Projeto Laboratórios de Informática III

Fase 2

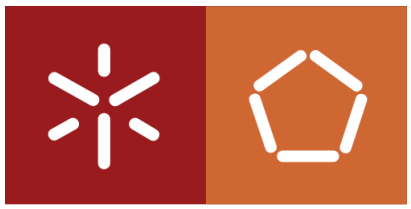
Projeto desenvolvido por

Daniel Pereira (A100545), Rui Lopes (A100643) e

Duarte Ribeiro (A100764)

Grupo 69

Licenciatura em Engenharia Informática



Universidade do Minho
Escola de Engenharia

Departamento de Informática

Universidade do Minho

Conteúdo

Capítulo 1

Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Laboratórios de Informática III do ano 2022/2023, unidade esta que pretende dar a conhecer aos alunos alguns dos princípios fundamentais da Engenharia de Software - *design* de arquiteturas, modularidade e encapsulamento de código. Foi-nos proposta a implementação de uma base de dados em memória que armazene dados fornecidos pelos docentes.

Após a conclusão da primeira fase, que consistia em implementar o *parsing* dos dados e o modo *batch*, completamos todos os outros requisitos do programa, nomeadamente tornar o programa capaz de realizar todas as nove *queries*, testes funcionais e de performance, verificação do conjunto de dados fornecido e criação de um modo interativo. O modo interativo consiste numa interface gráfica no terminal que permite fornecer ao programa a diretoria dos ficheiros com os dados e realizar *queries* introduzidas pelo utilizador.

As estruturas e algoritmos implementados anteriormente também seriam postos ao teste, visto que nesta fase foi introduzido um *dataset* novo, com dimensão dez vezes superior. Assim sendo, quaisquer ineficiências ou ideias mal-concebidas iriam ser ainda mais penalizadoras. Apesar de não haver critério específico para o tempo máximo de execução do programa, definimos uma meta pessoal de conseguir executar as 500 *queries* testadas pela equipa docente em menos de 45 segundos, medidos pelo site disponibilizado. Criámos este desafio de modo a refletirmos as nossas escolhas anteriores, pois fizemos algumas decisões que, em retrospetiva, não foram ideais, e porque ficamos interessados em testar os limites da nossa implementação.

Capítulo 2

Arquitetura e Estruturas

Recapitulando a estrutura base da primeira fase, optamos por armazenar os dados lidos pelo *parser* em três *hash tables* diferentes (uma para cada ficheiro de dados), que eram acessadas pelas *queries* de modo a obter as informações relevantes às mesmas. Esses resultados eram então colocados num ficheiro de texto.

Após a apresentação da primeira fase à equipa docente, compreendemos que a nossa visão e estrutura foi bem recebida, logo mantivemos a mesma filosofia e arquitetura, apenas adicionando e alterando componentes da mesma de modo a acomodar novas funcionalidades.

Tendo a segunda fase oficialmente começado, focamo-nos inicialmente em ter todas as *queries* funcionais e a completar os testes automáticos. Para tal, desenvolvemos várias estruturas auxiliares onde eram guardadas estatísticas relevantes às *queries* durante o *parsing*. Deste modo, quando chegasse à altura de executar a query, o programa teria uma *workload* menor e poderia devolver o resultado mais rapidamente. Claro que como todas as soluções, esta possui um *tradeoff*. Apesar da execução das *queries* ser muito mais célere, o programa tem um tempo de *startup* muito maior, pois tem de inserir os dados necessários nas estruturas. No entanto, como esta abordagem nos traz muitos benefícios para o modo interativo e para um grande número de *queries* em modo *batch*, consideramos esta a melhor opção.

Implementamos um total de dez estruturas auxiliares, seis das quais guardam informações já existentes nas *hash tables* originais de outras formas, quer agregando, agrupando ou reordenando os dados disponíveis de modo a simplificar o processamento de uma *query*.

As restantes quatro estrututras foram utilizadas para codificar e descodificar os dados já existentes nos ficheiros de dados em tipos de dados mais eficientes. Apenas decidimos codificar as cidades e os *usernames* dos utilizadores, por duas razões particulares. Primeiro, ambas eram *strings*, um tipo de dados que dificulta a sua replicação e comparação, visto que os caracteres têm de ser duplicados/comparados um a um pelo comprimento inteiro da *string*, ao contrário de inteiros por exemplo, e porque ambas estavam presentes em todas as *rides*, o ficheiro mais numeroso em entradas, e, consequentemente, com maior peso de processamento/armazenamento.

