République Algérienne Démocratique & Populaire
Ministère de l'Enseignement Supérieur & de la Recherche Scientifique
Ecole Supérieure en Informatique 08 mai 1945 de Sidi Bel Abbes
Année universitaire : 2023-2024 **Semestre 2**
2 SC : IASD **Module : Big Data Technologies**

TP 7 : Stream Processing

Exercice 1 :

Remarque : Sauvegardez vos réponses dans un document pour la consultation.

Pour lancer **docker-compose** :

```
cmd> docker-compose up
```

Pour lancer un **ksqldb-cli** :

```
cmd> docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

Surveillance des modifications sur Netflix

Netflix investit chaque année des milliards de dollars dans du contenu vidéo. Avec autant de films et d'émissions de télévision en production en même temps, il est important de communiquer les mises à jour à divers systèmes lorsqu'un changement de production se produit (par exemple, les changements de date de sortie, les mises à jour financières, etc.). Notre objectif consiste à résoudre ce problème en utilisant ksqlDB.

Le but de cette application est simple. Nous devons consommer un flux de changements de production, filtrer et transformer les données pour le traitement, enrichir et agréger les données à des fins de création de rapports.

Le type de changement sur lequel nous nous concentrerons sera les changements de durée de saison d'une émission (par exemple, Stranger Things, saison 4 pourrait être initialement prévue pour 12 épisodes, mais pourrait être retravaillée en une saison de 8 épisodes). Cet exemple a été choisi car il modélise non seulement un problème réel, mais aborde également les tâches les plus courantes auxquelles vous serez confronté dans vos propres applications ksqlDB.

L'architecture de l'application de suivi des changements est présenté dans la Figure ci-dessous.

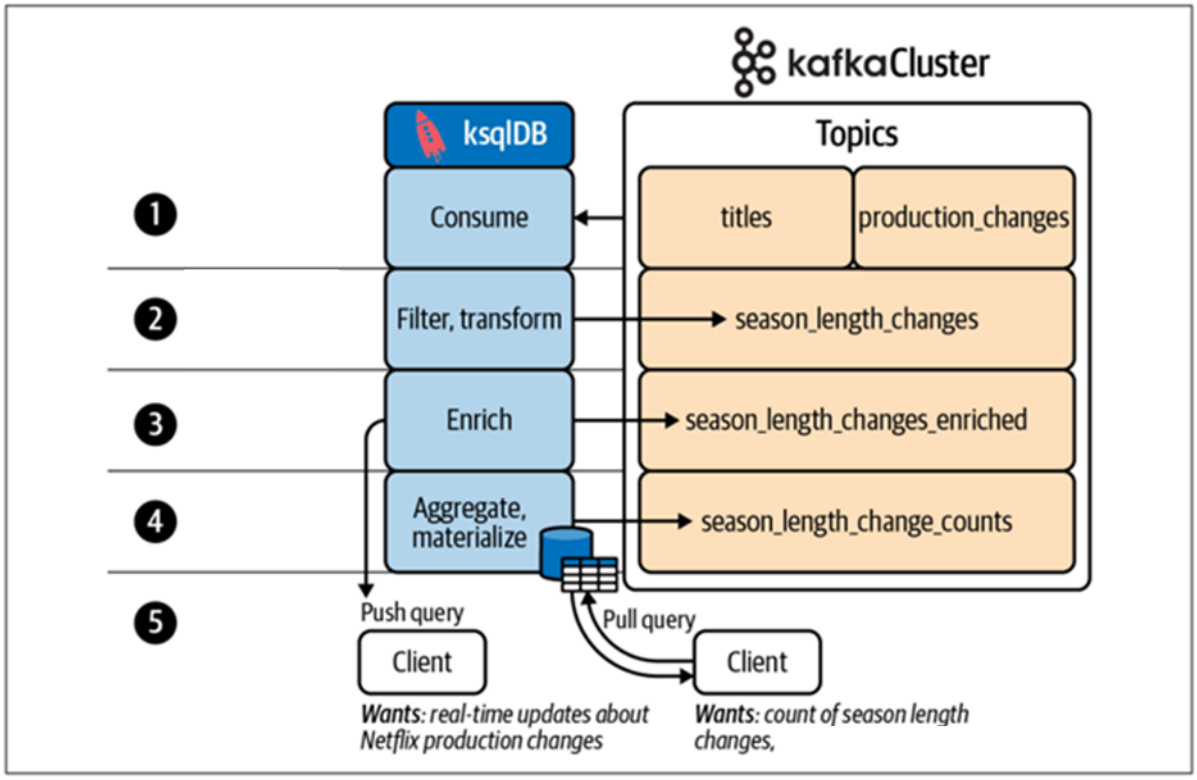

Figure 1 Architecture de l'application de suivi des modifications sur Netflix.

Description des étapes :

Etape 1 : Notre application va lire à partir de deux sujets (topics) :

- Le topic **titles** qui contient les titres des vidéos (films et des émissions de télévision) qui sont hébergés sur le service Netflix.
- Le topic **production_changes**. On écrit dans ce topic chaque fois qu'il y a un changement (de budget, de date de sortie, de durée de saison, etc.) pour un titre qui est en production.

Etape 2 : Après avoir consommé les données de nos topics sources, nous devons effectuer un prétraitement de base afin de préparer les données de **production_changes** pour l'enrichissement. Le flux prétraité, qui ne contiendra que les changements de durée de saison, sera écrit dans un topic Kafka nommé **season_length_changes**.

Etape 3 : Nous effectuerons ensuite un enrichissement de données sur les données prétraitées. Plus précisément, nous allons joindre le flux **season_length_changes** avec les données de **titles** pour créer un enregistrement combiné.

Etape 4 : Ensuite, nous effectuerons des agrégations fenêtrées et non fenêtrées pour compter le nombre de changements sur une période donnée. La table résultante sera matérialisée.

Etape 5 : Enfin, nous rendrons les données enrichies et agrégées disponibles à deux types de clients différents. Le premier client recevra des mises à jour continues via des requêtes push. Le deuxième client effectuera des requêtes pull de courte durée (Les requêtes pull renvoient l'état actuel au client, puis se terminent. En ce sens, elles ressemblent beaucoup plus à une instruction SELECT exécutée sur un SGBD normal).

Source Topics

Si vous avez des données dans un topic Kafka et vous voulez les traiter en utilisant ksqlDB, alors il faut examiner les données présentes dans ce topic, puis déterminer comment modéliser les données dans ksqlDB.

Dans notre cas, nous avons deux topics sources principaux : "titles" et "production_changes". Un exemple d'enregistrement dans chaque topic source est présenté comme suit :

Topic titles :

