RELATÓRIO DO PROJETO LI3

2º FASE

João Henrique da Silva Gomes Peres Coelho - a100596

Duarte Soares Pinto Oliveira Araújo - a100750

José Filipe Ribeiro Rodrigues - a100692

Grupo 05

Licenciatura em Engenharia Informática

Universidade do Minho
Escola de Engenharia

ÍNDICE

Introdução

No âmbito da cadeira de Laboratórios de Informática III foi-nos pedido que realizássemos um projeto utilizando a linguagem de programação C que visa a manipulação de 3 ficheiros, com tamanhos variados, e a utilização dos mesmos na obtenção de informações que consideramos importantes para responder às questões presentes no guião.

Ao longo da realização do projeto necessitámos de conhecimento sobre a manipulação de memória, encapsulamento, modularização e, por fim, estruturas de dados, todos estes conceitos permitiram a realização de um programa mais rápido e eficaz.

Estruturas

As estruturas utilizadas no projeto foram evoluindo ao longo do tempo. As hash tables implementadas mantiveram-se desde a primeira fase exceto a que correspondia às informações dos rides. Esta mudança deveu-se ao facto de que, no final do projeto, notamos que não necessitávamos da estrutura pelo que a mesma se revelou um peso na memória desnecessário. Em vez disso passamos a usar arrays para guardar informações do ficheiro pois permitia-nos ordenar os mesmos, mostrando-se bastante benéfico para a execução de algumas queries.

No entanto foi criada outra hash table para guardar informações dos drivers em cada cidade. Cada elemento da hash table é uma árvore binária de uma nova struct que apenas possui informações necessárias para resolver queries que pretendem obter informações de apenas uma cidade. As árvores são ordenadas, primeiramente, pelos id's dos drivers e, mais tarde, durante a execução do programa, podem vir a ser organizadas pelas avaliações médias com objetivo de facilitar a resolução de uma query.

Encapsulamento e modularização

Depois da defesa da primeira fase foi nos avisado que o nosso encapsulamento não estava bem implementado.

Com o objetivo de resolver o problema apresentado resolvemos, primeiramente, passar cada hash table para o seu respetivo ficheiro em vez de as ter declaradas no ficheiro 'main'. Para respeitar o encapsulamento decidimos declarar cada hash table com o prefixo "static" que bloqueia o acesso à variável a partir de outros ficheiros, mas fica acessível globalmente no respetivo ficheiro.

O projeto foi dividido em vários ficheiros de maneira a maximizar a modularização. Desta forma foram criados ficheiros próprios para cada query e catálogo. Para além disso, foram criados módulos para o parse,

manipulação de queries, manipulação de datas, verificação de entradas inválidas, modo interativo e, por fim, testes automáticos.

Parse

Em relação ao parse mostrado na primeira fase houve bastantes alterações. Como irá ser referido posteriormente na explicação das resoluções das queries, várias informações e estruturas são processadas ao longo desta fase.

Durante o parse do ficheiro *users.csv* é preenchida a respetiva struct e inserida na hash table e num array requerido à query 3.

De seguida é realizado o parse ao ficheiro dos *drivers.csv*. Neste ficheiro é apenas preenchida a respetiva struct e inserida na hash table correspondente.

Por fim é dado parse ao ficheiro dos rides. Neste ficheiro é calculado grande parte das informações requeridas às queries. Para cada viagem é incrementando os valores necessários ao user e driver envolvidos na mesma. Para além disso são adicionadas também as informações necessárias à árvore da cidade. Em adição, a struct da ride é adicionada aos arrays necessários às queries 5, 6, 8 e 9.