A estratégia de codificação foi praticamente igual nos dois casos. A cidade/*username* é lida(o), é lhe atribuída(o) um número baseado na quantidade de elementos já lidos anteriormente e o par *string(key)*/código*(value)* é introduzido numa *hash table* de modo a conseguirmos encontrar o valor correspondente à *string* facilmente. A *string* é também armazenada num *array* na posição correspondente ao *código* para ser possível uma espécie de *reverse lookup*.

Decidimos colocar estas estruturas junto do catálogo de cada uma das entidades que necessitavam delas[1]. Pois sentimos que todas as estruturas necessárias para reconstruir o *dataset* original deveriam estar juntas, e apenas as *hash tables* neste caso não eram suficientes, visto que não sabem realmente qual é a cidade ou *user*, apenas têm uma representação dos mesmos por via de um inteiro.

[1]É de notar que nesta fase realizamos a separação do catálogo geral em três catálogos, um para cada entidade.

Tendo detalhado as nossas estruturas, apresentamos assim o diagrama da arquitetura atual:

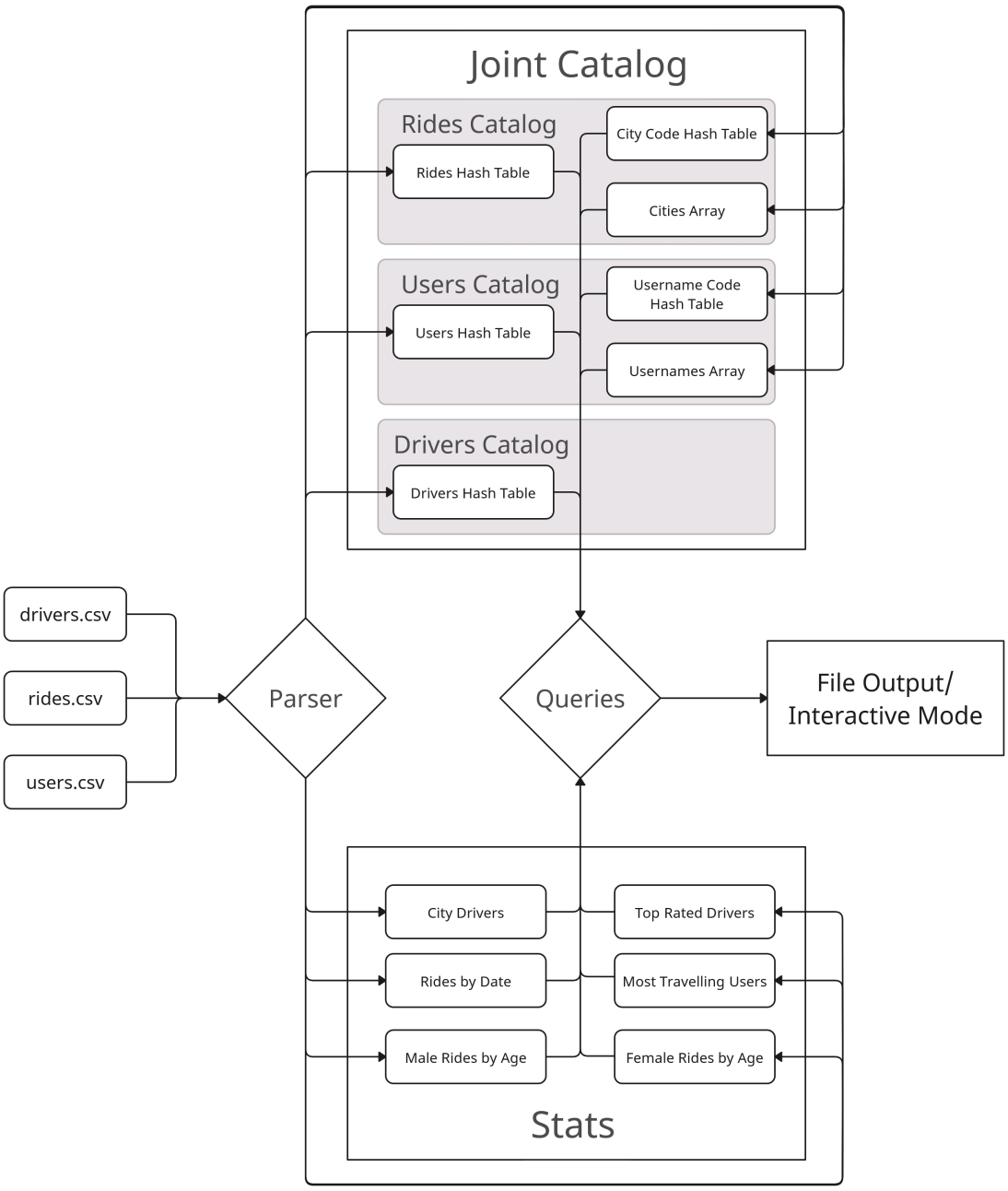


Figura 2.1: Diagrama da arquitetura na segunda fase

Capítulo 3

Queries

Na primeira fase, tínhamos decidido focarmo-nos nas *queries* 1,2 e 3, visto que foi as que consideramos mais simples de implementar. A única mudança que realizamos a estas *queries* foi alterar a estrutura temporária de ordenação dos utilizadores na segunda e terceira *queries* de uma *linked list* para um *array*, devido à maior eficiência de memória e rapidez. Nesta fase implementamos as restantes queries, ordenadas nesta secção pela estrutura auxiliar que utilizam:

City Drivers

