Relatório da Fase 1 - Grupo 03

Francisco Macedo Ferreira (A100660)
Hugo António Gomes Ramos (A100644)
Júlio José Medeiros Pereira Pinto (A100742)

22 de novembro de 2022

1 Introdução

Este relatório tem como intuito abordar a primeira fase do projeto da UC Laboratórios de Informática III do ano letivo 2022/2023. Nesta fase como objetivos o trabalho necessitava a implementação de *parsing* dos dados de entrada, funcionar através do modo de operação *batch*, realizar pelo menos 3 das 9 *queries* considerando que o tamanho dos ficheiros de entrada (*users.csv, drivers.csv* e *rides.csv*), terá 100.000, 10.000 e 1.000.000 linhas, respetivamente. Ainda mais, no projeto, em geral, existe uma necessidade do uso de conceitos de encapsulamento e modularidade.

Este relatório irá abranger as decisões tomadas pelo grupo tal como o método de raciocínio para desenvolvimento do mesmo.

2 Desenvolvimento

2.1 Pipeline atual

Neste momento a nossa pipeline funciona da seguinte maneira:

- Abre-se os ficheiros do dataset passados como parâmetro do programa (*drivers.csv, users.csv, rides.csv*).
- Cada linha desses ficheiros é passada para um parser genérico com informações de como interpretar essa linha e o que fazer com a estrutura *parsed*.
  - A informação de como interpretar a linha é um pointer para a função de *parse_line_user*, *parse_line_driver* ou *parse_line_driver*, uma em cada ficheiro de cada estrutura.
  - A informação de o que fazer com a estrutura *parsed* é um pointer para a função de *register_user*, *register_driver* ou *register_ride* no *catalog.c*.
- Ao finalizar a leitura, *parsing* e indexamento de todas as linhas no catálogo, este irá ordenar as *arrays* dentro do catálogo da maneira mais útil para a rápida execução das queries.
- Depois do catálogo pronto, lemos o ficheiro de input. Cada linha, em conjunto com o ficheiro de output correspondente criado, é passado para o *query_manager.c*, um controlador de queries, que irá verificar qual a query que está a ser pedida e chamar a função correspondente, separando a linha por espaços para a fácil interpretação dos parâmetros.
- Por fim, os ficheiros são fechados, a memória é libertada e o programa termina.

2.2 Estratégias Seguidas nas Queries

Para a primeira fase do trabalho decidimos implementar as bases do projeto, como o parser e definir as estruturas dos dados que utilizaremos ao longo do projeto (Como o catálogo para guardar os dados e os *users, drivers* e *rides*). Foram implementadas as *queries* 1, 2, 3, 4 e 5.

2.2.1 Query 1

Para acesso rápido a perfis pelo *id's* de condutores ou *username's* de utilizadores foram utilizadas duas *HashTables* (uma para indexar os perfis de condutores e outra para perfis de utilizadores) com *key id e username* e *value Driver* e User** respetivamente. Esses *HashTables* são populados conforme a leitura dos ficheiros .csv e informações como número de viagens, soma total de avaliações (para poder ser calculado a média), total gasto/auferido eram calculados e guardados na estrutura de dados dos utilizadores e dos condutores[1]. A decisão de buscar informações de utilizadores ou condutores é feita pela confirmação se o primeiro argumento da query é um número ou não: se for um número busca-se por condutores (*id*), se não busca-se por utilizadores (*username*).

Com isto é possível um acesso em tempo constante a essas informações, com o custo da leitura dos ficheiros ser um pouco maior (devido ao cálculo necessário de *hashes* para cada *key* e não só). Como na segunda fase o número de perfis irá aumentar exponencialmente e é esperado múltiplos acessos a estas informações, seja de várias queries destas ou do modo interativo ainda a implementar, o custo de leitura dos ficheiros superior é muito justificado.

2.2.2 Query 2

Na query 2, para rápido acesso aos condutores com maior média foi feito *sorting* da *array* (presente no catálogo) de condutores, conforme a sua média, no fim da leitura dos ficheiros. Isto poderá ser otimizado a fazer com que o *sorting* seja *lazy* (será abordado este tópico mais tarde). Durante a execução da query basta obter os N primeiros condutores da *array* e temos a execução em praticamente em tempo constante (agora é copiado os N elementos para uma array, mas poderá ser otimizado).

2.2.3 Query 3

Para a query 3, tal como na query 2, foi feito *sorting* da *array* (presente no catálogo) de utilizadores no fim da leitura dos ficheiros, de maneira a otimizar o acesso aos users com maior distância total percorrida. Por isso, durante a execução desta query o tempo de execução será praticamente constante (também tendo em conta a cópia dos N elementos para uma array). Tal como na query 2, o tópico de *lazy sorting* poderá ser aplicado da mesma maneira.

2.2.4 Query 4

Nesta query, o preço médio das viagens numa determinada cidade é calculada (em tempo linear conforme o número de viagens por cidade) durante a execução da query. Durante a leitura das viagens, é inserido a viagem conforme a sua cidade numa *HashTable (key: cidade, value: Array de Ride*).* O preço médio não é pré-calculado, pois o cálculo desta é relativamente rápido devido às viagens já estarem separadas por cidade e é expectável que só se aceda a este valor uma vez, por isso guardá-lo será desnecessário. Esta *HashTable* já existe devido à query 7 que necessitará de acesso rápido a viagens conforme a sua cidade.

[1] Pode ser uma possível quebra de encapsulamento. Ainda aguardamos a resposta do docente sobre esse quesito.

Futuramente, conforme a expansão do dataset, esta implementação poderá ter de ser reformulada, por causa do cálculo em execução mas para já tivemos bom desempenho com a atual.

