Kafka

Système de traitement en Spring Boot axé [BATCH]

ASKIA ABDEL KADER – BATTACHE NASSIM

Introduction

Notre objectif était d'implémenter un système de traitement en batch efficace, capable de traiter de grandes quantités de données en parallèle, afin d'optimiser les performances. Nous avons opté pour une architecture distribuée, en utilisant **Spring Batch** pour le traitement en batch, et **Kafka** pour la communication entre les différents composants de notre système.

Mise en œuvre détaillée

TME 1 :

La première étape a été d'analyser l'application existante et de comprendre son fonctionnement. Cela nous a permis d'identifier les modules qui pourraient être convertis en microservices indépendants.

Dans un deuxième temps, nous avons commencé à implémenter ces microservices. Pour ce faire, nous avons utilisé **Spring Boot**, un cadre largement utilisé qui simplifie le développement de microservices en Java.

Nous avons également utilisé Spring Batch pour orchestrer le traitement des données par lots, qui est une fonctionnalité commune dans de nombreuses applications. Spring Batch permet de traiter de grandes quantités de données de manière efficace et fiable.

L'une des tâches principales était de migrer la gestion des données de l'application. Pour cela, nous avons utilisé le framework Spring Data **JPA** pour interagir avec la base de données. Nous avons également créé des repositories pour chaque entité, ce qui permet de gérer facilement les opérations **CRUD**.

TME2 :

Notre système est divisé en deux parties principales : le **Master** et les **Workers**. Le Master est responsable de l'initialisation du processus, de la division de la tâche en sous-tâches, et

de l'assignation de ces sous-tâches aux Workers. Les Workers, quant à eux, effectuent le traitement réel des données.

Pour parvenir à une exécution parallèle efficace, nous avons utilisé une technique appelée partitionnement qu'on appelle donc le « Remote Chunking ». Le partitionnement consiste à diviser une grande tâche en plusieurs sous-tâches plus petites qui peuvent être exécutées en parallèle. Dans notre cas, le Master est responsable du partitionnement du fichier d'entrée et de l'assignation des partitions aux Workers.

Pour la communication entre le Master et les Workers, nous avons utilisé Kafka. Kafka est une plateforme de streaming distribuée qui permet une communication robuste et à haute vitesse entre les différentes parties de notre système. Le Master envoie les tâches à accomplir aux Workers via Kafka, et reçoit en retour les résultats du traitement.

Comme mentionné précédemment, la communication entre le Master et les Workers est assurée par Kafka, qui sert de canal de communication pour l'échange de messages. Dans ce contexte, nous avons défini deux flux principaux: inboundFlow et outboundFlow.

InboundFlow est responsable de la réception des demandes provenant du Master. En d'autres termes, il s'agit du processus par lequel le Worker reçoit une partition de travail à traiter. Le flux d'entrée est configuré pour écouter un certain topic Kafka. Lorsqu'un message est reçu sur ce topic, il est transformé en une demande de traitement et transmis à **l'ItemReader** correspondant. Dans notre cas, l'ItemReader est configuré pour lire les données d'un fichier texte, lettres.txt.

OutboundFlow, d'autre part, est responsable de l'envoi des réponses de retour au Master. Une fois que le Worker a terminé le traitement d'une partition, le résultat est transmis au flux de sortie, qui se charge de l'envoyer au Master via Kafka. Pour ce faire, il utilise un **KafkaProducerMessageHandler** pour publier le message sur le **topic** Kafka correspondant.

Ces flux d'entrée et de sortie fonctionnent de manière asynchrone et sont essentiels à l'orchestration du traitement en batch distribué. Ils permettent une communication robuste et efficace entre le Master et les Workers, tout en assurant que les Workers peuvent travailler en parallèle sur leurs partitions respectives.

Pour ce qui est du fichier lettres.txt, il s'agit du fichier d'entrée que nous avons utilisé pour tester notre système. Il est lu par l'ItemReader, qui découpe les données en partitions que le Master attribue ensuite aux Workers. L'ItemReader utilise une approche multithread pour lire le fichier, ce qui permet une lecture plus rapide et plus efficace des données. Chaque ligne du fichier est considérée comme une unité de travail distincte, et le fichier est découpé en partitions en fonction du nombre de Workers disponibles. Chaque Worker reçoit une partition et traite les lignes qui lui sont attribuées.

Ainsi, la combinaison de Kafka pour la communication, de Spring Batch pour le traitement en batch, et du partitionnement pour la parallélisation nous a permis de créer un système capable de traiter de grandes quantités de données de manière rapide et efficace.

Résolution de problèmes

Au cours du projet, nous avons rencontré quelques défis. L'un d'eux était une erreur de **NullPointerException** dans notre Job Spring Batch, causée par un mauvais agencement de nos **beans** Spring. En effet, le repository nécessaire pour l'opération de sauvegarde dans le **tasklet** n'était pas correctement injecté. Après avoir enquêté sur le problème, nous avons découvert que l'annotation **@RequiredArgsConstructor** générée par **Lombok** en combinaison avec notre constructeur personnalisé était la cause du problème. Une fois que nous avons résolu ce problème, le **job** a pu s'exécuter sans erreurs.

Résultats et conclusions

Notre système s'est avéré capable de traiter efficacement de grandes quantités de données en parallèle. Cela a été démontré par les tests que nous avons effectués, où le système a pu traiter un **label** d'entrée en un temps nettement inférieur à celui qu'aurait pris un traitement séquentiel. Cependant pour la seconde partie qui fait intervenir le traitement par un fichier texte. Il y a eu un problème du fait de la communication entre les deux serveurs. Par manque de temps c'était très dure de trouver l'erreur.

Au cours de ce projet, nous avons appris à mettre en œuvre un système de traitement en batch distribué et à utiliser Kafka pour la communication dans un tel système. Nous avons également appris à gérer les défis posés par la mise en œuvre du partitionnement et du traitement en parallèle.

Il y a plusieurs améliorations possibles à notre projet. Nous pourrions envisager d'ajouter une fonctionnalité de reprise après panne, qui permettrait au système de reprendre le traitement là où il s'est arrêté en cas de panne.

En conclusion, ce projet a été une **expérience** enrichissante qui nous a permis de mieux comprendre les défis et les solutions associées à la mise en œuvre d'un système de traitement en batch **distribué**.