Query 4: É nos fornecida a *string* da cidade e precisamos calcular o *"preço médio das viagens (sem considerar gorjetas) numa determinada cidade"*. Esta query é processada com ajuda da estrutura de estatísticas *City Drivers*, que consiste num *array* onde cada posição corresponde às estatísticas de cada cidade (segundo o código da mesma). Nessas estatísticas estão guardadas várias informações, mas as únicas relevantes para esta *query* são o valor total gasto e o número de rides efetuadas nessa cidade. A divisão do primeiro com o segundo é então devolvida como a resposta.

Query 7: É pedido para determinarmos o *"top n condutores numa determinada cidade, ordenado pela avaliação média do condutor"*. Nesta *query* voltamos a utilizar a estrutura de estatísticas anterior, mas utilizamos o array presente nessa estrutura, que armazena todos os condutores que realizaram viagens nessa cidade, juntamente com a avaliação total e o número de viagens realizadas pelo mesmo. Ao processar a *query*, esses valores são ordenados pela avaliação média e de seguida devolvidos.

Rides by Date

Query 5: Calcular o *"preço médio das viagens (sem considerar gorjetas) num dado intervalo de tempo"*. Esta *query* faz uso da *hash table Rides by Date*. Esta estrutura possui como chave uma data, e como valor um *pointer* para uma *struct*, denominada Rides of The Day. Dentro dessa *struct* podemos encontrar três valores, um que armazena o preço total de todas as viagens efetuadas nesse dia, um que armazena o número de viagens efetuadas nesse dia, e um *array*, onde cada um dos elementos do mesmo representa todas as *rides* efetuadas numa determinada cidade nesse dia. Todas as viagens efetuadas num determinado dia na cidade codificada com o número 2 estariam na terceira posição do *array*, por exemplo. Abaixo apresentamos um esquema de um estado possível da estrutura, depois de dados serem inseridos na mesma, de modo a tornar a sua visualização mais simples:

