Cube Kylin alimenté par une source en streaming avec Kafka

Karim BALDÉ & Walid BELAHDJI

14/01/2018

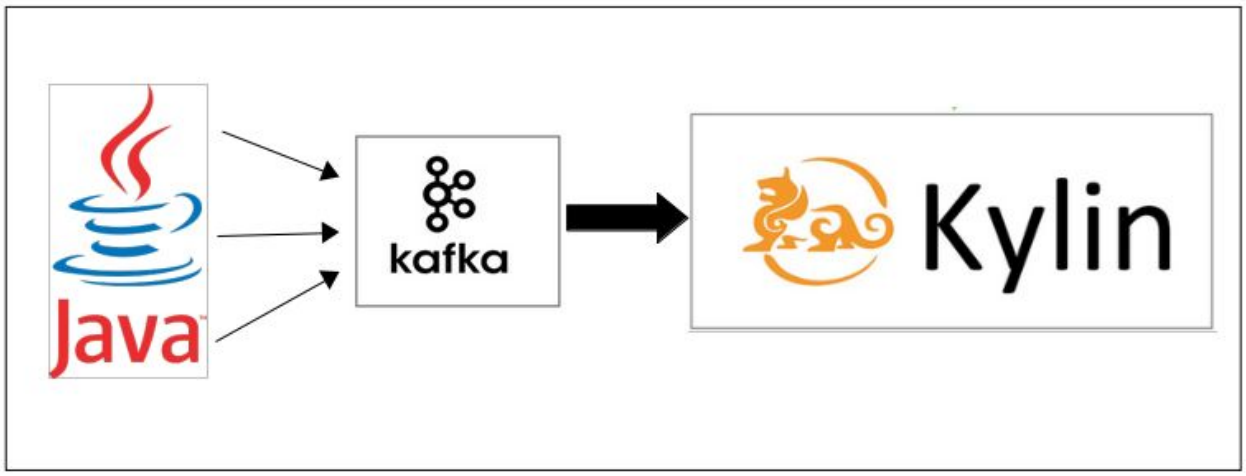
Introduction

La collecte des données est une partie essentielle des applications actuelles, bien plus encore dans le Big Data où le volume des données en est une caractéristique. Dans les architectures Big Data, il est aussi nécessaire d'avoir des mécanismes de traitements de données relativement rapides malgré le volume important. À titre d'exemple, les voitures modernes sont aujourd'hui équipées de 60 à 100 capteurs. Lorsqu'on gère un nombre important de véhicules, les données générées par tous ces capteurs peut atteindre plusieurs téraoctets chaque seconde. Dans de nombreux cas, le mode **batch** n'est pas suffisant au vu des exigences de certaines applications. Si le **micro-batch** (analyse et traitement des données avec un temps faible) s'avère efficace très souvent, le **temps réel** l'est encore plus et c'est ce qui va nous intéresser dans le cas présent.

Le temps réel consiste à traiter les données à la même vitesse qu'elles nous parviennent en minimisant la latence. Le passage à l'échelle se fait en mode horizontal sans perturber le système, les résultats des calculs restant disponible au fur et à mesure de leur traitement contrairement aux deux autres modes.

Pour gérer un flux de données temps réel, il faut deux composants : une file de message et un système de traitement de données temps réels. Apache Kafka est une implémentation d'application de file de message. Le système de traitement de données est souvent Spark, Storm ou Flink. Dans ce projet nous ne nous intéresserons pas aux outils de traitement mais plutôt à la gestion des files de messages ainsi que la transmission des données.

Nous allons concevoir un système permettant de créer des messages à partir d'une application Java que nous connecterons ensuite à **Apache Kafka** pour construire une file de messages qu'il transmettra à son tour à **Apache Kylin** afin d'exécuter des opérations propres à Kylin. L'architecture simplifiée de notre système est décrite dans le schéma ci-dessous.



Dans la suite de ce document, nous allons expliquer les différentes phases de la mise en place de cette architecture. Nous allons d'abord commencer par une présentation d'Apache Kafka en expliquant de façon brève ses composants. Nous détaillerons les configurations nécessaires à la mise en place d'un cluster Kafka et son déploiement dans des machines EC2. Dans un second temps, nous expliquerons le fonctionnement de notre application Java et les connexions à Kafka puis nous terminerons par le déploiement de Kylin sur un cluster EMR et des scénarios d'exécution de l'application de bout en bout.

Configuration de Kafka

Installation et exécution

Pour installer Apache Kafka, il faut simplement télécharger l'archive disponible sur le site officiel (https://kafka.apache.org/downloads) avec la commande wget par exemple sur le terminal et décompresser l'archive. Les binaires de kafka sont exécutés dans une **JVM**, il faut donc installer un jdk avant de pouvoir exécuter le client kafka.

Une fois l'installation terminée, on peut lancer Kafka après avoir lancé zookeeper qui est est un logiciel permettant de gérer toute la configuration dans un système distribué.

C’est donc la couche supérieure dans le cluster Kafka que nous mettons en place. Pour la suite du projet, il sera utile d’avoir une variable d’environnement qui pointe vers le répertoire de Kafka. On peut créer cette variable en exécutant sur le terminal la commande suivante :

```
> export KAFKA_HOME = chemin vers le répertoire où est installer kafka
```

Se placer dans ***$KAFKA_HOME*** et lancer la commande suivante :

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

Zookeeper écoute généralement sur le port **2181** et kafka le **9092**. Ces ports peuvent être modifiés dans les fichiers de configuration.

Lancement de Kafka :

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

Dans Kafka, les messages sont regroupés par topics (sujets). Pour envoyer un message il faut au préalable créer un topic.