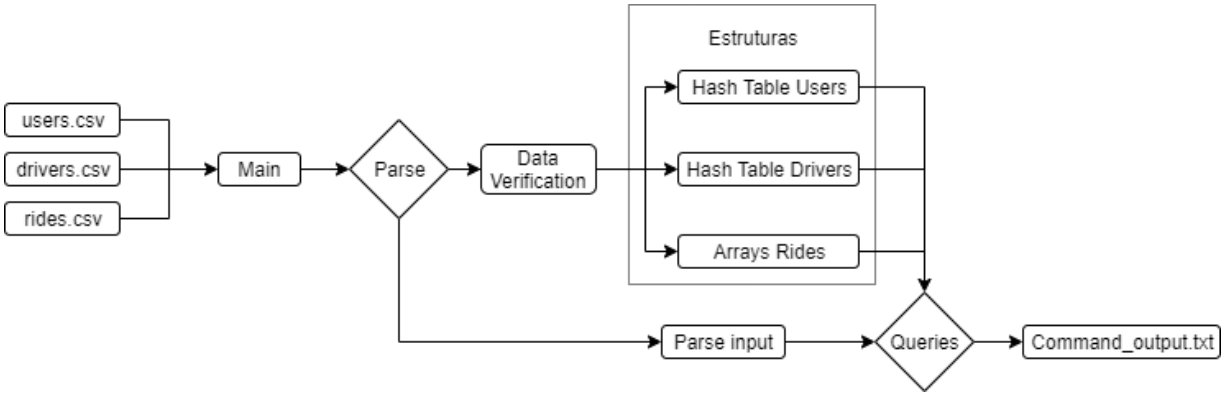

Figura 1: Esquema de funcionamento do programa

Queries

Query 1

A implementação desta query não foi alterada desde a primeira fase do projeto. Visto que todas as informações necessárias à resposta deste problema são calculadas durante o parse, é apenas realizado um lookup na hash table pretendida (users ou drivers) e imprimido as informações requeridas.

Query 2

A segunda query consiste em listar os N condutores com maior avaliação média e, no caso de empate entre este valor, a ordem pela qual os elementos em causa vão aparecer na lista é decidida através da viagem mais recente, e, em caso de novo empate, é decidido pelo id, aparecendo primeiro o que tiver maior id.
De maneira a resolver o problema, na primeira vez que a query é requisitada, é criado um array de drivers que é posteriormente ordenado utilizando o método referido acima com recurso à função *qsort* disponível na própria biblioteca da linguagem C. Assim é apenas necessário aceder ao array e imprimir os N primeiros elementos. Nas restantes vezes o array já se encontra criado e ordenado pelo que já só passa pela fase da impressão do resultado.

Query 3

A terceira query do projeto sofreu algumas alterações ao longo desta segunda fase.

As estruturas que originalmente se encontravam ficheiro query3.c foram movidas para o users.c, para respeitar melhor a modularização, e, atualmente, as estruturas dos users são adicionadas ao *ARRAY_USERS* durante o parse dos catálogos. O array é imediatamente ordenado (recorrendo ao quicksort anteriormente referido, sendo assim mais eficiente) e passa a ser preservado durante a execução do programa, de modo que todas as chamadas da query possam apresentar um reduzido tempo de execução, já que agora apenas é necessário realizar a impressão dos elementos desejados.

Query 4

Esta query pretende que seja calculado o preço médio das viagens numa cidade.

A primeira implementação para resolver a query consistia em percorrer a hash table dos rides de maneira a incrementar o número total e o preço total das viagens na cidade pretendida e, assim, devolver a média pedida.

No entanto, a técnica utilizada anteriormente revelou-se muito ineficiente visto que maior parte da hash table dos rides continha informações de outras cidades. Desta forma, a resolução foi alterada de maneira a utilizar, pela primeira vez, a nova hash table de cidades. Assim, depois de aceder à árvore da cidade pedida, é feita uma travessia para poder calcular os valores pretendidos e responder à query com um tempo de execução bastante inferior ao anterior.

Query 5

A query 5 visa retornar o preço médio das viagens realizadas, excluindo as gorjetas, entre duas datas dadas como argumento.

Na primeira implementação desta query decidimos, sempre que a mesma é chamada, percorrer a totalidade da hash table dos rides e adicionar à estrutura exclusiva da query 5 as viagens que tinham sido realizadas entre as duas datas referidas anteriormente. Após este processo ter sido concluído calcula-se o preço médio dos rides presentes na estrutura e imprime-se o resultado. Apesar do valor obtido estar correto, esta solução da query 5 demonstrou ser bastante lenta e ineficaz logo foi alterada.

A nova implementação da query 5 consiste em percorrer um array de todos os rides ordenado por datas que é criado durante o parse. Este array é utilizado para realizar todas as chamadas da query 5 e 6. No caso da query 5, incrementa-se o valor do preço total e do número de viagens quando a mesma ocorreu dentro do intervalo. A pesquisa no array termina quando se chega ao elemento da estrutura que possui uma data da viagem mais antiga que o limite inferior do intervalo, descartando o tempo de visita a rides desnecessários. Por fim, é feita impressão do valor do preço médio.

Query 6

A query 6 devolve a distância média percorrida, numa cidade especifica, num determinado intervalo de tempo. Inicialmente esta tarefa era abordada através de um foreach que percorria toda a hashtable das rides e recolhia os dados das viagens que cumprissem todos os requisitos, armazenando-os numa estrutura própria para tal. De seguida, as estatísticas recolhidas (distância total e número de viagens) eram usadas para calcular a distância média, e o resultado era impresso. Esta solução provou-se pouco eficiente, logo, um esforço foi feito para a obtenção de melhores resultados.

À semelhança da query 5, esta também utiliza o novo array de rides ordenado pela idade das viagens. A única tarefa realizada pela query atualmente é o somatório das distâncias das rides pertinentes, não sendo necessário procurar por rides a partir do momento em que é encontrada uma viagem com data mais antiga que o limite inferior do intervalo. Por fim é calculada e imprimida a distância média percorrida.

