INFO-F201 SmallDB : rapport

MARKOWITCH Romain HENNEBICQ Jérémy MARNETTE Pol

16 décembre 2022

Table des matières

1 Introduction

Ce projet est la prolongation de la première partie "tinydb" en prenant une nouvelle approche. Le but est le même : la réalisation d'une base de données en C++. Ici le serveur est différencié du client et communiquent ensemble via des sockets. Aussi, les requêtes des différents clients seront maitenant gérés par des threads individuels. Cette nouvelle approche soulève des questions concernant l'implémentation. Il faudra faire attention au niveau de l'accès concurrent à la base de données, trouver un moyen de passer les réponses au client, etc.

Dans ce rapport nous allons décrire notre implémentation, revenir sur certains points qui nous ont posés problèmes et comparer cette nouvelle implémentation à la précédente.

2 Implémentation serveur client

2.1 Implémentation des mutex

Pour éviter les accès concurrents à la base de données, nous avons du implémenter des mutex. Nous avons utilisé le pseudo-code fournit dans l'énoncé pour créer 4 fonctions qui sont appelés avant et après les lectures et les écritures. L'implémentations de celles-ci se trouve dans le fichier `guard.cpp`. Voici l'explication de son fonctionnement :

1. Nous créons une mutex `new_access` qui sera utilisé pour bloquer le processus de registration du lecteur ou de l'écrivain. Elle sera bloqué au début de chaque fonction `_before` et unlock à la fin,
2. Nous créons une mutex `write_access` qui permet de restreindre l'accès à un unique écrivain à la fois. La fonction `write_guard_before` la verrouillera et la fonction `write_guard_after` la déverrouillera,
3. Nous créons une mutex `reader_registration` qui restreindra l'accès à la variable `readers_count` pour éviter que deux threads la modifies en même temps,
4. Ensuite, dans la fonction `read_guard_before` on regarde s'il n'y a pas encore de lecteur, si c'est le cas, nous bloquerons l'écriture et incrémenterons le nombre de lecteur,
5. Finalement dans la fonction `read_guard_after` on décrémentera le nombre de lecteur et si celui-ci est maintenant à 0, nous déverrouillerons l'accès à l'écriture.

Cette approche peut sembler à première approche complexe. Mais celle-ci évite les soucis de famine.

2.2 Difficultés rencontrés

2.2.1 Communication des résultats au client

Pour communiquer les résultats d'une requête envoyé par le client au serveur nous avons d'abord essayé de transmettre au client un structure `query_result_t`. Cette structure contins un `std::vector` des étudiants sélectionnés, la requête, le type de requête (SELECT, etc), l'état de la requête et éventuellement un message d'erreur. Nous avons donc essayé d'envoyer cette structure au client pour qu'il puisse lire les résultats de sa requête. Voici le bout de code que nous avons essayé :

```
write(socket, query_result, sizeof(query_result_t));
```

Nous n'avons pas réussi à faire fonctionner la communication avec cette approche. Nous en avons déduit que communiquer un `std::vector` via les sockets n'était pas trivial. Nous avons aussi essayé avec une liste dans le style de C (avec des pointeurs) mais le soucis restait.

Nous avons donc opter pour une autre solution : envoyer ligne par ligne les résultats à imprimer chez le client. Nous avons défini un marqueur d'arrête (`**` en l'occurence) qui indiquera au client la fin des réponses a afficher dans le terminal. Cette solution fonctionne mais nous laisse perplexe au niveau de la charge réseaux, une autre implémentation nécessitant moins de `read` et `write` pourrait être optimal. Voici un bout de code de notre implémentation :

```
// Server
for (auto student: query_result.students) {
        student_to_str(buffer_response, &student, 1024);
        write(socket, buffer_response, 1024);
}
snprintf(buffer_response, 1024, "%s", RESULT_EN_MARKER.c_str());
write(socket, buffer_response, 1024);

// Client
while ((bytes_read = read(sock, response_buffer, 1024)) > 0
        && strcmp(response_buffer, RESULT_EN_MARKER.c_str()) != 0) {
        std::cout << response_buffer << std::endl;
}
```

2.2.2 Nombre d'étudiants supprimés

Quand un client exécute la requête `delete filter=value`, le serveur supprime les éléments et renvoie au client `n student(s) deleted`. Le code fournit pour ce projet contenait déjà la logique de la fonction delete mais pas un moyen de récupérer facilement le nombre d'étudiant supprimé. Cette fonction utilise `remove_if` qui retourne un itérateur sur le vecteur. Pour compter les éléments nous avons itéré sur les éléments supprimés pour les compter. Voici un bout de code de l'implémentation :

```
auto new_end = remove_if(db->data.begin(), db->data.end(), predicate);

for (auto s = new_end; s != db->data.end(); ++s) {
        query_result.students.push_back(*s);
}
```

3 Comparaison avec la première partie