que les connecteurs sont en cours d'instanciation. Une fois que le processus Kafka Connect a démarré, le connecteur source va commencer à essayer de lire les lignes à partir de test.txt et à les produire dans le topic connect-test, et le connecteur puits va commencer à lire les messages à partir du topic connect-test et à les écrire dans le fichier test.sink.txt.

A l'heure actuelle nous n'avons pas créé ce fichier texte. Constater le comportement de Kafka.

Dans un nouveau terminal, aller dans le dossier spécifié et créer un fichier test.txt dans ce dossier, par exemple :

```
echo -e "line1\nline2" > test.txt
```

Vérifier les logs dans le terminal précédent.

Introduction à Kafka

Prise en main

On peut vérifier que les données ont été transmises dans l'ensemble du pipeline en examinant le contenu du fichier de sortie :

```
$ more <chemin_vers_sink>/test.sink.txt
```

A noter que les données sont stockées dans le topic Kafka connect-test, donc nous pouvons également exécuter un consommateur de console pour voir les données dans le topic (ou utiliser du code de consommateur personnalisé pour les traiter) :

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning
```

Les connecteurs continuent de traiter les données. Ajouter des données au fichier et visualiser qu'elles traversent le pipeline :

```
$ echo Another line >> test.txt
```

Vous devriez voir la ligne apparaître dans la sortie du consommateur de console et dans le fichier de destination.

Introduction à Kafka

Kafka Streams

A l'adresse suivante se situe une implémentation classique du wordCount en java, adaptée pour Kafka Streams :

<https://github.com/apache/kafka/blob/trunk/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java>

Si besoin, relancer les commandes suivantes dans des terminaux séparés :

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

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

Introduction à Kafka

Kafka Streams

Créer le topic d'entrée streams-plaintext-input :

```
$ bin/kafka-topics.sh --create \  
  --bootstrap-server localhost:9092 \  
  --replication-factor 1 \  
  --partitions 1 \  
  --topic streams-plaintext-input
```

Créer le topic de sortie streams-wordcount-output avec compaction activée (nous reviendrons sur la raison) :

```
$ bin/kafka-topics.sh --create \  
  --bootstrap-server localhost:9092 \  
  --replication-factor 1 \  
  --partitions 1 \  
  --topic streams-wordcount-output \  
  --config cleanup.policy=compact
```

Introduction à Kafka

Kafka Streams

Démarrer l'application wordCount :

```
$ bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

L'application lira à partir du topic d'entrée streams-plaintext-input, effectuera les calculs de l'algorithme WordCount sur chacun des messages lus, et écrira continuellement ses résultats actuels dans le topic de sortie streams-wordcount-output. Par conséquent, il n'y aura aucune sortie STDOUT sauf les entrées de journal au fur et à mesure que les résultats sont écrits dans Kafka.

Démarrer la console du producer dans un terminal séparé pour écrire des données d'entrée dans ce topic :

```
$ bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic streams-plaintext-input
```

Introduction à Kafka

Kafka Streams

Inspecter la sortie de l'application de démonstration WordCount en lisant à partir de son topic de sortie avec la console du consumer dans un terminal séparé :

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \  
  --topic streams-wordcount-output \  
  --from-beginning \  
  --formatter kafka.tools.DefaultMessageFormatter \  
  --property print.key=true \  
  --property print.value=true \  
  --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \  
  --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

Dans l'output, la première colonne est la clé du message Kafka au format `java.lang.String` et représente un mot qui est compté, et la deuxième colonne est la valeur du message au format `java.lang.Long`, représentant le dernier décompte du mot.

Continuer à écrire avec la console du producer dans le topic d'entrée `streams-plaintext-input`.

