Indexation avec table de hachage

Présentation du contexte

Vous développez un système de stockage que l'on retrouve dans certains SGBD du type NoSQL, notamment ceux basés sur le modèle de données clé-valeur. Ce système est pertinent lorsque plusieurs valeurs sont associées à une même clé, c'est par exemple le cas si on souhaite compter le nombre d'occurrence d'un élément identifié par une clé unique ou obtenir la valeur la plus récente associée à une clé donnée. Dans la Figure 1, on retrouve 3 clés (abc,def et ghi). Il y a 2 valeurs différentes (def:732756769******* et def:54434194********) pour la clé "def". Dans la figure, on considère que la ligne contenant les valeurs représente le fichier persistée sur disque.

N.B.: J'ai intégré la clé ('def') dans la valeur pour simplifier les explications sur le modèle de stockage.

L'objectif de l'approche de stockage est de garder un archivage de toutes les données et de pouvoir retrouver efficacement la dernière valeur d'une clé donnée. Pour cela, il nous faut une solution de stockage de fichiers sur disque ET une solution d'indexation. Cette dernière sera de préférence en mémoire principale (plus efficace que sur disque en terme d'entrées/sorties). On garde donc l'index en mémoire et les données sont persistées sur disque (des fichiers pour l'archivage).

L'approche de stockage consiste à ajouter des données à la fin d'un fichier (approche qualifiée d'*append-only* en anglais). Une stratégie d'indexation simple correspond alors à conserver une table de hachage en mémoire dans laquelle chaque clé correspond à un décalage d'octets dans le fichier de données, c'est-à-dire l'endroit où la valeur peut être trouvée. Dans la suite, on considère que les enregistrements (les valeurs des paires clé/valeur) sont d'une taille fixe (20 octets dans notre exemple) et que l'on peut avoir plusieurs entrées pour une même clé. Dans la figure 1, le tableau correspond à la table de hachage, on y observe 3 clés (abc, def et ghi). Nos enregistrements sont stockés les uns à la suite des autres dans un fichier (souvent nommé *log segment*).

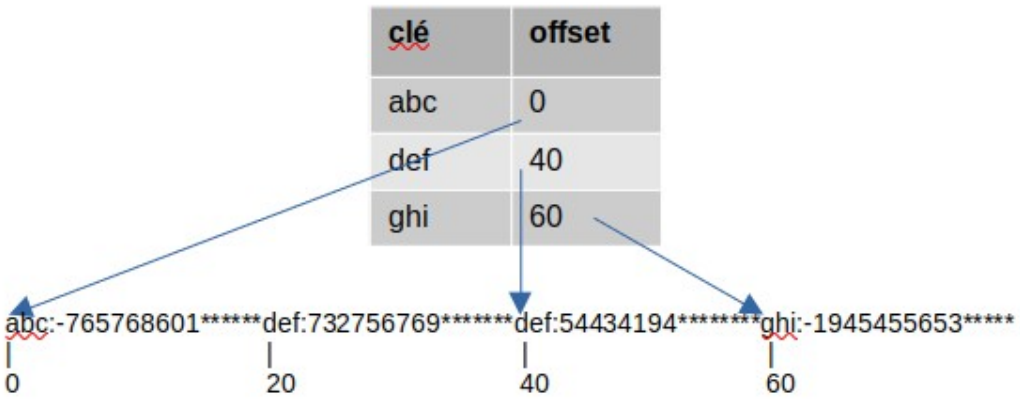

Figure 1: Indexation avant compactage

Dans le contexte l'exemple de la Figure 1, on considère que le prochain enregistrement aura la clé 'mno' et la valeur 'mno:681147641********. Cette valeur sera stockée à la position 80 (car chaque

enregistrement est de taille fixe 20 et que le précédent enregistrement était à la position 60) et sa clé sera insérée dans la table de hachage avec ‘mno’ pour la clé et 80 pour offset.

Après un certain temps, on se retrouve avec de gros fichiers qui contiennent des données rarement requêtées (car obsolètes); par exemple pour la clé ‘def’, la valeur def:732756769******* n’est pas la plus récente et il n’y a donc pas d’index permettant d’y accéder rapidement. Dans ce cas, cela fait sens de compacter les fichiers après un certain temps. L’impact du compactage est mis en évidence en le lançant sur notre exemple, nous obtenons alors une nouvelle version de l’index qui est présentée dans la Figure 2.

