PROJETO DE LI3

-Grupo 32-

(Fase 2)

Este projeto tem como objetivo a criação e desenvolvimento de queries, com base nos princípios de modularização e reutilização do código, assim como a implementação de estratégias de encapsulamento e abstração.

O nosso projeto tem a seguinte estrutura:

- **Parsing de dados (Valid entries)**

Para fazer a leitura dos ficheiros *.csv* decidimos inicialmente dividi-lo inicialmente por três ficheiros. Cada um deles corresponde ao parsing de um ficheiro (*drivers.csv*/*rides.csv*/*users.csv*). Na segunda fase mudamos este processo, visto que não fazia sentido estarmos a criar 3 ficheiros praticamente iguais. Decidimos então criar apenas 1 ficheiro contendo todo o parsing.

Para efetuar o parsing criamos uma função que lê linha a linha o ficheiro *.csv* e a cada vez que lê uma linha, envia-a para uma outra função responsável pelo tratamento da linha. Essa separa a linha, que originalmente tem toda a informação separada por ';' em todos os seus parâmetros, inserindo-a na hash table.

Implementamos também nesta 2ª fase uma nova estratégia de parsing. Antigamente utilizávamos um ciclo que separava manualmente os vários parâmetros dos ficheiros para enviar para a hash, mas agora utilizamos apenas o *sscanf* que faz o mesmo de uma forma muito mais rápida e a utilizar muito menos memória.

- **Parsing de dados (Invalid entries)**

Nesta segunda fase, foi necessário implementar um parsing que reconheça quando uma linha do ficheiro é inválida. Para o fazer utilizamos a mesma estrutura descrita acima e apenas adicionamos uma condição antes de inserir na hash. Por outras palavras, só inseríamos na hash table a linha lida se essa fosse válida. Verificamos a sua validade através de funções auxiliares que se encontram no ficheiro *func_auxliliares.c* que fazem essa validação (ex. verificar se a distância é um inteiro maior que zero, verificar se a data é válida, etc.).

- **Catálogo de dados (Hash Tables)**

Decidimos catalogar os dados utilizando hash tables. Decidimos utilizá-las, pelo facto de serem bastante rápidas, quer a executar, quer a procurar um elemento na mesma. Isto acontece, porque quando se dá a inserção de uma linha na hash table, fica com ela associada uma key, que mais tarde pode ser procurada para encontrar essa linha sem ter de iterar pela tabela toda. Note que decidimos criar as nossas hash tables e não utilizar as predefinidas da biblioteca Glib.

Criamos três hash tables que contém todos as linhas do ficheiro drivers, users e rides respetivamente. Decidimos criá-las, porque nas queries são muitas vezes pedidas informações sobre um user/driver/ride específico. Para além destas criamos mais algumas hash tables feitas para certas determinadas queries.

Por último fizemos também uma modificação no tamanho das hash tables. Na primeira fase utilizávamos um define para definir manualmente o tamanho de cada hash, mas para esta segunda fase não seria possível visto que existem datasets com tamanhos diferentes. Decidimos então alocar memória para as hashs de forma dinâmica, ou seja, inserimos linha a linha na hash até que ela encha, quando encher utilizamos o *realloc* para criar mais espaço para a hash, até que o processo termine.

- **Interpretação de comandos**

Este módulo é dedicado a ler o input das queries do ficheiro *.txt*. Efetuamos essa leitura do mesmo modo que efetuamos o parsing de dados, lemos linha a linha. Inicialmente lemos o primeiro caracter da linha para sabermos de que querie estamos a falar. Em seguida é encaminhada para a função dessa querie respetiva assim como o resto da linha contendo o input para a querie.

- **Queries**

Na primeira fase, apenas tínhamos conseguido concluir com sucesso 2 queries, a 1 e a 4. Nesta segunda fase conseguimos executá-las todas, para tal tivemos de mudar a abordagem de como as executávamos.

Iremos agora falar com um pouco mais de profundidade de cada uma das queries:

-Query 1

Esta query era a mais simples de implementar. Tudo o que era necessário fazer era verificar, através das hash tables se o utilizador ou driver colocado como input existe na hash table e se a sua conta está ativa ou não. Se o utilizador/driver não existir ou se a conta estiver inativa teremos de criar um ficheiro de output vazio.

Caso contrário significa que teremos de escrever as informações pedidas na querie. Obtemos estas informações indo às hash tables relacionadas com a função rides, visto que é lá que a maioria da informação pedida está.