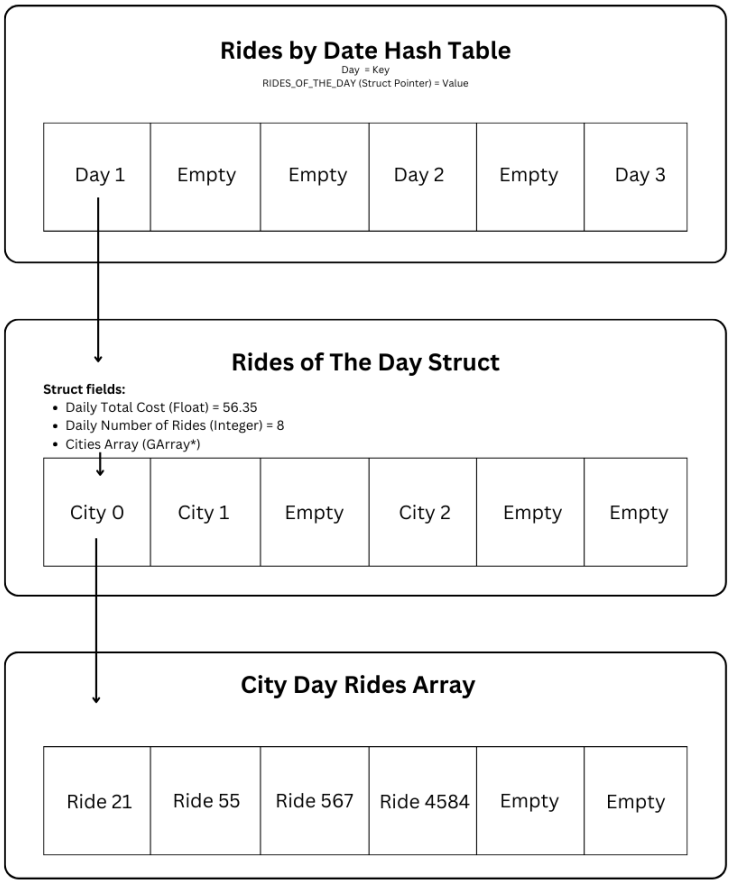


Figura 3.1: Diagrama de uma possível disposição da estrutura Rides by Date

A *query* 5 faz uso desta estrutura para procurar se existem viagens dentro do intervalo pedido (aproveitando o *lookup* instantâneo das *hash tables*), e adiciona custo e número de viagens total nesse dia ao total do intervalo que deseja procurar. No entanto, se o valor dos mesmos for -1, significa que o valor total de todas as viagens desse dia ainda não foi calculado. Esse valor será calculado pela *query*, de modo a que *queries* posteriores com sobreposições em intervalos de dias aproveitem esse valor já calculado em vez de o terem de recalcular de raiz. No final, a soma dos custos totais de todos os dias no intervalo é dividido pelo número de viagens realizadas em todos esses dias, obtendo assim a média do preço das viagens durante o intervalo indicado.

Query 6: Esta *query* é muito semelhante à anterior, mas pede-nos a distância média em vez do valor, e para além de também restringir os dados pertinentes a um intervalo de tempo, limita-se a ter em conta viagens em uma única cidade, fornecida pela *query*. Esta query utiliza a mesma estrutura auxiliar da anterior, e limita-se a aceder a todos os dias dentro do intervalo, verificar se existem viagens realizadas nessa cidade, e adicionar as distâncias das mesmas a um total, tal como o número total de viagens realizadas. Depois de todos os dias no intervalo serem tidos em conta, a distância média é calculada e devolvida.

Query9: É pedido para *"listar as viagens nas quais o passageiro deu gorjeta, num certo intervalo de tempo, ordenadas pela distância percorrida"*. Esta *query* faz também uso da estrutura de estatísticas *Rides by Date*, percorrendo todos os dias dentro do intervalo e registando todas as viagens do mesmo num *array* auxiliar. Finalmente essas *rides* são ordenadas pelos critérios definidos e devolvidas.

Male/Female Rides by Date

Query8: *"Listar todas as viagens nas quais o utilizador e o condutor são do género passado como parâmetro, e têm perfis com x ou mais anos. O output deverá ser ordenado de forma que as contas mais antigas apareçam primeiro."*. Dependendo do género fornecido como argumento, a estrutura de estatísticas *Male/Female Rides by Age* é utilizada. As estruturas são iguais, mas separadas por género, e ambas registam todas as viagens onde o condutor e o utilizador possuem o mesmo género. O *array* escolhido é simplesmente ordenado segundo as idades do utilizador e/ou condutor e devolvido.