```
{
  "id": 1,
  "title": "Stranger Things"
}
```

Topic production_changes :

```
{
    "rowkey": "key1",
    "title_id": 1,
    "change_type": "season_length",
    "before": {
            "season_id": 1,
            "episode_count": 12
    },
    "after": {
            "season_id": 1,
            "episode_count": 8
    },
    "created_at": "2021-02-08 11:30:00"
}
```

Types personnalisés

1) Créer en ksqldb le type personnalisé **season_length** qui représente le type des deux attributs **before** et **after**.

Une fois le type créé, vous pouvez utiliser la requête **SHOW TYPES** pour le visualiser :

```
ksql> SHOW TYPES ;
```

Si on veut supprimer notre type personnalisé, on exécute l'instruction suivante :

```
ksql> DROP TYPE season_length;
```

Les collections

2) Choisir le type de collection le plus approprié pour **titles** et pour **production_changes** en justifiant votre choix.
3) Créer la collection **titles** et la collection **production_changes,** sachant que :
   - **id** est l'identifiant de la collection **titles.**
   - **rowkey** est l'identifiant de la collection **production_changes**.
   - La colonne **created_at** de **production_changes** contient le timestamp que ksqlDB doit utiliser pour les opérations temporelles (e.g., windowed aggregations and joins).
   - on suppose que le nombre de partition = 4 dans les deux collections.

Affichage des Streams et des Tables

La syntaxe de l'instruction qui permet d'afficher des informations sur tous les streams et les tables actuellement enregistrés est (**LIST** et **SHOW** sont interchangeables) :

```
{ LIST | SHOW } { STREAMS | TABLES } [EXTENDED];
```

Exemple :