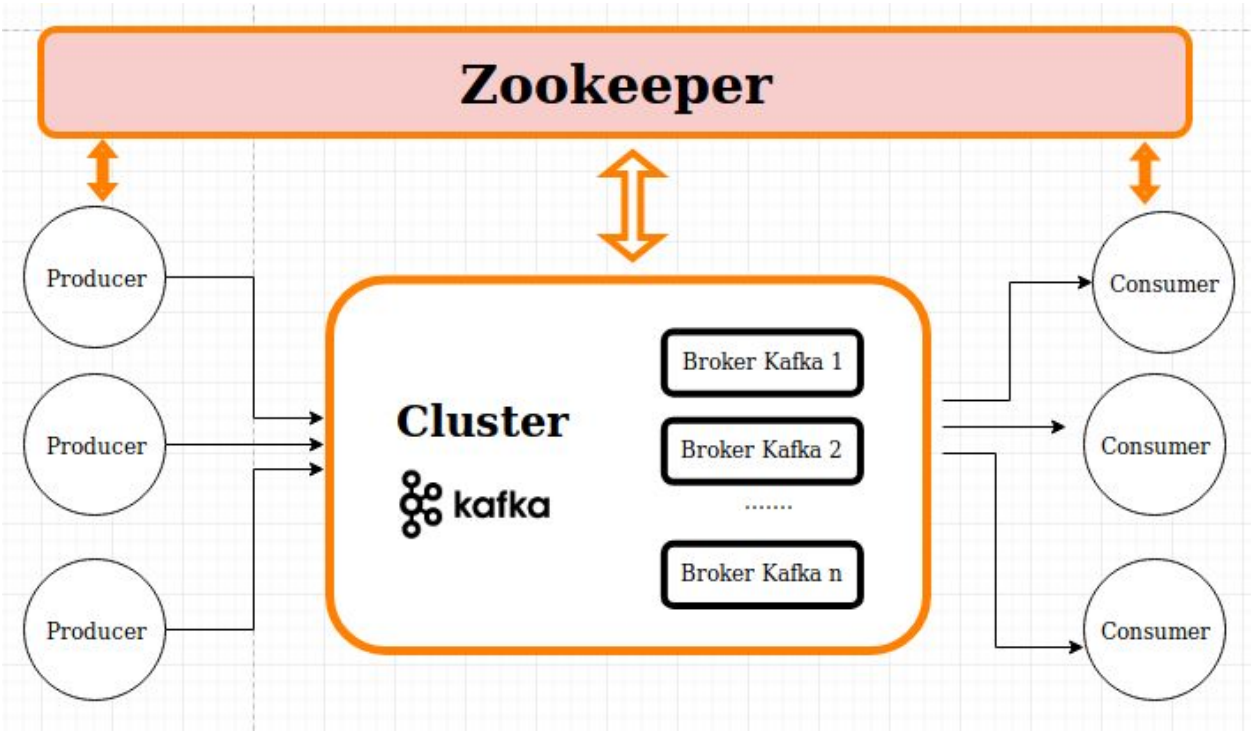
```
> ./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic nomTopic
```

Dans un premier temps on lance Kafka en local, ensuite nous le ferons sur une instance **EC2**. Dans la commande ci-dessus ***nomTopic*** est le nom du topic (à personnaliser).

On peut vérifier que le topic a bien été créé en listant les topics existant on peut alors afficher les propriétés du topic créé.

```
> ./bin/kafka-topics.sh --list --zookeeper localhost:2181
> ./bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic nomTopic
```

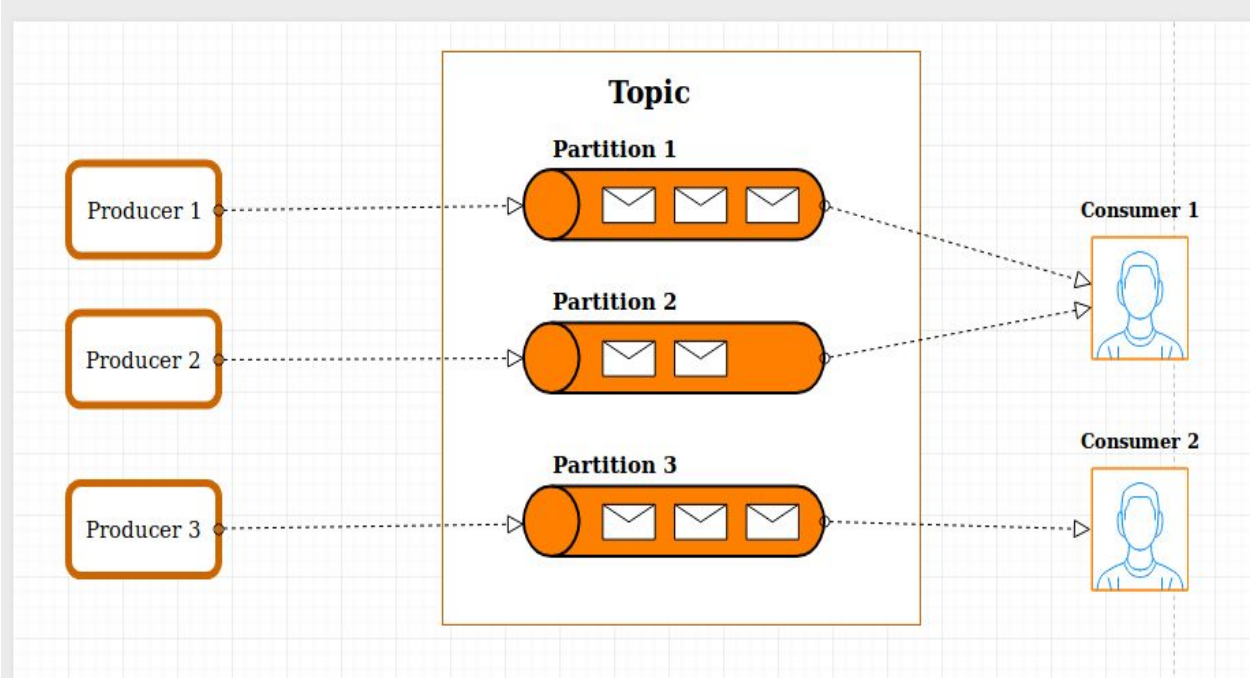
Dans la commande de création du topic, il y a un certain nombre d’option qu’on est obligé de renseigner. Kafka peut être déployé dans une architecture distribuée; le cluster peut ainsi, avoir plusieurs instances appelées **brokers.**



Comme sur **HDFS**, on peut répliquer les données sur plusieurs instances en précisant un facteur de réplication qui est l'option *replication-factor* de la commande de création de topic que nous avons défini à 1 car dans cet exemple, on ne travail que sur une seule instance en local. Les partitions sont l'équivalent des blocs de données sur HDFS. Ce sont donc ces partitions qui vont stocker les messages qui arrivent en temps réel et constituer la file de message Kafka. Que l'on soit en environnement distribué ou en local on peut avoir plusieurs partitions pour chaque topic.

