

# Rapport d'OS

Andrius Ezerskis & Moïra Vanderslagmolen & Hasan Yildirim

November 3, 2022

# 1 Introduction

Il y a peu de temps, notre université a décidé qu'il était temps de changer notre base de données des étudiants afin d'améliorer l'accessibilité et le confort du système, ce qui permettra de fournir plus d'informations et une meilleure entrée à nos camarades étudiants. La base de données précédente ne répondait plus aux besoins depuis que le nombre d'étudiants a considérablement augmenté ces dernières années. Il était donc nécessaire que notre université s'occupe de ce problème de croissance de la base de données. Nous avons demandé ce projet pour gérer la création d'une nouvelle base de données. Notre groupe est composé de 3 étudiants qui ont joué des rôles différents dans ce projet mais en coopération. L'ancienne base de données donnait beaucoup d'échecs. Les répétitions de noms, les informations manquantes sur les étudiants, ou les comptes supprimés étaient les quelques exemples parmi tant d'autres survenus pendant l'utilisation active de la base de données. Notre projet a été divisé en 8 parties différentes. Les parties suivantes sont les parties de notre projet.

## 2 Stockage de la base de données

Afin de stocker les données de milliers d'étudiants, plusieurs choix se sont offerts à nous. Tout d'abord, nous avons utilisé `std::map`, comparable à un dictionnaire en python, dont les clés ont été l'identifiant de l'élève et les valeurs la struct `student`. Seulement, cela prenait beaucoup de place en mémoire et nous avons donc délaissé l'idée. Nous avons donc décidé de faire un tableau.

## 3 Checker l'identifiant

Pour voir si l'identifiant est déjà présent dans la base de données, nous avons pensé à mettre chaque étudiant dans son emplacement "correspondant" dans le tableau (étudiant `id=500` va à l'emplacement `db[500]`). La recherche par `id` et le `delete` auraient été rapides, et on aurait pu rapidement savoir si un étudiant est déjà présent dans le tableau ou non. Seulement, si l'utilisateur introduit un `id` très grand, le stockage de la data base sera très grand, pour un seul étudiant. Nous avons donc essayé d'implémenter une boucle `for`, qui vérifiait si l'`id` de l'étudiant n'était pas encore présent en itérant dans toute la base de données. Cela prenait beaucoup trop de temps. Nous avons donc décidé d'avoir un tableau trié d'étudiants. Pour cela, lorsque nous ajoutons un étudiant, nous vérifions si l'étudiant à l'emplacement du tableau -1 est plus petit que lui. Si il est plus petit que lui, nous l'ajoutons juste derrière. Sinon, nous décrétons la variable au fur et à mesure, jusqu'à ce que l'étudiant aie un `id` + petit que lui. Lorsqu'il trouve cet étudiant, il s'insère juste derrière lui et décale toute la database afin de s'insérer en plein milieu.

## 4 Fin d'une query

Pour signaler la fin d'une query, nous avons implémenté une message queue grâce à une pipe. Lorsqu'on envoie une query à un processus, nous incrémentons une variable globale `operation_i` *progress qui indique le nombre d'opérations bloquant, ce qui permet au processus principal de continuer même si il n'est pas dans la pipe. Lors d'une transaction, on limite les opérations*

## 5 Gestion de signaux

Plusieurs choix se sont offerts à nous afin de terminer un programme. Nous avons d'abord pensé à 4 pipes qui indiquent au programme principal la fin d'une requête, mais ce n'était pas optimal. Nous avons repris la message queue qui traite les query, et dans cette message queue, le processus principal écrit un message, "kill". Les processus enfants le liront après avoir terminé leur requêtes, et le kill arrêtera la boucle et exit le programme.

## 6 Classes

Nous avons mis la `database_t` et le `query_result_t` dans une classe, afin de rendre le code plus lisible. Par conséquent, nous avons implémenté un constructeur dans `query_result_t`, au lieu du `query_result_init`, pour que le code crée directement le `query_result_t`, sans devoir appeler l'initialisation à chaque création d'objet. Dans `query_result_t`, nous avons aussi ajouté `log_query`, pour pouvoir mettre les attributs en privés. Nous n'avons pas mis `student_t` en classe, parce que les attributs d'un student sont souvent accédés par la classe `query_result_t`. et nous n'aurions donc pas su mettre les attributs en privés.

## 7 Améliorations possibles

### 7.1 `std::map`

Nous avons comme idée de stocker, dans une structure de données `std::map` (équivalent d'un dictionnaire en Python), les identifiants des étudiants en clés ainsi que leur indice de position dans la base de données en valeurs de ces clés.

Selon nous, cela aurait permis de faciliter et d'améliorer l'efficacité de la vérification de si un étudiant est déjà dans la base de données ou non.

En effet, lors de l'ajout d'un étudiant, nous aurions juste à vérifier si l'identifiant est présent dans le `map` pour éviter d'avoir des doublons dans la base de données.

### 7.2 Threads

Au lieu d'utiliser des processus, nous aurions pu utiliser des threads. Seulement, si le processus main crashe, alors tous les threads crashent aussi. L'avantage des processus est leur indépendance, ce qui les rend plus stable que des threads.