```
ksql> SHOW TABLES ;

ksql> SHOW STREAMS ;
```

Si vous avez besoin de plus d'informations sur les collections, vous pouvez utiliser la variante **EXTENDED** de l'instruction **SHOW**, comme illustré ici :

```
ksql> SHOW TABLES EXTENDED ;

ksql> SHOW STREAMS EXTENDED ;
```

Description des streams et des tables

La description des collections est similaire à la commande **SHOW**, mais elle opère sur une seule instance de stream ou de table à la fois. La syntaxe pour décrire un stream ou une table est la suivante :

```
DESCRIBE <identifier> ;
```

<identifier> est le nom d'un stream ou d'une table. Par exemple :

```
ksql> DESCRIBE titles ;
```

Insérer des valeurs

La syntaxe pour insérer des valeurs dans une collection est la suivante :

```
INSERT INTO <collection_name> [ ( column_name [, ...]] ) ]
VALUES (
   value [,...]
);
```

4) Insérer des données dans les deux collections **titles** et **production_changes**.

Les requêtes

Dans cette section, nous examinerons quelques méthodes de base pour filtrer et transformer les données dans ksqlDB. Rappelez-vous qu'à ce stade, notre application de suivi des modifications consomme des données provenant de deux topics Kafka différents : titles et production_changes. Nous avons déjà terminé l'étape 1 de notre application (voir la Figure 1) en créant des collections pour chacun de ces topics.

Ensuite, nous aborderons l'étape 2 de notre application, qui nous oblige à filtrer production_changes pour les changements de durée de saison uniquement, à transformer les données dans un format plus simple et à écrire le flux filtré et transformé dans un nouveau topic appelé **season_length_changes**.

Remarque : Pour la lecture des données depuis le début de nos topics, il faut affecter la valeur '**earliest**' à la propriété '**auto.offset.reset**' :

```
SET 'auto.offset.reset' = 'earliest';
```

5) Ecrire une requête Push qui permet de retourner tous les changements de production créés avant le 2023-04-14 à 12:00:00.
6) Ecrire une requête Push qui permet de retourner tous les enregistrements de **production_changes** où la colonne "change_type" commence par le mot "season".
7) Créer à partir de **production_changes** un stream dérivé nommé **season_length_changes**, qui ne contient que les changements de production de type **season_length**.
   - On suppose que **season_length_changes** écrit dans un topic qui porte le même nom, le nombre de partition = 4, le nombre de réplique = 1 et les messages stockés dans le topic sont encodés en JSON.
   - season_length_changes contient les colonnes :
     - ROWKEY, title_id, created_at
     - season_id : cette colonne reçoit la valeur de after->season_id,

       Si after->season_id est null, season_id reçoit la valeur de before->season_id.

     - old_episode_count : cette colonne reçoit la valeur de before->episode_count

- new_episode_count : cette colonne reçoit la valeur de after->episode_count

8) Ecrire une requête Push qui permet de retourner les titres de toutes les vidéos contenues dans **season_length_changes.**
9) Créer le stream **season_length_changes_enriched** qui contient tous les attributs de **season_length_changes** plus l'attribut **title** de la collection **titles**.
    - La colonne **created_at** de **season_length_changes_enriched** contient le timestamp que ksqlDB doit utiliser pour les opérations temporelles.
10) Créer la table dérivée **season_length_change_counts** à partir de **season_length_changes_enriched.**
    - La table **season_length_change_counts** représente le nombre de changement effectués pour chaque **title**.
    - De plus, elle contient l'attribut **episode_count** qui représente la dernière valeur de **new_episode_count**.
    - La table **season_length_change_counts** est créé à partir d'un WINDOW TUMBLING d'une heure.

Exercice 2 :

Présentez un exemple d'utilisation de Spark Streaming (Structured Streaming) avec Apache Kafka.

Remarque : le code Python doit inclure : (1) la création d'un Streaming Dataframe pour lire les données à partir de Kafka, (2) un exemple d'exécution d'une requête en illustrant la mise à jour du résultat après chaque insertion.

Conseil : afin d'éviter les situations de blocage, utiliser un fichier .py au lieu d'un notebook et réaliser l'exécution dans un invite de commandes.