Les messages stockés sur les partitions d'un topic Kafka peuvent arriver de différentes sources. Ces sources sont appelés **producers** (producteurs de message). Dans notre système, c'est l'application Java qui va jouer ce rôle. On peut cependant créer des producers dans une nouvelle console et écrire des messages dans la console, ainsi chaque ligne écrite sera traitée comme message et le retour à la ligne permettra à Kafka d'envoyer le message vers sa file et attendre qu'une autre application vienne consommer ces messages. L'application qui consomme les messages est appelée **consumer**. Les consumers peuvent, à l'image des producers, être nombreux. Dans notre projet ce sera Kylin qui va jouer ce rôle, mais dans un premier temps on va créer

un consumer dans une nouvelle console. L'architecture de Kafka ressemble au schéma ci-dessous.



Pour créer un producer dans la console, il faut exécuter le script correspondant en l'attachant au bon topic.

```
> ./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic nomTopic
```

L'option broker-list permet de lister l'ensemble des brokers du cluster kafka où est stocké le topic ***nomTopic***.

Pour créer un consumer il faut exécuter la commande ci-dessous.

```
> ./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic nomTopic
--from-beginning
```

On obtient alors l'ensemble des messages envoyés dans la file Kafka. L'option *from-beginning* permet de lire tous les messages à partir du début. Cependant si le consumer s'arrête pour une raison ou une autre, et qu'on le relance on est obligé de préciser à nouveau cette option pouvoir relire tous les messages, car si on ne précise pas l'option, on ne peut lire que les messages émis pendant que le consumer était connecté. Si on suppose que le consumer s'arrête pendant deux minutes, et que

pendant ce temps un producer envoie 5 nouveaux messages dans la file, en relançant le consumer sans l'option from-beginning, il ne verra pas ces messages. Pour pallier à ce problème, Kafka offre une solution consistant à affecter chaque consumer à un groupe, ce qui lui permettra de mémoriser l'emplacement des groupes dans ses files de messages. Il pourra ainsi savoir à quel endroit de la file il doit se placer lorsqu'un consumer d'un certain groupe se connecte pour ne lui renvoyer que les messages qu'il n'a pas lu.

Comme nous le disions plus haut, un topic peut être composé de plusieurs partitions chacune contient des messages différents. Lorsqu'un Producer émet un message, il choisit à quelle partition il veut l'ajouter (selon une stratégie de round robin ou selon de contenu du message, …) chaque partition représente une **FIFO**. Les messages stockés dans chaque partition sont transmis à chacun des groupes de consumers.

Pour garantir que les messages d'une partition sont traités en suivant leur ordre d'arrivé, il faut avoir au plus un consumer par partition dans chaque groupe. Donc le nombre de consumer doit être inférieur ou égal au nombre de partitions.

Pour assigner un consumer à un groupe, il faut rajouter à la commande kafka-console-consumer, l'option ***--consumer-property group.id=nomGroup***

On peut augmenter le nombre de partitions d'un topic après sa création en modifiant simplement sa structure.

```
> bin/kafka-topics.sh --alter --zookeeper localhost:2181 --topic nomTopic --partitions
nbPartitions
```

Pour créer plusieurs brokers (instances kafka), on modifie simplement les fichiers de configurations. Par exemple si on veut avoir deux brokers supplémentaires, on réplique le fichier config/server.properties et on édite les deux fichiers comme ceci :

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

On ouvre ensuite le fichier config/server-1.properties dans un éditeur de texte et on modifie l'identifiant du broker, le port utilisé ainsi que son fichier de log.

```
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/tmp/kafka-logs-1
```

Pareil pour le fichier config/server-2.properties

```
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/tmp/kafka-logs-2
```

Ensuite en lançant kafka-topics, on modifie le facteur de réplication. Ainsi nous aurons trois instances kafka qui tournent respectivement sur les ports 9092, 9093 et 9094.

Déploiement de Kafka sur EC2

Dans cette étape, nous allons créer une machine EC2 qui va contenir le cluster Kafka, une seconde machine qui va contenir le Producer (pour l'instant kafka-console-producer mais plus tard ce sera l'application Java) et une dernière qui va contenir le Consumer (pour l'instant kafka-console-consumer mais plus tard Kylin qui ne sera pas sur EC2 mais sur le cluster EMR).

Nous choisissons des serveurs Linux avec pour le serveur Kafka, une machine t3 medium et pour les deux autres des t2 micro pour limiter la consommation. Il faut ensuite assigner les deux machines (producer et consumer) au groupe de sécurité de Kafka ensuite définir dans la table de routage du groupe de sécurité de ces trois machines, une règle qui dit que toutes les machines du groupe peuvent communiquer.

Après la création de ces machines, on installe dans chacune, un jdk et un client kafka en définissant les variables d'environnement **JAVA_HOME** et **KAFKA_HOME** qu'on exporte dans le .bashrc. Une troisième variable est nécessaire sur les instances EC2 t2 micro où la mémoire est limitée afin de limiter également la mémoire utilisée par les processus kafka. Pour cela, on définit la variable **KAFKA_HEAP_OPTS="-Xmx256M -Xms128M"** ainsi, le tas de la JVM utilisera 128 Mo à son démarrage et ne dépassera pas 256Mo pour rester dans la limite des machines EC2 t2 micro.

Une fois ces installations terminées, on lance zookeeper dans l'instance contenant le broker Kafka en maintenant la connexion activée ce qui va permettre de dissocier l'état du processus à la session (si on ferme le terminal par exemple le processus continuera de s'exécuter grâce à nohup).

```
> nohup bin/zookeeper-server-start.sh config/zookeeper.properties &
```

Avant de lancer le producer et le consumer il faut au préalable éditer le fichier config/server.properties de chacune des instances, modifier la ligne zookeeper.connect=localhost:2181 et remplacer *localhost* par l'adresse IP de la machine EC2 du cluster kafka, car en mode distribué, on doit utiliser le zookeeper du broker kafka. Les instances Producer et Consumer ne sont pas des brokers kafka donc, la ligne broker.id=0 du fichier config/server.properties doit être commentée ou supprimée, car la file de message n'est pas dans ces deux instances mais seulement dans le broker kafka.