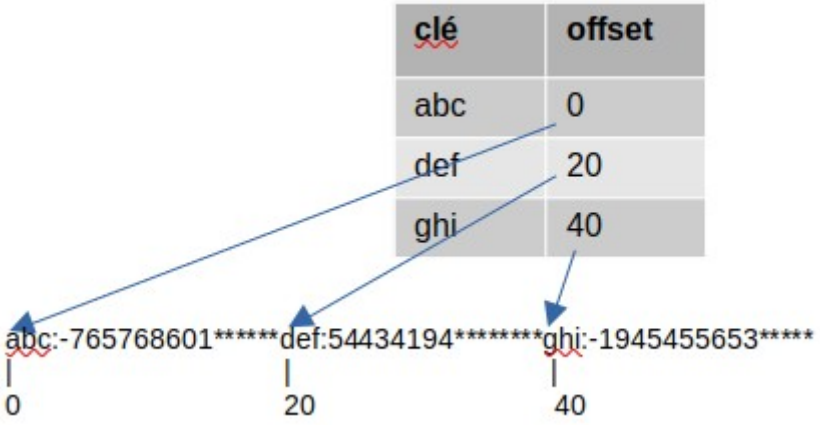

Figure 2: Indexation après compactage

Il ne reste qu’un seul enregistrement dans l’index avec la clé ‘def’ (def:54434194******** qui est le plus récent, et la table de hachage est à jour. Le nouvel index, résultat de ce compactage, est prêt à recevoir de nouveaux enregistrements.

Un fichier sur disque fichier précédent est archivé sur disque et il sera donc possible de rechercher des valeurs dans celui-ci, par exemple pour obtenir l’historique de l’évolution d’une clé dans le temps. Dans cet exercice, on ne vous demande pas de gérer le requêtage sur les fichiers archivés.

Travail à faire

Vous devez implémenter ce système de stockage en Scala en utilisant autant que possible des fonctions pures. Vous organisez votre code de la manière qui vous semble la plus pertinente mais en adoptant les grandes lignes d’un développement orienté programmation fonctionnelle.

Pour simplifier votre développement, on considère que la taille des enregistrements est fixe. Concernant la stratégie de compactage, on adoptera la solution basée sur le rapport entre le nombre d’enregistrements pertinents sur le nombre total d’enregistrements non-pertinents dans le fichier. Si ce rapport est inférieur à un seuil préfixé (par vous même et paramètrable) alors le système lancera le compactage. Par exemple, avec la suite des 8 enregistrements suivants: abc1, ghi1,abc2, def1,ghi2,def2,def3,abc3; le rapport pertinents/non-pertinents est de 3/8=0.375. Si votre seuil était à fixé 0.4, alors il faut lancer le compactage sinon votre programme peut continuer à accepter des enregistrements sans lancer le compactage.

Dans votre programme, vous définissez ce rapport à l’aide d’une constante. Dès que le rapport de votre stockage à un instant t est inférieur à ce rapport, vous devez lancer le compactage.

Dans un object Scala, vous spécifiez les paramètres de votre programme, à minima le seuil d'activation du compactage, la taille des enregistrements. Dans l'objectif de générer aléatoirement des jeux de données, vous pourrez également enregistrer les paramètres supportant cette création , par exemple le préfixe des fichiers à enregistrer à la suite d'un compactage, les valeurs des clés et le nombre d'enregistrements à générer lors d'un expérimentation.

Vous devez également permettre le requêtage du système de stockage depuis l'index. Dans la Figure 1, le requêtage sur la clé 'def' doit retourner la valeur def:54434194********, la valeur de la clé def la plus récente.

Vous rendrez votre projet sur le site de elearning (à la suite du TP de vendredi 4/10) en y incluant le code source et le résultat d'au moins une expérimentation. Celle-ci devra mettre en évidence quand le compactage s'est effectué, l'état de l'index avant et après compactage et l'état du fichier enregistré sur disque. Vous ajouterez également le résultat de requêtes 1) obtenant la valeur la plus récente d'une clé, 2) ne trouvant pas une clé dans l'index.