Rapport ABDR

BELLACICCO Denis
SIMONET Lucas

13 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. Nous avons decidé d'implementé une couche d'abstraction appelé "Gateway", qui s'occupera de recevoir les requètes et de retourner les resultats au client. Le Gateway possede connaissance de la localisation de chacun des profils, ainsi que leur tailles, et la taille des KVDB. Ainsi il aura la capacité de determiner sur quel Serveur un profile doit etre migré, lorsque celui ci est surchargé. Finalement, le client ne connaitra que 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.

Pour l'implementation de cette solution, nous avons donc eu besoin que le Gateway possede une base de donnée mise à jour regulièrement possedant :

- Une map "mapServeur" des serveurs en fonction des profils. Cette map permet de savoir avec quel serveur rmi nous devons discuter pour trouver le profil.
- Une map "mapProfile" du nombre d'objets en fonction d'un profil. Cette map permet de savoir le nombre d'objets actuellement present dans le profil.
- Une map "confServeur" avec les informations du KVStore en fonction du serveurRmi. Cette map permet de migrer les donnée d'un KVStore à un autre sans avoir à repasser par le Gateway.
- Une map "serveurSize" du nombre d'objets en fonction du serveur. Cette map permet de verifier la charge d'un serveur.

Nous avont par ailleur créé la methode "IServeur needsMigration(IServeur serv,String profil)" qui observe la charge de tous les serveurs et la compare à la charge du serveur qui possede actuellement le profil. Si jamais il trouve un serveur qui à 2fois moins d'objets que lui, et que l'ajout de ce nouveau profil ne risque pas de simplement inverser la charge. Alors il retourne ce Serveur qui servira de destination pour la 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.

Plus précisement :

La methode "IServeur needsMigration(IServeur serv,String profil)" du Gateway est appelé à chaque fois que nous appelons la fonction "int comit(int profile,boolean migrate)" avec la migration activée. Si le retour n'est pas nul, alors il existe un serveur sur lequel il peut etre interessant de migrer les données. La fonction appele alors "int migrate(String profile, IServeur serveurDest)" qui lance la migration a partir du serveur. Une fois la migration effectué alors le Gateway envoi au serveur une demande de suppression de la donnée pour eviter les doublons. Il met ensuite à jour les differentes valeurs stoqué dans les map. Du coté serveur, lorsque la fonction "migration(String profile,String[] kvstore, int lastObjetId)" est appelé alors le serveur creer une transaction qui aura pour but de recuperer toutes les données du profil sur le KVStore et de les copier sur un autre KVStore. La decision de suppression des ancienes données est faite par le Gateway après avoir vérifié que l'opération s'est bien déroulée.

L'avantage de cette solution est que du coté client, la totalité de ces interations sont invisibles. Le client ne possède que la connaissance des numéros de profils. La localisation de la donnée et la prise de decision d'une migration est faite par le Gateway. C'est egalement lui qui s'occupera de gerer toutes les requetes adressées aux serveurs.

L'interface client épuré ne possede donc que les actions relatives à ajouter un profil, supprimer un profil ou afficher un profil.

2.4 Multiclés

Pour les transaction sur plusieurs profils venants de differents KVStore, nous avons décidé d'observer la charge de tous les Stores impliqué et de migrer toutes les données vers ce dernier, pour ensuite executer toutes les transactions sur le meme KVStore.

Plus précisement :

A partir du Gateway, la methode "int comitMultiCle(int[] profiles)" va observer la charge des differents serveurs grace à la map "mapServeur" en prenant le premier serveur en reference. Une fois ce serveur trouvé elle va alors declancher la migration de tous les profils vers son KVStore. Puis finalement elle va exectuer les transactions sur chacun de ces profils. Ne monopolisant alors qu'un seul KVStore pour cette liste d'opperations.

L'avantage de cette solution est que du coté serveur, la transaction multiclé n'est percu que comme une suite d'actions atomiques ; "migrate" puis "commit". Deplus du coté client, la transaction multiclé n'est vu que comme une liste d'opperations. L'abstraction créée par le Gateway permet de n'avoir que cette classe à modifier pour pouvoir creer des fonctionnalités complexes.