Capítulo 4

Otimizações

Após concluirmos todas as *queries*, sentimos que a *performance* do programa estava longe de ser ideal. O tempo de execução do *dataset* grande aproximava-se de um minuto e meio no servidor da equipa docente e a memória estava quase nos 4 *gigabytes*. Logo, apesar do tempo de execução não ser um critério específico de avaliação, tentamos reduzi-lo ao máximo. Tal apenas seria possível observando as nossas implementações originais e perceber quais eram os seus pontos fracos, alterando-os para soluções mais eficientes. Para isso, utilizamos muitas vezes ferramentas para análise de performance - nomeadamente, o *gprof* e o *callgrind*.

A primeira das nossas ideias foi repensar a nossa implementação das datas. Até esse momento as datas estavam definidas como uma *struct* com três inteiros, representando cada um deles o dia, mês e ano respetivamente, mas resolvemos alterá-lo para apenas um inteiro com o formato AAAAMMDD. Desta forma, para além de gastarmos um terço da memória com datas, a comparação entre datas passou a ser instantânea pela subtração, enquanto anteriormente era necessário comparar ano, em caso de igualdade o mês, e em caso de igualdade o dia. Esta otimização pode parecer simples, mas reduziu o tempo de execução do programa em quase três segundos.

A partir daí continuamos a otimizar o nosso programa, e conseguimos superar a nossa meta de 45 segundos. Atualmente, o programa consegue processar o maior *dataset* em cerca de 40 segundos no *website*. Abaixo encontra-se uma tabela com o nosso progresso em tempo de execução e utilização de memória, e as principais otimizações que foram feitas para obter esses tempos.

Otimização	Tempo	Uso de RAM
Original (*queries* todas implementadas)	69,58s	3,9GB
Codificação das datas em apenas um inteiro	66,72s	3,45GB
Codificação dos driver e ride ids num inteiro	60,45s	2,96GB
Mudança de *GList* para *GArray* nas *queries* 2 e 3	60,13s	2,9GB
Utilização do *qsort* nas funções de sort	59,63s	2,9GB
Introdução da validação do input	63,87s	2,95GB
Codificação da cidade num inteiro	43,93s	2,32GB
Mudança de *GTree* para *GArray* na *query* 7	29,39s	2,2GB
Estrutura *Rides by Date mais complexa* (*queries* 5 e 6)	27,91s	2,2GB
Substituição de *sscanf* por funções mais eficientes	23,85s	2,2GB
Alteração das keys nas hash tables para *pointers*	21,3s	1,565GB

Nota: O tempo foi obtido compilando e executando cada uma das otimizações 10 vezes, removendo o pior e o melhor resultado, calculando a média dos 8 restantes. Pode-se ainda referir que foi medido numa máquina com um CPU Ryzen 7 5700U e 16GB de RAM.

Podemos assim concluir que o tempo de execução foi descendo à medida que implementamos mais otimizações, apenas aumentando quando implementamos a verificação do input, visto que todos as entradas tiveram de ser verificadas, o que adicionou mais tarefas ao programa.

Capítulo 5

Verificação de input

Esta componente do programa, apesar de simples, é crucial de modo a certificarmonos que o *input* está correto, de modo a não causar problemas de dados inválidos ou vazios, resultando em *queries* incorretas.

A implementação destes requisitos, apesar de não muito complexa, tinha de ser eficiente de modo a não impactar seriamente a performance do programa. Inicialmente ponderamos utilizar *regex* para verificar se o *input* cumpria os formatos solicitados, tendo até implementado verificações para alguns campos, mas devido ao enorme tempo de compilação dos mesmos, resolvemos verificar os caracteres um a um, apenas utilizando a função *sscanf* nas verificações mais complexas, mas que posteriormente também acabamos por substituir por verificações diretas, mais eficientes.

Capítulo 6

Modo interativo

Um dos requisitos principais desta fase foi a implementação de um modo interativo, com interface gráfica no terminal. Recorremos à biblioteca *ncurses* para tal, visto que esta facilita a criação de um modo gráfico e uniformiza e simplifica a nossa implementação. O modo gráfico é capaz de:

- Processar *queries* e devolver os seus resultados;
- No caso dos resultados serem grandes demais para serem imprimidos no ecrã ao mesmo tempo, os resultados são paginados e o utilizador pode fazer "scroll" pelos mesmos;
- Um menu de ajuda, detalhando a função de cada *query*.

