Cahier de TP
« KafkaStream et kSQL »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- JDK21+, Maven
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode
- Docker, Git

Images docker utilisées :

- docker.io/bitnami/kafka:3.7
- docker.redpanda.com/redpandadata/console:latest
- confluentinc/cp-schema-registry

Table des matières

Atelier 0 : Cluster Kafka et production de message

0.1 Démarrage cluster

Visualiser le fichier ***docker-compose.yml***

Il permet de démarrer un cluster 3 nœuds, une console RedPanda et un Registre de schémas

Démarrer cette stack avec :

```
docker compose up -d
```

Observer les logs de démarrages d'un nœud

```
docker logs -f kafka-0
```

Accéder à la ***Redpanda Console*** : http://localhost:9090

0.2 Production de message

Créer un ***topic*** position avec 5 partitions, 3 répliques et un ***min-insync-replica*** de 2

Le programme fourni ***Tps/0_KafkaProducer/producer*** permet de générer des messages représentant la position courante d'un coursier.

Le programme prend 3 arguments :

- *nbThreads* ~ le nombre de coursier
- *nbMessages*
- *sleep* : l'intervalle entre chaque envoi de poisition

Exécuter dans une console :

```
mvn clean package
```

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 100 10000 100
```

Ce qui représente une flotte de 100 coursiers qui envoient 10000 fois leur position tous les 100 ms

Atelier 1 : Première application et Serde

L’objectif de cet atelier est de transformé le format de sérialisation Json du topic position en un format de sérialisation Avro

1.1 Développement

Récupérer le projet fourni, observer les dépendances et le Serde Json

Exécuter une compilation pour générer :

```
mvn generate-sources
```

Compléter la classe principale ***PositionStream*** :

- Utiliser ***SpecificAvroSerde*** pour se créer un Serde Avro capable de sérialiser la classe Coursier générée
- Instancier une classe ***Properties*** et positionner les configurations nécessaires pour le Stream
- Construire une topologie avec en entrée le topic position et en sortie un topic position-output
- Instancier un KafkaStream
- Permettre une interruption du programme et démarrer le Stream

Tester et lorsque le programme fonctionne visualiser les traces en particulier :

- La création des consommateurs
- Leur affectation de partitions
- Les tâches assignées

1.2 Scalabilité

Augmenter le nombre de threads ou démarrer plusieurs la même instance. Vous pouvez construire un jar exécutable via :

```
mvn clean package
```

Atelier 2 : Configuration ExactlyOnce

Parfaire la configuration précédente, en fixant :

- le degré de réplication des topics internes
- la garantie Exactly Once
- un max.poll.records à 5000
- Le gestionnaire d’exception de sérialisation à ***LogAndContinueExceptionHandler***

Tester la consommation de message avec de la tolérance panne :

- Démarrer 2 instances, en arrêter une puis la redémarrer
- Arrêter puis redémarrer un broker

Vérifier qu’aucun message n’est perdu ni doublonné

Atelier 3 : Opérateurs stateless

Dans un premier temps, transformer les informations de la position d'un coursier en arrondissant la longitude et la latitude à 1 décimal.

Tester, regarder les timestamp des messages

Supprimer le topic de sortie et le groupe de consommation associé au stream

Et compléter le stream en inversant les clés et Valeurs (La position du coursier devient la clé, l'id du coursier la valeur)

Tester

Insérer l'opérateur ***branch*** pour créer 2 topic de sortie un contenant les positions dont la latitude est supérieur à

45.0 et l'autre le reste

Finalement modifier votre code pour obtenir 3 topics :

- Un topic avec la conversion JSON → Avro
- 2 topics avec la séparation sur la latitude

Tester

Atelier 4: Agrégations

4.1 Count

Sur les branches nord/sud précédentes, effectuer une agrégation de type Count(), i.e compter le nombre de coursier ayant passé à une position donné.

Tester

Modifier en utilisant une fenêtre temporelle de 5 minutes.

4.2 Aggregate

Nous voulons dorénavant avoir en temps-réel, la liste des coursiers associés à une position.

Implémenter via une fonction aggregate, la valeur agrégée étant une liste

- Partir du topic position
- Transformer la valeur en Position en arrondissant la latitude et la longitude au dixième prêt
- Transformer en une table contenant la dernière Position du Coursier
- Grouper la table par Position
- Aggréger en utilisant un adder et substractor

Atelier 5 : Jointures

5.1 Jointure Kstream → KStream

Le programme fourni permet d'alimenter un topic ***coursierStatut*** avec la structure de données suivantes :


