Rapport ABDR

BELLACICCO Denis
SIMONET Lucas

6 janvier 2014

1 Transaction et concurrence

1.1 Exercice 1

Dans le tme, il nous est demandé d'écrire un programme, exécutant en parallèle deux threads, qui accèdent en simultanné à un même profile. Lors de l'analyse des résultats obtenus après l'exécution de ce programme, on constate que l'on a d'importante perte d'écriture. En effet, le but de l'application étant que chaque thread incrémente la valeur d'un couple clé,valeur de un pendant mille itérations. On s'attend à ce que, lors de la terminaison de l'application, cette valeur soit de égale à sa valeur initiale plus deux milles. On utilise, pour ajouter et modifier les couples clés, valeurs, la fonction proposé par kvstore, KVStore.put() Or, la valeur dans la base de donné après l'exécution de l'application est inférieur à la valeur attendue. Ce problème est du à l'absence de contrôle lors des accès en concurrence de plusieurs applications sur la même base de données. Pour résoudre ce problème, il faut s'assurer que la version de la valeur que l'on veut modifier est la même que celle que l'on a lu.

1.2 Exercice 2

Dans l'exercice précédent, nous avons résolu le problème du à l'incohérence entre la lecture et l'écriture des données. Or lors de l'execution de deux programmes M1 simultanément ne concerve pas la cohérence des données. Lors d'une itération, les incréments effectués sur les produits sont faites de manière totalement indépendante. Du coup, deux programmes peuvent se chevaucher lors de l'incrément des produits et donner des résultats incohérents. Si les produits on, au départ, tous la même valeur, ils ne l'auront pas forcément à la fin de l'execution du programme. On décide de résoudre ce problème en incrémentant le max des valeurs de tout les produits et que ces derniers soit alors égale à cette nouvelle valeur. Cela permet d'être sur que les valeurs de tout les produits seront les même. On écrit donc un programme M2 en conséquent.

Lors de l'exécution de deux programme M2 en série, la quantité pour tous les produits vaut 201. Mais lors de l'exécution de deux programmes M2 en parallèle, on observe que le résultat obtenu n'est pas le même que précédemment. Lors d'une itération, certain produit peuvent avoir été mis à jour par un autre programme au même moment.

La solution pour résoudre ce problème, est de rajouter chaque opération à effectuer lors d'une itération dans une liste d'opération et de tout exécuter en une fois. KVStore permet cette solution, du coup toutes les opérations sont effectuées de façon "atomique". On retrouve une cohérence des données avec une execution en série.

2 Transaction et équilibrage de charge

Pour stocker les profils dans la base de données, nous avons choisi comme couple clé, valeur :
- clé majeure : Profil, Objet
- clé mineure : Attribut
- Valeur : valeur de l'attribut

2.1 Etat initial

Nous avons opté pour la première étape, d'une application muti-threader. Lors du lancement de notre application, un certain nombre de threads sont créés. Chaque thread est l'équivalent d'une application. L'application iter pendant 10 seconds. Lors de chaque iteration, elle execute un transaction en calculant son temps d'execution. Avant de se

terminer, l'applicaion effectue une moyenne du temps d'execution de toutes les transactions qui ont été executé. Une transaction crée un nombre d'objets définit pour un profil définit. C'est la transaction qui interagit avec kvstore.

2.1.1 Charge ciblant un seul store

Nous avons N applications (N compris entre 1 et 10) effectuants des insertions, chacune sur des profils différents, mais sur la même base de données KVDB. Les applications sont lancées de façon parallèle, mais comme elles effectuent leurs opérations sur des profils différents, il n'y a pas de problème de concurrence. Les temps observés, lors de l'augmentation des threads, évoluent de façon linéaire. On en déduit que le goulot d'étranglement se situe lors des écritures sur le disque.

Nombre de threads	Temsp d'execution moyen
1	43
2	43
3	74
4	126
5	200
6	230
7	260
8	309
9	363
10	400

2.1.2 Charge ciblant deux stores

Nous avons la même architecture que dans le cas du dessus, à l'exception que nous nous retrouvons avec deux stores avec la moitié des threads effectuant des opérations sur l'un des stores et l'autre moitié sur l'autre store. On observe que lorsque les dix threads sont lancés, nous obtenons les mêmes vitesses d'executions que dans le cas où cinq threads sont executés sur un seul store, ce qui parait logique car c'est l'equivalent du cas précédent avec deux machines.

Nombre de threads	Temsp d'execution moyen
1	45
2	46
3	73
4	132
5	198

2.2 Catalogue

A partir de cette partie, nous développons notre application de façon distribué. Chaque KVDB que nous aurons seront géré chacun par un serveur. Il y aura un serveur spécial, appelé "gateway", qui s'occupera de recevoir les requètes. Il connaitra la localisation de chacun des profils, ainsi que leur tailles, et la taille des KVDB. C'est lui qui décidera de la migration d'un profil, ainsi que de sa destination. Finalement, le client connaitra l'adresse du "gateway" et executera les opérations sur les profils à travers celui là. L'ensemble de l'application communique grâce à l'API RMI.

Les informations à connaitre pour migrer un profil d'un store à un autre sont, sa localisation courante. Nous n'avons pas besoin de plus d'information, en effet, comme c'est notre gateway qui s'occupe de migrer les profils, qui gère toutes les requètes adressées aux serveurs et comme il n'y a pas de multi-threading dans notre application, on peut être sur que lors d'une migration, le profil n'est pas en cours d'utilisation ou déjà en cours de migration.

2.3 Déplacement

Notre politique de déplacement est de regarder, après chaque requète, si la base de données sur laquelle la requète n'est pas au moins deux fois plus grosse qu'une autre. Si c'est le cas, alors on migre le profil sur lequel on a fait les opérations sur le nouveau serveur.