2.2.5 Query 5

Já na query 5, no fim da leitura das viagens é feito *sorting* das viagens pela sua data, por ordem crescente. Com isto, assumindo que <data A> e <data B> são os argumentos da query, basta aceder ao primeiro elemento a partir do qual <data A> é menor ou igual do que a data desse elemento. A partir daí, podemos percorrer a lista até encontrar uma viagem que a sua data seja maior que <data B>, acumulando o preço das viagens para no fim calcular a sua média. O tal primeiro elemento é encontrado com uma implementação semelhante à **std::lower_bound** de *C++*, usando *binary search*. Isto é possível devido à lista estar organizada pela data das viagens.

Para evitar percorrer a lista, podia ter sido pré-calculado uma *array* em que cada índice tinha o somatório de preços para trás desse índice e o preço médio era calculado subtraindo o *upper_bound* com o *lower_bound* do *range* das datas e dividindo pelo número de elementos entre eles. Essa ideia foi rapidamente descartada devido às datas dos argumentos serem relativamente perto, portanto são poucas as viagens a iterar. Esta solução também iria aumentar consideravelmente o tempo de leitura dos ficheiros.

2.3 Análise de desempenho

Comparando o desempenho da execução das queries 1, 2, 3, 4 e 5 para os ficheiros de input do conjunto de testes expandido das pastas *tests_1* e *tests_2* (excluindo obviamente as queries não implementadas) temos na Tabela 2 os resultados conforme as especificações dos computadores na Tabela 1. Ainda não existe um *standard* para a medição de desempenho dos programas, por isso, medimos da seguinte forma:

- O programa foi compilado com as flags *-O3 -flto -funroll-loops -march=native*.
- **Loading time** é o tempo de leitura, *parse* e indexamento dos ficheiros de input (incluindo o tempo de *sorting* no fim). Para já o loading é independente das queries executadas.
- **Query time** é o tempo de execução de todas as queries no ficheiro, incluindo a escrita do output nos ficheiros.
- Não são considerados tempos de *free* de memória (no fim da execução do programa) em nenhum dos tempos.
- O resultado é uma média de 3 execuções, após uma primeira execução de aquecimento.
- O tempo foi medido no código com o utilitário *GTimer* do *GLib*.

	PC 1	PC 2	PC 3
CPU	M1 Pro 8-core (6 perf. e 2 ef.)	Intel i7-8550U 4-core	Intel i7-1165G7 4-core
RAM	16GB LPDDR5	8GB DDR4 2400MHz	16GB DDR4 3200MHz
Disco	500GB NVME	500GB NVME	1TB SSD
OS	MacOS Ventura 13.0.1	ArcoLinux Kernel 6.0.9	Windows 11 WSL 1.0
Compilador	Clang 15.0.5 (ARM64)	GCC 12.2.0	GCC 9.2.0 (Ubuntu 20.04)

Tabela 1: Especificações dos PCs

	PC 1	PC 2	PC 3
Loading de ficheiros	767.1ms	1466.1ms	1427.4ms
Execução de 13 queries (tests_1.txt)	8.0ms	23.3ms	32.7ms
Execução de 29 queries (tests_2.txt)	14.1ms	45.2ms	66.5 ms

Tabela 2: Tempos de execução em diferentes PCs

Sobre o uso de memória, o programa usa, em pico, próximo de um total de 168 MB.

2.4 Possíveis otimizações e futuras limitações

Como podemos ver na Tabela 2, o tempo de loading dos ficheiros é muito grande comparado com o tempo de execução das queries. Após alguns simples testes, chegamos à conclusão que uma parte significativa do tempo de loading é do *parsing* das linhas das estruturas de dados e outra parte é da inserção, *sorting* e indexamento das viagens, condutores e utilizadores nas estruturas adequadas. Para melhorar este tempo, podemos implementar multi-threading na leitura dos ficheiros de condutores e utilizadores, pois não há dependências entre eles, e talvez rever a forma como as estruturas estão a ser *parsed*. Como o número de viagens, condutores e utilizadores ainda

vai aumentar exponencialmente na segunda fase do projeto, será realmente necessário explorar possíveis melhorias nesta parte.

A parte positiva é que parece que as escolhas de implementações das *queries* foram excelentes, pois o tempo de execução é super-rápido. Ainda assim, podemos melhorar a performance das queries com, por exemplo, as otimizações de *lazy sorting* referidas nas explicações das implementações das *queries*. Este *lazy sorting* pode também aliviar o tempo de loading dos ficheiros, já que este só será feito quando necessário, ou seja, quando uma *query* for executada.

Sobre o uso de memória, também estamos confiantes para a segunda fase já que para o dataset atual de 93.8 MB, apesar do nosso programa usar 1.8 vezes mais (168 MB), ainda conseguimos fazer várias otimizações nas estruturas de dados para a reduzir. Se esse rácio se mantiver, ainda podemos lidar com um dataset de 1 GB sem atingir o limite estipulado de 2 GB de uso de memória.

3 Conclusão

Para concluir, acreditamos que o desenvolvimento desta primeira fase foi muito satisfatória. Consolidamos os nossos conhecimentos de C com conhecimentos mais avançados relacionados a performance, gestão de memória, encapsulamento e modularidade.
Tópicos como o uso de *getters com clone* e o *encapsulamento de estruturas* fizeram um pouco de confusão a serem aplicados numa linguagem de tão baixo nível sem suporte a objetos: o código ficou mais verboso e menos performático do que se não houvesse essa "restrição".
Apesar de tudo, ainda existe espaço para melhorias nestes conceitos e pretendemos desenvolvê-los a um nível ainda mais avançado do que os apresentados na próxima fase do trabalho.