3 Consistency Tradeoffs in modern distributed database system design

3.1 Fiche de lecture

L'industries ayant mal compris le théorème CAP, a implémenté des systèmes de bases de données limités par rapport à leur potentiel. Cependant, les bases de données réparties ont été développés avec l'idée que le système ne peut fonctionner sans un compromis entre cohérence et latence.

3.1.1 CAP is for failure

CAP déclare que les concepteurs des BDR peuvent choisir parmis deux de trois propriétés qui sont la cohérence, la latence et la tolérance aux fautes. Les concepteurs ont fait la suposition que étant donné que les systèmes de BDR doivent être tolérents aux fautes, il faut choisir entre disponibilité et consistence. Or CAP spécifie que le compromis entre les propriétés de cohérence et de latence ne doit être mis en place qu'en cas de fautes du réseau. Or la probabilité d'avoir des fautes du réseau dépendent de l'implementation du système (système réparti sur un WAN). La tolérence aux fautes du réseau étant rare, le théorème ne justifie pas complètement la conception par défaut des systèmes de BDR.

3.1.2 Consistency/latency tradeoff

Les systèmes de bases de données modernes ont été développé pour intéragir avec des pages web créées dynamiquement et relié à un utilisateur du site. La latence étant un facteur critique pour concerver des consommateurs. Or il y a un compromis fondamental entre consistence, latence et disponibilité causé par le fait que pour avoir une haute disponibilité, le système doit répliqué des données via un WAN pour se protéger des échecs d'un datacenter entier.

3.1.3 Data replication

Trois stratégies de réplication des données possible :

Données envoyé à tout les serveurs de réplication L'ordre des mises à jour peut être décidée par chaque machine, ce qui cause des problèmes de consistence. L'ordre des mises à jour est commune à toutes les machines via une couche de prétraitement, ce qui cause des temps de latence dû au traitement et, si la couche de prétraitement est répartie sur plusieurs machine, au protocol d'accord qui permet l'ordre des opérations, ou, si la couche de prétraitement se situe sur une seule machine, à la distance que les données doivent parcourir.

Données envoyées à une machine maitre Il existe trois options de réplications :

- Réplication synchrone : augmente le temps de latence
- Réplication asynchrone, provoque deux type de compromis possibles lors des lectures :
  1. Toutes les lectures sont redirigées vers le neud maitre, ce qui provoque des temps de latence à cause de la distance et si le noeud maitre est surchargé ou s'il est tombé.
  2. Tout les noeuds peuvent effectuer des lectures mais avec des risques d'incohérences si toutes les mises à jour n'ont pas fini d'être propagées.
- Réplication synchrone sur certains noeuds, asynchrone sur le reste. On subit de nouveau deux compromis entre cohérence et latence dû aux lectures :

1. La lecture est routée vers un noeud synchrone, la cohérence est bonne mais on retrouve les différents types de latence vu précédemment.
2. Des lectures peuvent être effectuées par tout les noeuds. Des problèmes de cohérence peuvent apparaitre.

Données envoyées à une machine aléatoire Les données sont envoyées à un noeud puis répliquées à tout les autres noeuds. Deux options pour choisir entre cohérence et latence :
- Réplication synchrone sur tout les noeuds permet d'être cohérent mais produit d'importe latence.
- Réplication asynchrone permet une bonne latence mais des risques d'incohérences.

3.1.4 Tradeoff examples

Il n'y a pas de stratégie concervant une faible latence avec une forte cohérence lors de réplication via un WAN. Les systèmes de bases de données développés dans l'industrie utilisent toutes des stratégies avec une faible latence, sacrifiant ainsi la cohérence des données.

3.2 Avis personnel

Cet article nous a permis d'apprendre beaucoup de chose sur le fonctionnement des systèmes de base de données réparties ainsi que les différents compromis auquels ils sont soumis dû à la réplication des données via un WAN.