On peut enfin lancer le serveur kafka sur l'instance t3 medium, la console producer dans l'une des machines t2 micro et la console consumer dans l'autre et vérifier que la communication fonctionne correctement. C'est à dire qu'un message écrit dans le producer est envoyé au broker kafka qui le redirige vers le consumer.

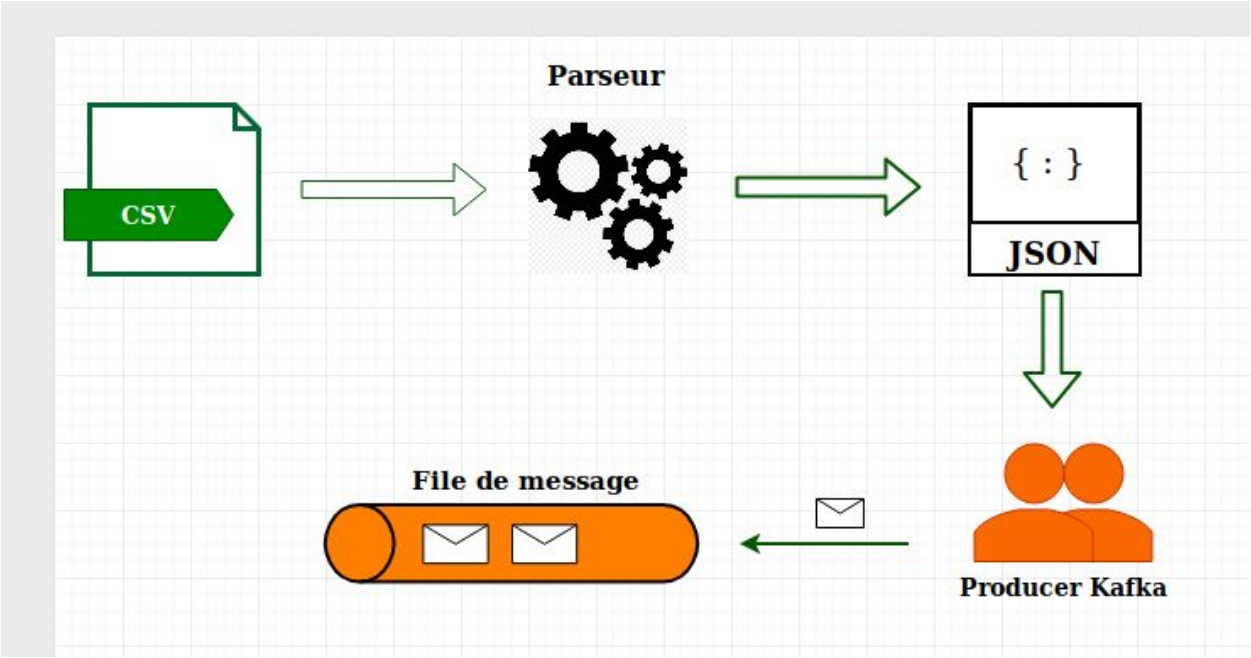
Insertion de données

Pour l'insertion de données en temps réel, nous avons développé une application Java. Cette application est le point d'entrée de notre système d'échange de données de Kafka vers Kylin. Nous avons fait le choix de travailler avec des données de vélos en libre service à Paris. On peut trouver dans les ***opendata*** toutes sortes de données. Cependant pour les contraintes de Kylin dont on parlera dans la section suivante, il nous faut des données contenant à la fois des informations temporelles, des données numériques et de catégories. Il existe une API sur le site opendata de Paris où on peut retrouver l'ensemble des stations de Vélib (Vélo libre-service). On aurait pu développer une application qui se connecte à cette API et qui récupère les informations qui nous intéresse et ensuite les envoyer à Kafka, mais nous avons fait le choix de télécharger un échantillon de donnée en fichier .***csv*** pour éviter notamment des requêtes **HTTP** trop nombreuses et trop coûteuses.

Nous avons donc nettoyé le fichier pour ne garder que les informations qui nous intéressent (identifiant de la station, capacité, nombre de vélo disponible, latitude, longitude, horodatage). Ainsi, un utilisateur pourra savoir à un instant donné si une station dispose d'un vélo en location et où se trouve la station ou alors lorsqu'il aura fini son service, il pourra savoir quelle station proche de lui a des places de libres pour qu'il rende son vélo.

Pour interagir facilement avec le topic Kafka et pour l'évolutivité de notre programme, nous avons défini une classe qui s'occupe uniquement de toute la configuration Kafka (url de connexion au serveur, nom du topic où les données seront envoyées, chemin d'accès au fichier .csv contenant les données de base, …) ainsi, notre programme reste un peu plus générique et facile à réutiliser.

Initialement, le programme lit le contenu du fichier .csv ligne par ligne et parse chaque ligne en document **json**. Ensuite, chaque document json est envoyé à kafka via le producer qu'on a défini dans le programme Java. L'ensemble de ces documents va alors constituer la file de message de kafka. Lorsque le programme aura envoyé toutes les données du fichier, il va rester en attente de nouvelles données et répéter la procédure.



C'est donc un programme qui s'exécute continuellement et l'utilisateur n'aura qu'à rajouter une ligne dans le fichier .csv et l'enregistrer, le programme recevra aussitôt les informations concernant la nouvelle donnée insérée et enverra uniquement cette donnée (pas la totalité du fichier) à la file de message kafka puis va continuer d'écouter l'insertion de nouvelles données.

Configuration de Kylin

Apache Kylin est un moteur d'analyse de données distribué open-source qui fournit une interface SQL et offre des fonctions de traitement OLAP sur un environnement Hadoop. A partir de Kylin v1.5.2, Il a intégré le streaming en le couplant avec Kafka. Il crée un cube toutes les 5 minutes par exemple pour optimiser le traitement streaming des données.

un cube OLAP est une structure de donnée qui stocke les données sous format

multi-dimensionnelle. Une mesure qui est une information numérique mesurant un aspect défini comme le stock d'une entreprise ou son chiffre d'affaires. La mesure est projetée sur plusieurs dimensions. Par exemple, un cube composé de 4 dimensions : Region, catégorie, année, produit. Ce cube est optimisé pour les groupements, agrégations et jointures. Il est souvent pré-calculé pour permettre d'accélérer ce type de requêtes.