Query 7

A sétima query tem como objetivo devolver uma lista de N drivers numa determinada cidade, ordenados pela avaliação média das viagens na mesma. Na primeira resolução foi adicionada uma hash table à struct dos drivers. Essa hash table utilizava cidades como chaves e cada elemento consistia numa struct que continha as informações desse driver na respetiva cidade, no caso, número de viagens e avaliação total. Sempre que a query era chamada percorria-se a hash table dos drivers de maneira a ir buscar as

structs que continham as informações da cidade pretendida e criava-se um array dessas estruturas. De seguida ordenava-se o array da maneira que a query pedia e imprimia-se os N elementos pedidos. Por fim, o array era libertado.

É fácil de se notar que o método anterior não era eficiente pois, para além de não guardar os arrays criados, fazia um foreach na hash table dos drivers. Desta forma, com o objetivo de melhorar o tempo da query, encapsulamento e modularização, passou -se a utilizar a nova hash table de cidades (já falada na query 4). Assim, visto que já tínhamos todas as informações das cidades agrupadas, apenas reordenamos a árvore, se ainda não tivesse sido feito, de acordo com a query. Depois disso basta fazer uma travessia in-order da árvore para dar os top N elementos.

Query 8

A query 8 procura 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 resultado deverá ser ordenado de forma que as contas mais antigas apareçam primeiro, mais especificamente, ordenar por conta mais antiga de condutor e, em caso de empate, pela conta do utilizador. Se persistirem empates, ordenar por id da viagem (em ordem crescente).

Na primeira implementação da query 8 eram criadas duas estruturas exclusivas que continham todos os dados necessários para o bom funcionamento da query, de seguida, era realizado um foreach na hash table dos rides que adicionava a essas estruturas os dados dos rides que passavam a todos os critérios referidos anteriormente. Rapidamente esta resolução demonstrou ser pouco efizaz e custosa logo a estrutura da query foi alterada.

Decidimos então criar dois arrays ao longo do parse, dedicado a rides em que os sexos do condutor e do utilizador são iguais (maculino ou feminino) e ordenámos os mesmos de acordo com os critérios descritos previamente. Assim, quando a query 8 é chamada, só é necessário percorrer o array que corresponde ao sexo recebido como argumento, e imprimir todos os elementos até que a idade da conta do condutor ou do utilizador não seja superior à que foi dada como parâmetro.

Query 9

A nona query requer que sejam listadas as viagens nas quais o passageiro deu gorjeta, num determinado intervalo de tempo, ordenadas por ordem decrescente de distância percorrida.

Inicialmente, a implementação desta query envolveu o uso de um foreach que percorria a hash table das rides e armazenava a informação das viagens necessárias em estruturas, que por sua vez eram colocadas num array, ao

qual era aplicado um quicksort que tivesse também em conta os casos de desempate.

De modo a otimizar a query, mais tarde, no desenvolvimento do projeto, foi decidido utilizar uma nova estratégia. Passámos a usar um array de rides criado na fase de parse, ordenado apenas uma vez e preservado ao longo do programa. Desta forma, apenas o print dos resultados é realizado quando a query é chamada.

Memória

O uso de memória foi evoluindo ao longo do desenvolvimento do projeto.

A primeira coisa que fizemos para melhorar o uso de memória foi retirar informações desnecessárias das structs dos users, drivers e rides. Desta forma foram retirados os seguintes elementos: método de pagamento do user, placa do carro e cidade do driver e, por fim, comentários da ride. Para além disso alguns tipos foram convertidos: estado da conta e género dos users e drivers passaram apenas a guardar o primeiro char da string, assim como a classe do carro do driver, as avaliações do user e do driver assim como a distancia das viagens passaram a ser guardadas como inteiros em vez de strings.

Outro método utilizado para minimizar a memória usada foi aplicar o conceito de *struct alligment*. Assim, tendo em base os alinhamentos das estruturas, organizamos as mesmas de maneira que ficassem a ocupar o menor espaço possível.

Por fim, e como já foi referido anteriormente, a hash table dos rides foi removida do projeto visto que já não era usada e só revelava ser um peso desnecessário na memória.

Modo interativo

Uma parte do trabalho consiste em desenvolver uma interface no terminal para que o utilizador possa interagir melhor com o programa. Desta forma, quando o programa é iniciado neste modo, o terminal é apagado para só aparecer a interface interativa. Primeiramente é pedido ao utilizador para inserir o caminho para a pasta dos catálogos que pretende abrir para assim poder começar a executar as queries pretendidas.

É apresentada uma tabela que contém um resumo de todas as queries e respetivos argumentos antes de se especificar a query a executar. Depois de ser inserida a query pretendida e respetivos argumentos é apresentado o resultado no terminal.