-Query 2

Esta query foi das mais complicadas de implementar pelo facto de o output ter de estar ordenado. Inicialmente decidimos criar uma hash table que continha todos os driver e lá dentro continha os parâmetros de desempate necessários a esta query como a sua viagem mais recente e a sua avaliação média de todas as viagens. Em seguida fazíamos um loop por essa hash table até encontrar o maior, em seguida fazíamos o mesmo para o seguindo maior e assim sucessivamente.

Este método tinha um grande problema que era, quanto maior fosse o número de linhas para dar output, mais lenta ela se tornava. Então decidimos modificar o método de ordenação e utilizamos uma função de ordenação da hash table (*qsort*). Assim só era necessário ordená-la uma vez e sempre que esta query fosse chamada era só ir buscar as N primeiras linhas dessa hash e dar output por ordem. Método esse extremamente mais rápido.

-Query 3

Esta query é executada da mesma forma que a query 2, as únicas diferenças são que se tratava de ordenar os users e não os drivers e que os parâmetros de ordenação eram ligeiramente diferentes.

-Query 4

Esta query também foi de fácil implementação. Tudo o que era necessário era calcular o preço médio das viagens numa determinada cidade. Para isso fizemos um ciclo na hash rides vendo a cidade de cada linha, se essa cidade for igual à pedida na query então é necessário ir à linha de hash desse driver ver qual era a classe do carro dele, fazer a multiplicação com a distância percorrida e colocar num somador. Se a cidade for diferente passa para a linha seguinte. No final apenas é necessário dar output a esse somador.

-Query 5

Esta query tem o mesmo princípio da query 4, a única diferença é que em vez de verificar se a cidade é igual à pedida na query, verificamos se a data da viagem está contida entre duas datas de input. Se estiver então soma no somador, caso contrário passa a frente. No final apenas é necessário dar output a esse somador.

-Query 6

Esta query tem o mesmo princípio da query 4, a única diferença é que em vez de verificar se a cidade é igual à pedida na query, verificamos para além disso se a data da viagem está contida entre duas datas de input. Se estiver então soma no somador, caso contrário passa a frente. No final apenas é necessário dar output a esse somador. A grande vantagem desta query é que não precisa de aceder à hash dos drivers, visto que só precisa da distância total.

-Query 7

Esta query executa da seguinte maneira. Em primeiro lugar é criada uma hash table sempre que esta query é chamada que contém apenas as linhas da hash tables rides que têm cidade igual à do input pedido. Em seguida através da função de ordenação *qsort* ordenamos essa mesma tabela pelos parâmetros pedidos. No final apenas é necessário dar output das primeiras N linhas da tabela.

-Query 8

Esta query é a mais lenta de todas e executa de uma forma idêntica à query 7, a diferença é que em vez de inserir na hash table apenas as cidades iguais input, vai inserir apenas as linhas da hash table rides que pertencem ao mesmo género que o input (tanto o driver como o user) e também as contas que têm uma idade maior ou igual a X. E é neste mesmo parâmetro que ela se torna lenta, pois a cada linha das rides é necessário ir a hash table dos drivers e user verificar a idade das suas respetivas contas o que é bastante mais lento do que se isto não acontecesse. No final também existe uma diferença visto que é necessário dar output de todas as linhas que não sejam NULL da tabela.

-Query 9

Esta query executa de uma forma idêntica à query 7, a diferença é que o critério para entrar na hash já não é a cidade e passa a ser se a viagem estiver contida num conjunto de datas. Para além disso não utilizamos o qsort, em vez disso fazemos um loop cada vez que é preciso dar output a uma linha. No final

também existe uma diferença visto que é necessário dar output de todas as linhas que não sejam NULL da tabela.

Função main

A função onde tivemos, e atualmente ainda temos, dificuldade em fazer é a main, visto que necessitamos de ler argumentos passados por input no modo Batch. Neste modo, o programa é executado com dois argumentos, o primeiro é o caminho para a pasta onde estão os ficheiros de entrada. Já o segundo corresponde ao caminho para um ficheiro de texto que contém uma lista de comandos (queries) a serem executados.

Tivemos problemas na sua implementação, maioritariamente pelo facto de termos de abrir ficheiros vindos de um caminho, tanto para ler o ficheiro *.csv*, como para ler o ficheiro de texto que contém os inputs das queries.

Makefile

Outro ficheiro também muito importante de referir é o Makefile. Este ficheiro é essencial para a compilação do programa e criação de um executável. Este ficheiro foi fácil e rápido de fazer, visto que já tínhamos a estrutura do mesmo num guião do Blackboard. Dentro desse ficheiro encontra-se também o comando make clear para eliminar o executável e todos os ficheiros *.o*.