Déploiement sur le EMR

Avant de pouvoir communiquer avec Kylin, il faut au préalable le déployer sur le cluster EMR. Pour cela, on exécute simplement le script appelé *deploy_kylin.sh* sur l'utilisateur hadoop avec la commande suivante :

```
> sh deploy_kylin.sh
```

La version de Kylin que nous avons installée est la 2.4.0 Avec les versions récentes, nous avions rencontré un problème avec Kylin qui ne pouvait pas trouver les dépendances de Kafka.

Kylin a besoin d'un client Kafka afin de récupérer les messages en streaming envoyés à un certain topic. Pour rendre cette communication possible, il faut définir la variable **$KAFKA_HOME** comme nous l'avons mentionné plus haut. La version de Kafka compatible à celle de Kylin est la 0.10.

Une fois l'exécution du script terminée et la variable d'environnement définie, il faut démarrer Kafka sur le cluster et redémarrer Kylin pour qu'il puisse en prendre connaissance et trouver les dépendances Kafka.

Exécuter ces commandes en se plaçant dans le répertoire d'installation de Kylin.

```
> ./bin/kylin.sh stop
> ./bin/kylin.sh start
```

Lorsque Kylin démarre, on peut accéder à son interface web sous le port **7070**. Le lien suivant permet de s'y connecter http://hostname:7070/kylin.

hostname doit être remplacer par l'adresse du cluster EMR (exemple : http://ec2-176-34-154-202.eu-west-1.compute.amazonaws.com:7070/kylin)

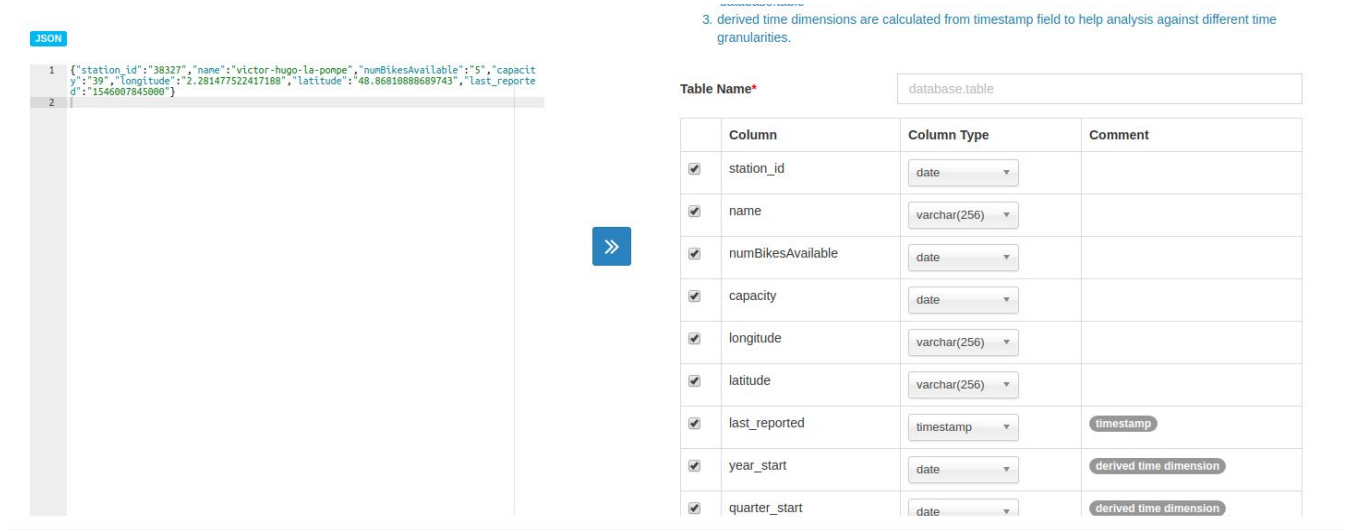
Ensuite on s'authentifie pour pouvoir accéder aux fonctionnalités de Kylin.

Alimentation

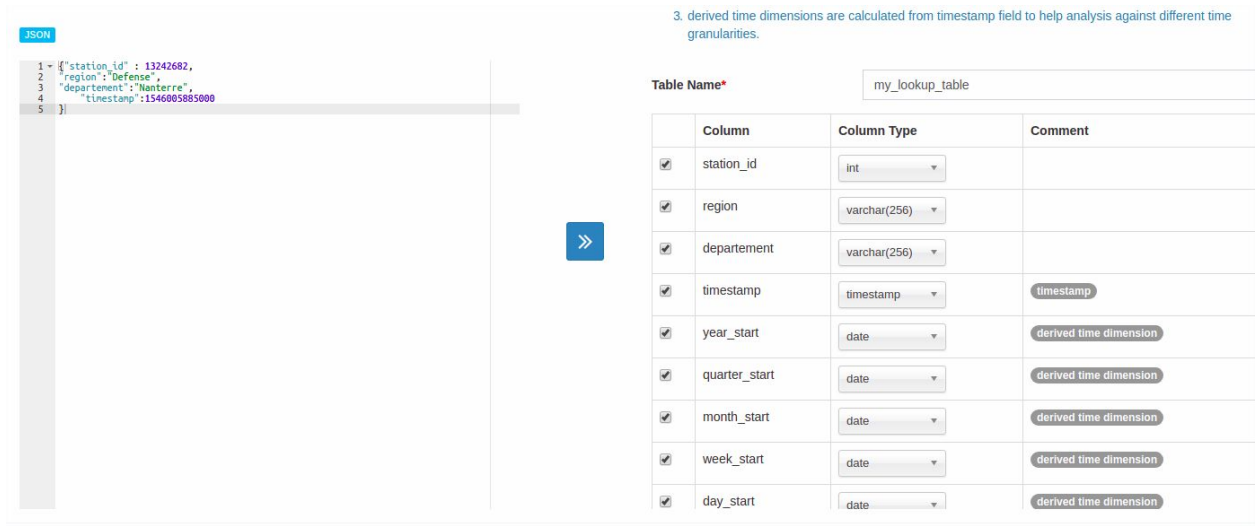
Dans cet exemple, on crée un nouveau topic appelé **bikesLocation** qui regroupe les messages envoyés par notre application.