A interface está dividida em várias páginas de 30 linhas. No final de cada página o utilizador tem a opção de avançar ou retroceder uma página, saltar

para uma página mais distante, existente, e finalizar o programa. Todas as linhas que são impressas no terminal são guardadas numa matriz de strings, tornando assim possível o utilizador viajar entre as páginas sem perda de informações.

No final de cada execução de uma query é inquirido se o utilizador pretende continuar ou terminar a execução.

Testes automáticos

Uma das funcionalidades requeridas no projeto é a capacidade de o mesmo realizar, automaticamente, testes para cada query e demonstrar quanto tempo as mesmas demoraram a executar e se passaram a todos os testes.

Os testes automáticos foram realizados num ficheiro à parte, de maneira a respeitar a modularização, que incluiu uma main que, tal como a main principal, recebe o path para os catálogos e criar todas as estruturas do projeto a partir dos dados presentes nos mesmos. De seguida foram criadas três funções para testar os resultados dos testes com as respetivas soluções, cada função é responsável por testar queries que recebem um número de argumentos específico (1/2/3 argumentos).

Assim sendo, em cada uma das funções é criado o respetivo ficheiro onde o resultado do teste vai ser impresso e, de seguida, é chamada uma função responsável por verificar se o resultado obtido é idêntico à solução ou não. Por fim, dependendo dos resultados dos testes, é impresso no terminal se cada query executou corretamente ou não.

Desempenho

De seguida apresenta-se uma tabela com os tempos de execução médios nas máquinas dos três elementos do grupo, e respetivas especificações, no modo de testes.

O.S.	Ubuntu	Ubuntu	WSL(Ubuntu)
CPU	Intel Core i7-12650H 3.50 GHz	Intel Core i7-1165G7 2.80GHz	Intel Core i7-10875H 2.3Ghz
Cores/Threads	10/16	4/8	8/16
RAM	16GB DDR5 4800MHz	16GB LPDDR4X 4267MHz	16GB DDR4 3200MHz
Disco	512GB SSD M.2	1TB SSD M.2	1TB SSD M.2
Tempo Total	1m 0,9906s	1m 11,848s	1m 28,399s

Query 1	0,00002s	0,00002s	0,00002s
Query 2*	0,66s	0,74s	0,828s
Query 3	0,0027s	0,0032s	0,004s
Query 4	0,0167s	0,016s	0,023s
Query 5	0,107s	0,182s	0,489s
Query 6	0,194s	0,213s	0,477s
Query 7	0,139s	0,172s	0,209s
Query 8	0,0167s	0,0189s	0,031s
Query 9	0,123s	0,236s	0,554s

*O tempo corresponde à primeira chamada da função, tendo um tempo menor que 0,00004s nas seguintes chamadas.

Tabela 1: Tempos de execução para máquinas do grupo

As melhorias aplicadas às queries ao longo do projeto refletiram-se numa grande redução de tempo nos testes automáticos disponibilizados pelos professores. Desta forma, o tempo reduziu de um máximo de 920 segundos para um mínimo de 207 segundos (data-large sem entradas inválidas).

Dificuldades

Uma das maiores dificuldades sentidas foi o manejamento da memória ocupada. No início não tínhamos planos definidos para resolver este problema que surgiu quando usamos os catálogos de maior dimensão. Desta forma, foi preciso muito tempo de pesquisa e de reflexão para conseguirmos chegar a um resultado favorável, sendo que as mudanças foram surgindo ao longo do tempo, quando poderíamos ter utilizado, desde o início, uma estratégia que acomodasse todas as nossas necessidades e assim minimizar o trabalho e o tempo.

Para além disso, as estruturas usadas também se revelaram um problema. Tal como referido ao longo do projeto, este aspeto foi bastante alterado, não só para diminuir o tempo usado, como também a memória. Deste modo, a escolha das estruturas revelou ser um grande desafio, pois, se não pensássemos muito no que queríamos fazer acabávamos por obter um método muito lento, situação que aconteceu visivelmente com a hash table dos rides, que se tornou inútil. Assim, foi preciso dedicar algum tempo a tentar pensar em estratégias melhores.

Conclusão

Apesar de termos concluído todos as tarefas que nos pediram para desenvolver neste projeto, temos a consciência que existem alguns aspetos a serem melhorados. Embora tenhamos melhorado bastante o tempo de execução de todas as queries em geral, para alcançar tal feito, foi preciso sacrificar o tempo de carregamento dos ficheiros e da criação das estruturas necessárias que era bem menor antes de terem sido feitas as alterações. Pensamos que, com mais algum trabalho, este tempo poderia ter sido mais reduzido. Em adição, a memória, mesmo estando dentro dos limites estabelecidos, toma valores não muito ideais, já que, no tempo fornecido, não conseguimos encontrar uma estratégia mais adequada.