Um dos pontos fulcrais para o sucesso do modo interativo foi o desenvolvimento de uma espécie de biblioteca de componentes. Basicamente, componentes como menus, *option switchers*, *labels*, entre outros, estão definidos num módulo de componentes e podem ser utilizados por qualquer página do modo interativo, já que recebem *propriedades* bastante modulares.

Figura 6.1: Menu principal, onde estão todas as funcionalidades do programa

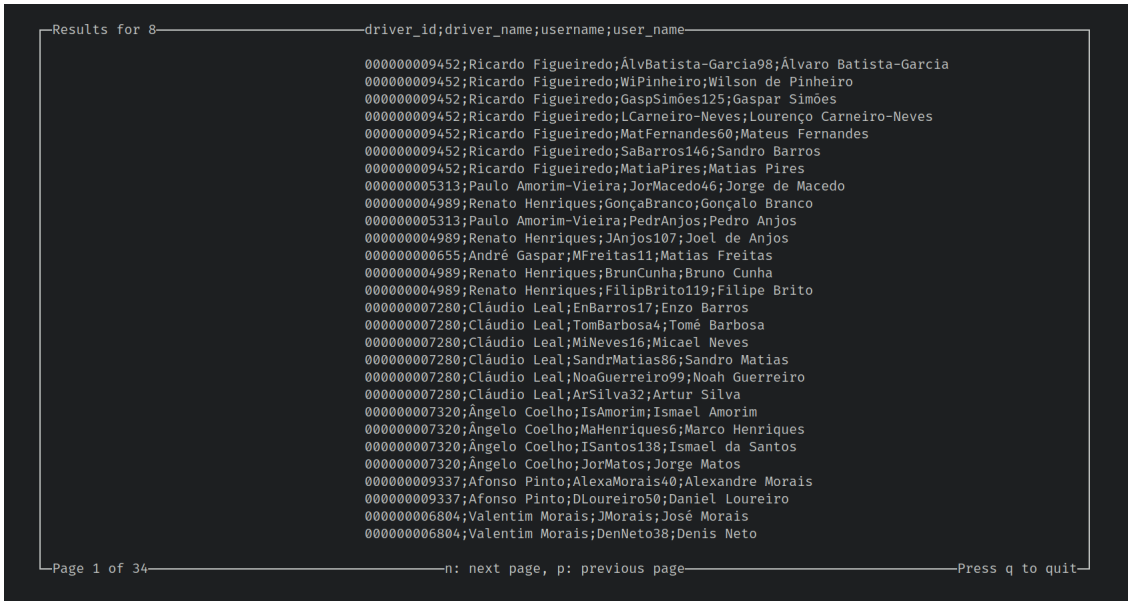


Figura 6.2: Resultados de uma *query* com muitas entradas, onde é possível passar para a próxima "página" de resultados

Capítulo 7

Dificuldades sentidas

Nesta fase, apesar dos desafios serem de muito maior complexidade, como já tínhamos a arquitetura e ideia base de como iríamos estruturar o projeto, sentimo-nos mais organizados e preparados. No entanto, não quer dizer que não tenhamos encontrado dificuldades durante o desenvolvimento.

Inicialmente, tivemos alguma dificuldade em implementar as estruturas auxiliares corretamente, visto que ainda não estávamos completamente confortáveis com a *glib*, de onde utilizamos funções para criar estruturas, e visto que a documentação da mesma poderia ser mais clara em relação a algumas situações específicas.

Também encontramos alguns entraves ao encapsulamento. Inicialmente pensávamos que o nosso programa estava completamente encapsulado, mas mais tarde percebemos que as nossas estruturas de dados estavam a ser diretamente acessados pelos módulos das estatísticas e das *queries*, quebrando assim o encapsulamento. Resolvemos esse problema definindo funções dentro do módulo de cada estrutura. Além disso, existiu sempre alguma dificuldade associada à modularidade do programa. Temos pleno noção que algumas partes poderiam estar mais modulares, como, por exemplo, o módulo do output.