On peut alors trouver l'ensemble des messages envoyés (sous forme de documents json) au topic lorsqu'on s'y connecte. On crée une nouvelle table "bikesLocation_table" qui constituera notre table de fait. Kylin a besoin d'un échantillon de ces messages pour détecter le schéma des données et construire la nouvelle table. Pour créer une table streaming. Dans l'onglet *Model*, choisir *Data source* puis *Add Streaming Table.*

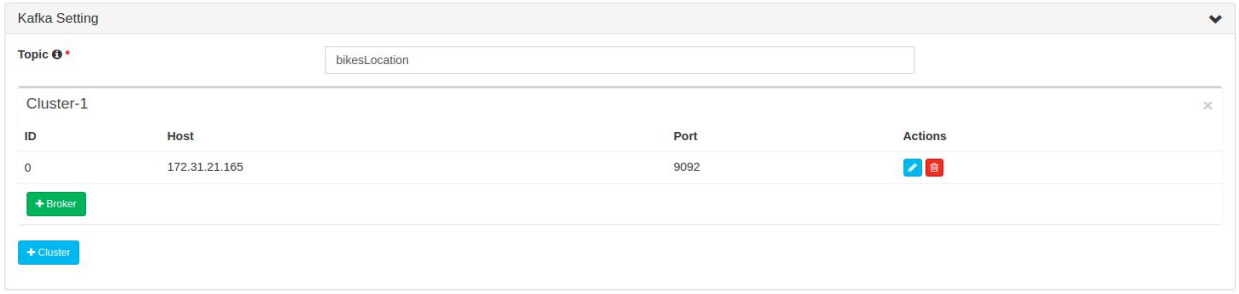
Pour transmettre l'échantillon, on crée une nouvelle table streaming et on colle un des messages reçu dans la console consumer, puis on clique sur le bouton ">>". On doit ensuite choisir une colonne représentant la donnée temporelle en spécifiant comme type **timestamp**. Dans notre cas cette colonne est appelée **last_reported**. Les colonnes temporelles doivent être en format timestamp UNIX en millisecondes (pas en seconde comme c'est le cas dans la plupart des outils).



On rajoute une table de dimension. Cette fonctionnalité est disponible à partir de Kylin v2.4.0.



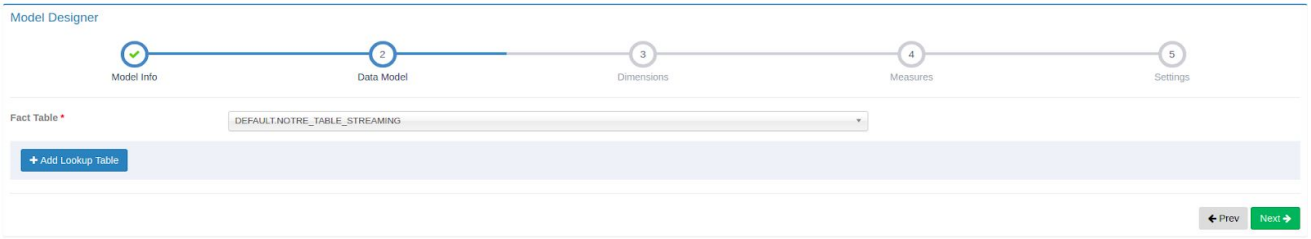
À l'étape suivante, il faut spécifier le topic duquel Kylin va récupérer les messages. On va également préciser la configuration des brokers : on précise l'adresse IP de chaque broker et le port utilisé.



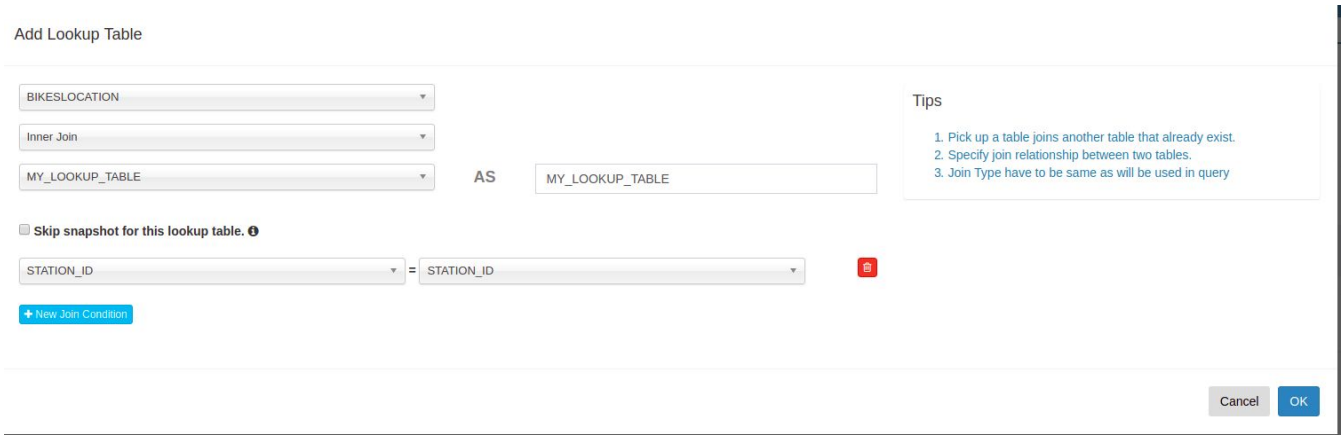
On peut afficher la liste des brokers dans une console grâce à la commande

```
> $KAFKA_HOME/bin/zookeeper-shell.sh 172.31.21.165:2181 <<< "ls /brokers/ids"
```