```
{
"id": "0",
"statut": "REPOS",
"firstName": "Antoine",
"lastName": "Thibau"
}
```


Le programme s'exécute comme le programme alimentant position :

```
mvn clean package
```

```
java -jar target/statut-0.0.1-SNAPSHOT-jar-with-dependencies.jar <nbCoursiers>
<nbMessages> <frequenceEnvoi>
```

Dans l'application KafkaStream, implémenter une jointure sur ses 2 topics et déverser les messages générées dans un topic de sortie ***coursiers-statut-position.***

- Commencer par créer les topics avec le même nombre de partitions
- Implémenter la topologie KafkaStream avec une fenêtre d'agrégation de 5min.
- Démarrer les 2 programmes d'alimentation des topics d'entrée en simultané avec 10 coursiers, 100 messages et des fréquences d'envoi sensiblement différente
- Visualiser les messages générés et le nombre de message.

5.2 Jointure Kstream → KTable

Transformé le code précédent en une jointure Ksream → KTable.

Le stream est alimenté par le topic ***position***

La table par ***coursierStatut***

Effacer le topic de sortie et le groupe de consommation.

Réexécuter le programme et visualiser le nombre de messages générés.

5.3 Jointure Kstream → KGlobalTable

Transformé le code précédant en une jointure Kstream → KTable

Supprimer le topic ***courierStatut*** puis le recréer avec un nombre de partitions différents.

Alimenter le avec 1000 messages

Effacer le topic de sortie et le groupe de consommation lié à l'application KafkStream

Réexécuter le programme et visualiser le nombre de messages générés.

Atelier 6. Requêtes interactives

Compléter le programme précédent afin qu'il affiche sur la console toutes les secondes le statut du coursier ayant l'id 0

Pour démarrer une tâche toutes les secondes vous pouvez utiliser le code suivant avant de démarrer le stream

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

// Définition de la tâche périodique
Runnable queryTask = () -> {
    try {
        // Récupération du store d'état

        // Exécution de la requête - exemple : récupération d'une clé spécifique


        // Traitement du résultat

    } catch (Exception e) {
        System.err.println("Erreur lors de l'interrogation de la KTable : " +
e.getMessage());
    }
};
// Planification de la tâche toutes les secondes
scheduler.scheduleAtFixedRate(queryTask, 0, 1, TimeUnit.SECONDS);
```

Atelier 7 : ksqlDB

7.1 Démarrage

Démarrer la stack avec le docker compose fourni.

Ensuite, démarrer une console interactive ksql-cli

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

Créer un stream associé à un topic

```
CREATE STREAM riderLocations (profileId VARCHAR, latitude DOUBLE, longitude
DOUBLE) WITH (kafka_topic='locations', value_format='json', partitions=1);
```

Créer une table contenant les derniers emplacements des riderLocation

```
CREATE TABLE currentLocation AS
SELECT profileId,
LATEST_BY_OFFSET(latitude) AS la,
LATEST_BY_OFFSET(longitude) AS lo
FROM riderlocations
GROUP BY profileId
EMIT CHANGES;
```

Créer une table contenant les données agrégées (liste de coursier, nombre) par distance par rapport à un point d'origine

```
CREATE TABLE ridersNearMountainView AS
SELECT ROUND(GEO_DISTANCE(la, lo, 37.4133, -122.1162), -1) AS distanceIn-
Miles,
COLLECT_LIST(profileId) AS riders,
COUNT(*) AS count
FROM currentLocation
GROUP BY ROUND(GEO_DISTANCE(la, lo, 37.4133, -122.1162), -1);
```

Exécuter une PUSH QUERY

-- Mountain View lat, long: 37.4133, -122.1162

```
SELECT * FROM riderLocations
WHERE GEO_DISTANCE(latitude, longitude, 37.4133, -122.1162) <= 5 EMIT
CHANGES;
```

Démarrer une autre session :

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES
('c2309eec', 37.7877, -122.4205);INSERT INTO riderLocations (profileId,
```

```
latitude, longitude) VALUES
('18f4ea86', 37.3903, -122.0643);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES
('4ab5cbad', 37.3952, -122.0813);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES
('8b6eae59', 37.3944, -122.0813);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES
('4a7c7b41', 37.4049, -122.0822);
INSERT INTO riderLocations (profileId, latitude, longitude) VALUES
('4ddad000', 37.7857, -122.4011);
```

Puis exécuter une PULL QUERY

```
SELECT * from ridersNearMountainView WHERE distanceInMiles <= 10;
```