Também existiu alguma dificuldade inicial em implementar verificação do *input* pois encontramos um bug que foi difícil diagnosticar. A origem estava na função em que baseamos o nosso *parser*, a *strtok*, visto que esta reagia mal a campos vazios no *input*, pois alocava um *pointer*, portanto, ao tentarmos acessá-lo atingíamos um *segfault*. Resolvemos este problema substituindo a *strtok* pela *strstr* e alterando o nosso *parser* para a acomodar.

Capítulo 8

Desempenho e testes

Neste capítulo iremos abordar o desempenho final do nosso programa e o módulo de testes que desenvolvemos para validar a correta implementação das *queries*. Para testar a performance escolhemos o *dataset* grande, sem erros, visto que o mesmo, com os seus testes acompanhantes, era o mais demorado, logo era o melhor a demonstrar diferenças de performance entre máquinas.

Para além de realizar apenas as 500 *queries*, fizemos um versão alternativa onde realizamos um total de 2000 *queries*, e registamos também o seu desempenho.

Em baixo apresentamos os resultados do tempo de execução em três diferentes máquinas.

	Máquina 1	Máquina 2	Máquina 3
CPU	Intel Core i5-8300H	Apple M1	Ryzen 7 5700U
Cores/Threads	4/8	8/8	8/16
RAM	16GB DDR4 2666	16GB DDR4 4266	16GB LPDDR4 4200
Disco	512GB SSD M.2	512 GB SSD M.2	512GB SSD M.2
OS	Arch Linux	macOS Monterrey	Arch Linux
Compilador/Versão	gcc 12.2.1	gcc 12.2.0	gcc 12.2.1
Pico de memória	1,641GB	1,641GB	1,641GB
Tempo 500 queries	17,379s	13,501s	17,043s
Tempo 2000 queries	18,960s	14,237s	17,996s

Nota: O tempo foi obtido compilando e executando o programa 10 vezes, removendo o pior e o melhor resultado, e calculando a média dos 8 restantes.

É possível perceber que o tempo das 2000 queries é bastante próximo ao tempo das 500 queries. Isto deve-se principalmente ao sistema de *caching* que desenvolvemos individualmente para cada *query*. Além disso, deve-se também ao referido *workload* inicial do programa.

Em termos de memória, foi atingido um pico de 1641 *megabytes* em cada uma das máquinas, algo pouco significante tendo em conta o tamanho dos dados inseridos e bem abaixo do limite de 4GB pedido pelos docentes. A nossa utilização de memória chegou a estar bem próxima do limite de 4GB, aquando da implementação de todas as *queries*, mas devido às várias otimizações implementadas, já abordadas no capítulo 4, conseguimos reduzir esta utilização aproximadamente em 60%.

Já o módulo de testes faz vários tipos de testes todas as queries, com vários *inputs*, medindo o tempo individual, total e a memória em pico. Este módulo é bastante útil para verificar se o programa funciona corretamente, sendo essencial para novas implementações ou manutenção do programa, pois é possível alguma mudança ter acidentalmente mudado o funcionamento do programa, e estes testes ajudam a diagnosticar tais problemas. O mesmo foi desenvolvido com recurso à *framework* de testes da glib, que é bastante simples e intuitiva de implementar. Além disso, por necessidade própria, desenvolvemos outras formas de testar o nosso programa - desenvolvemos um script em bash que consegue rapidamente testar a diferença entre o conteúdo dos ficheiros entre duas pastas (bastante útil para comparar o resultado esperado, cedido pelos docentes, com o resultado obtido). Desenvolvemos também alguns actions no GitHub, nomeadamente para verificar a compilação, formatação e testar o nosso programa.

Capítulo 9

Conclusão

Resumindo, apesar deste projeto e desta segunda fase, mais especificamente, serem um grande desafio, com muitos conceitos novos e de implementação complexa, sentimos, apesar de termos cometido alguns erros, estarmos à altura do desafio, em grande parte pelo nosso trabalho na primeira fase e as boas bases que desenvolvemos. Sentimos que melhoramos o nosso trabalho em relação à fase anterior em todos os aspetos, e que aplicamos corretamente os fundamentos base deste projeto, como a modularidade e encapsulamento, que pensamos serem conceitos fundamentais que qualquer engenheiro de software deveria ser capaz de aplicar.