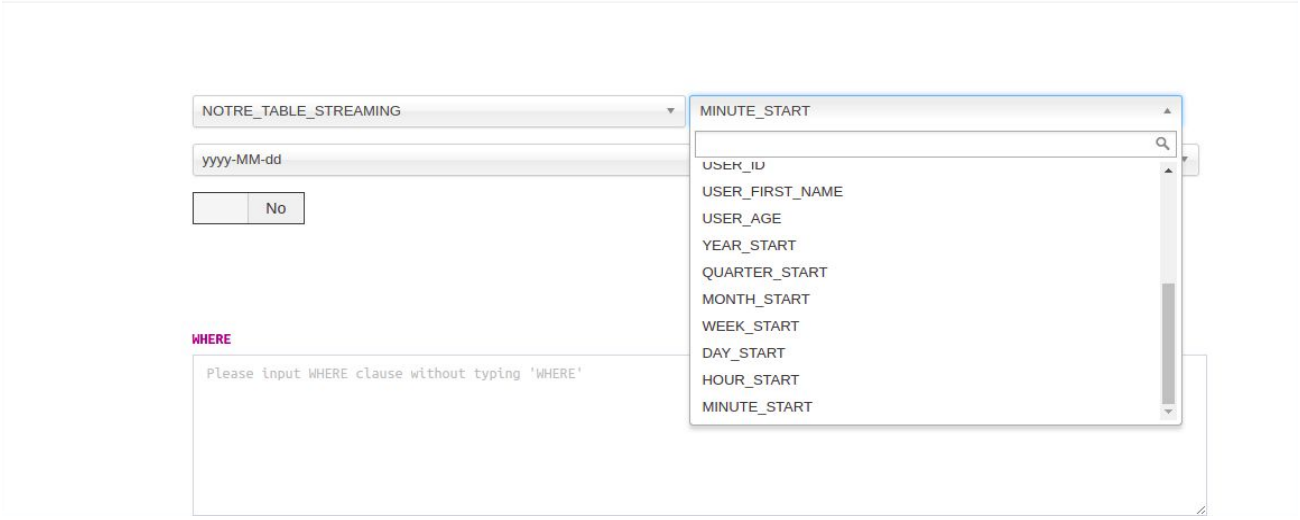
Pour créer un modèle, on sélectionne l'onglet *Model* puis *Models -> New -> New Model*. On choisit ensuite la table à utiliser.



On rajoute une table de dimension. On précise que la condition de jointure se porte sur la colonne STATION_ID

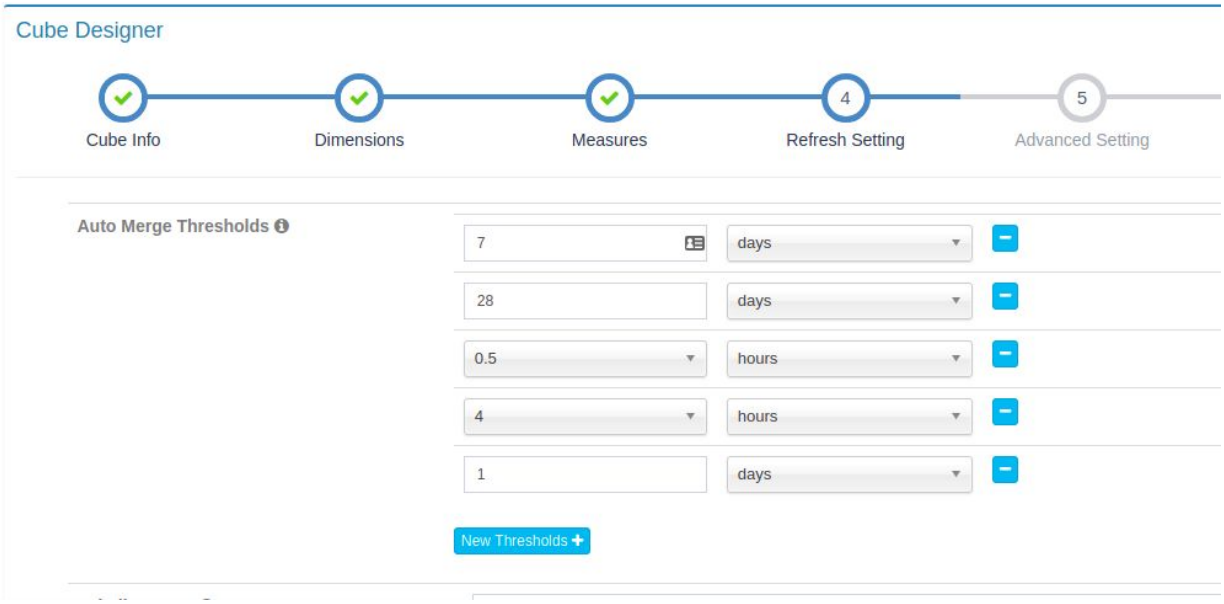


Lors de l'insertion de l'échantillon de données dans Kylin, Il faut spécifier la colonne de type timestamp. De nouvelles variables temporelles seront générées en fonction de l'horodatage, on obtiendra ainsi la minute, l'heure, le jour, la semaine, le trimestre et l'année. Pendant la construction du modèle, on spécifie la fréquence de traitement des données (à la minute ou à l'heure). Par exemple on peut choisir MINUTE_START

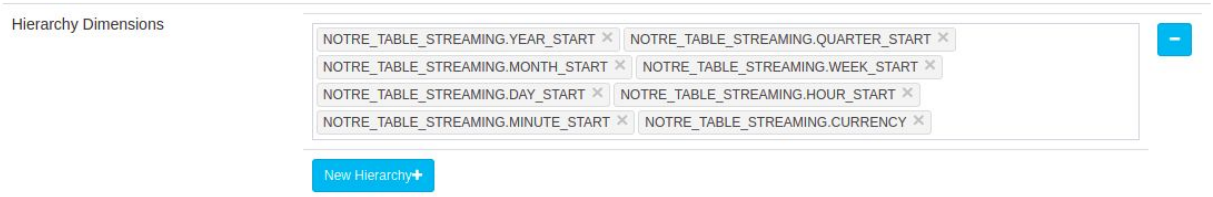


On crée ensuite un cube toujours dans l'onglet Model : Models -> New -> New Cube

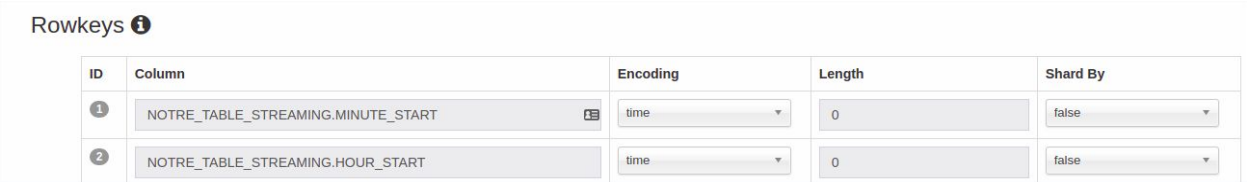
On ajoute toutes les dimensions du modèle en précisant le temps de rafraîchissement afin de faire un merge des segments du cube.