Introduction à Kafka

Kafka Streams

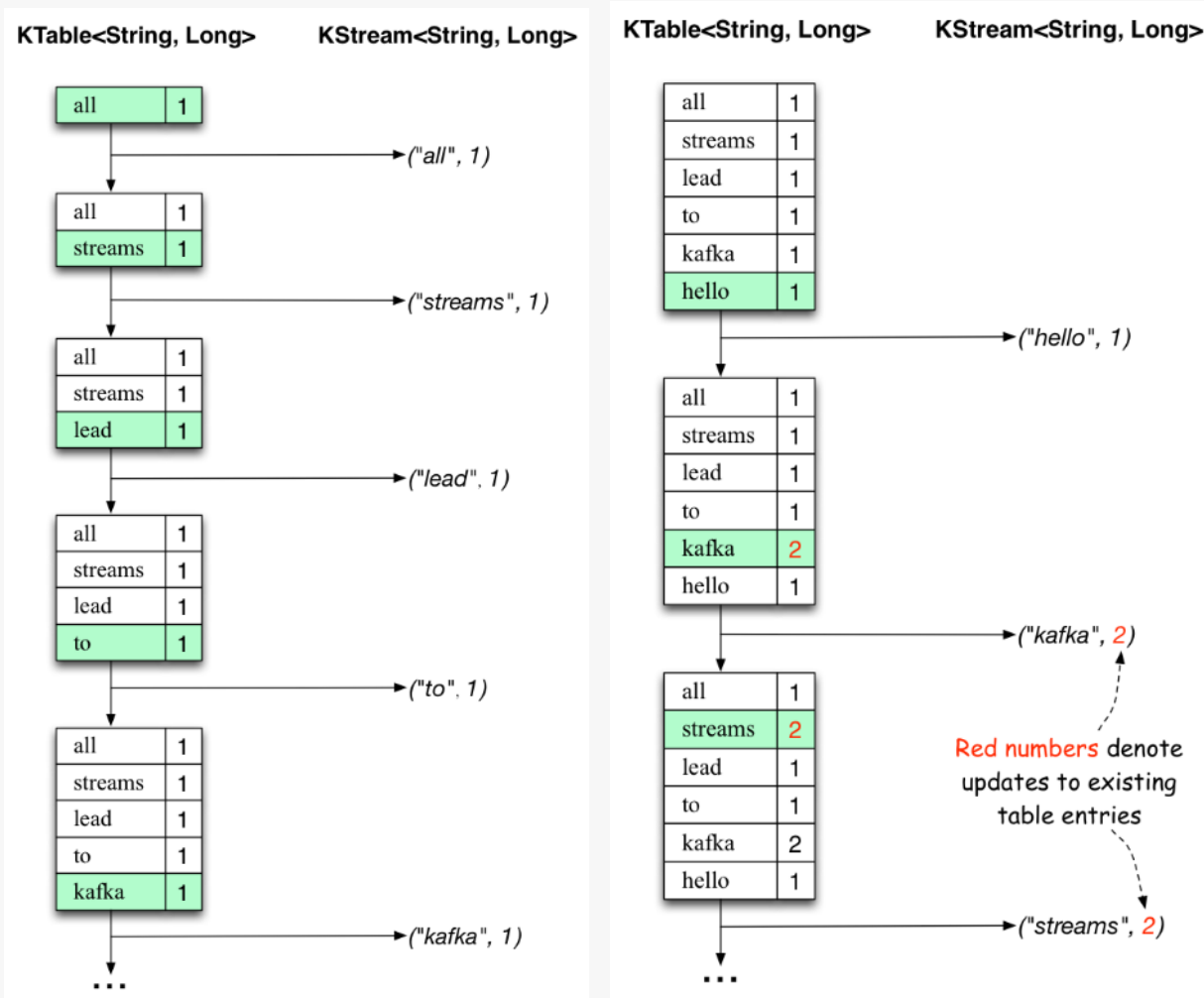
Les sorties de l'application Wordcount sont en fait un flux continu de mises à jour, où chaque enregistrement de sortie (c'est-à-dire chaque ligne dans la sortie originale ci-dessus) est un décompte mis à jour d'un seul mot, également appelé clé d'enregistrement (comme expliqué dans la présentation de Kafka).

Pour plusieurs enregistrements avec la même clé, chaque enregistrement ultérieur est une mise à jour du précédent.

Arrêter le consumer, le producer, l'application Kafka, le broker et le serveur ZooKeeper (CTRL-C).

Introduction à Kafka

Kafka Streams



La première colonne montre l'évolution de l'état actuel de la KTable<String, Long> qui compte les occurrences de mots pour le compteur.

La deuxième colonne montre les enregistrements de changement qui résultent des mises à jour d'état de la KTable et qui sont envoyés au topic Kafka de sortie streams-wordcount-output.

Fin du complément

Merci !