Dans la dernière phase de création d'un cube ("Advanced Settings"), on spécifie une hiérarchie des dimensions temporelles (année puis trimestre puis mois puis semaine, …) pour réduire la taille du cube et optimiser le temps d'exécution des requêtes.

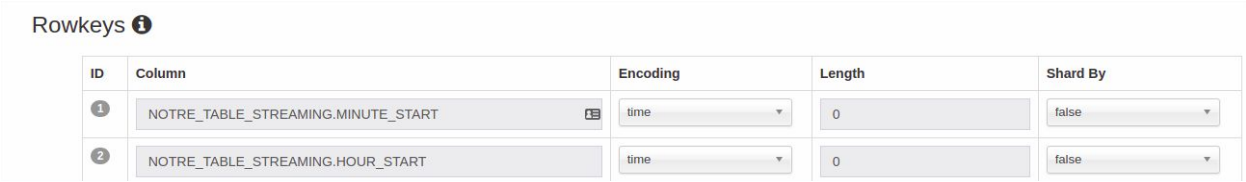


Puisqu'on va faire des requêtes sur les dimensions temporelles, on va les mettre en premier sur les rowkeys. Pour cela dans les champs Rowkeys, on choisit d'abord la dimension temporelle (à savoir MINUTE_START dans ce cas-ci).



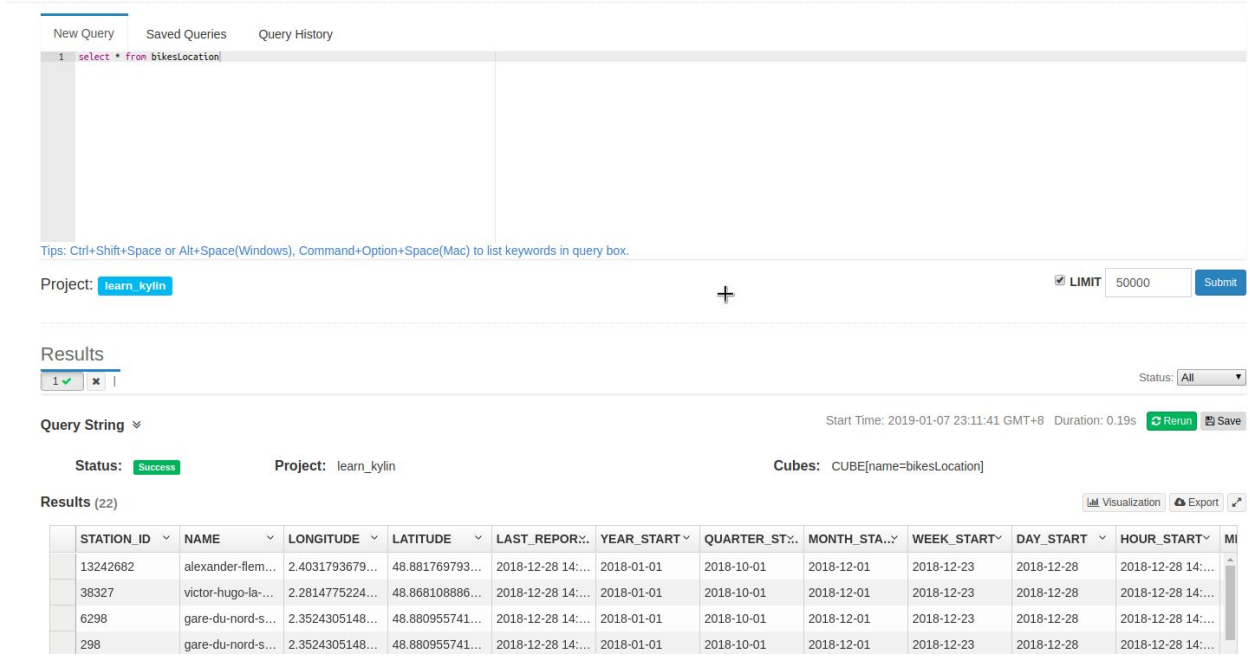
Réception de données

Après la création du cube, on sera capable de recevoir les données de la file de message Kafka.



On lance le build du cube pour les prendre en compte.

Une fois le build terminé, on pourra exécuter les requêtes SQL que nous voulons à partir de l'onglet "**Insight**"



Pour traiter les nouveaux messages, on va lancer un script planifié. Après le premier build qu'on va lancer manuellement sur l'interface web de Kylin, on va planifier des builds incrémentaux du cube. On peut choisir par exemple d'exécuter le script toutes les 5 minutes. Kylin enregistre le dernier offset après chaque build ce qui va lui permettre de savoir où il doit se placer dans la file de message Kafka pour le prochain build. On peut lancer le build grâce à l'API REST de Kylin et n'importe quel outil de planification comme cron sur linux.

```
> crontab -e */5 * * * * curl -X PUT --user ADMIN:KYLIN -H
"Content-Type:application/json; charset=utf-8" -d '{"sourceOffsetStart":0,
"sourceOffsetEnd": 9223372036854775807, "buildType": "BUILD"}'
http://localhost:7070/kylin/api/cubes/{your_cube_name}/build2
```

Dans la commande ci-dessus :

- **0** représente le dernier message que Kylin a lu,
- **9223372036854775807** (correspond à la plus grande valeur entière qu'une variable peut prendre), l'offset du dernier message sera donc forcément inférieur ou égal à ce nombre
- {**your_cube_name**} est le nom du cube qu'on veut builder.

Conclusion

Le système de transfert de messages de bout en bout fonctionne correctement. La file de message de Kafka a été implémentée avec succès. Notre application Java est capable de transmettre des données ajoutées dans le fichier .csv à la file Kafka qui transmet à son tour à Kylin les nouvelles données insérées. Cependant la version actuelle de Kylin n'est pas capable de prendre en compte les données au fur et à mesure qu'elle lui parviennent. Au lieu de cela, il fait un build à fréquence régulière pour voir s'il y a de nouvelles données à intégrer. Ce qui masque un peu le concept de temps réel et nous rapproche plus du micro-batch. Si l'application Java est capable d'envoyer des données en temps réel des données à Kafka avec une latence très faible, Kylin n'est pas encore capable de les traiter aussi rapidement. Les futures versions du moteur d'analyse Apache Kylin résoudrons probablement ce problème et permettront le déploiement d'applications en streaming plus efficace.