Visto que não utilizamos o Glib existem flags que não foram necessárias incluir no ficheiro. Nesta segunda fase foi necessário modificar o Makefile várias vezes devido à adição de novos ficheiros como por exemplo incluir a libraria *ncurses* e a *flag* -O3 (explicada mais à frente).

Testes funcionais e de desempenho

Este modo tem como objetivo a execução de 3 coisas. A primeira é verificar se o output criado pelas queries está correto, a segunda é verificar o tempo que cada processo demora e por último verificar a quantidade de memória que o programa utiliza.

Para o fazer apenas tivemos de criar uma nova função main que é idêntica à original, mas as únicas diferenças são as funções que contam o tempo que o programa demora a executar, as que contam o tempo que as hash tables demoram a ser executadas, e as que contam o tempo que as queries demoram a ser executadas e conta a memória gasta pelo programa. Tudo isto foi feito com a ajuda da libraria *time*.

Para além disso foi também necessário fazer uma pequena alteração e adicionar uma função que verifica o output das queries. Para isso o utilizador executa o programa sem argumentos e, quando o faz, é lhe apresentado uma caixa de texto onde pode colocar toda a informação necessária para o funcionamento do programa sendo eles os que se seguem. O path para os ficheiros *.csv*, a localização do ficheiro *.txt* e a localização da pasta onde se encontram as soluções dos outputs das queries. Com isso feito apenas foi necessário contar o tempo de execução de cada parcela e utilizar a função *memcmp* que verifica se 2 ficheiros são iguais. Se o forem, o utilizador verá a mensagem CORRECT escrita e verde e se forem diferentes o utilizador verá a mensagem INCORRECT escrita a vermelho.

Executando o programa estas foram os resultados nos nossos computadores:

PC Diogo PC Tomás	Regular dataset without invalid entries	Regular dataset with invalid entries	Large dataset without invalid entries	Large dataset with invalid entries
Tempo de execução do programa	3.762 seg 3.568 seg	4.201 seg 3.879 seg	257.052 seg 246.876 seg	234.214 seg 212.700 seg
Tempo de execução das Hash Tables	1.632 seg 1.556 seg	1.603 seg 1.494 seg	19.678 seg 18.291 seg	19.855 seg 17.634 seg
Tempo de execução das Queries (total)	2.055 seg 2.045 seg	2.680 seg 2.542 seg	242.045 seg 239.462 seg	237.992 seg 222.210 seg
Tempo de execução da Query mais lenta	0.527 seg Input 17 / Query 18 0.542 seg Input 17 / Query 8	0.512 seg Input 50 / Query 8 0.475 seg Input 50 / Query 8	9.732 seg Input 444 / Query 8 8.691 seg Input 487 / Query 8	9.534 seg Input 444 / Query 8 8.450 seg Input 444 / Query 8
Memory leaks (MB)	0 MB 0 MB	0 MB 0 MB	0 MB 0 MB	0 MB 0 MB
Memória total utilizada (MB)	361 MB 361 MB	350 MB 350 MB	3606 MB 3606 MB	3495 MB 3495 MB

* Os valores apresentados são a média de uma amostra de 15 testes diferentes para todos os parâmetros com 3 casas decimais.

Especificações	Especificações PC Diogo	Especificações PC Tomás
Processor	11th Gen Intel(R) Core(TM) i7-1165G7 @ 3.00GHz	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Number of Cores	8	8
Speed	7.7GHz	7.7GHz
RAM	16 GB	16 GB
Video Card	NVIDIA GeForce GTX 1650 4 GB GDDR6	Intel(R) Iris(R) Xe Graphics
Size	64-bit	64-bit
Memory	1 TB	1 TB

Modo Interativo

Este modo consiste em o utilizador executar o programa sem argumentos fazendo com que apareça na consola um menu no qual o utilizador consegue interagir e executar as queries. Para fazer este modo decidimos utilizar a biblioteca "ncurses" pois pareceu-nos a mais simples de utilizar para fazer o menu.

No menu, o utilizador pode escolher 3 opções:

- Inserir o path para os ficheiros .csv. Este botão quando acionado recebe o input do username e é responsável por verificar se o path inserido existe e é valido. Se não for, o utilizador receberá uma mensagem de erro pedindo para inserir novamente. Se for válido então são iniciadas as hash tables e as funções de ordenação das hash tables.
- Inserir uma query. Este botão é responsável por receber uma query do utilizador verificando a sua validade e em seguida executa-a. Se o output da query for maior do que 15 linhas, criamos páginas de modo a facilitar a visualização dos outputs. Quando isto acontece, o utilizador recebe uma mensagem dizendo todas as possíveis interações com esse output. Essas são:
  - Andar uma página para a frente através da seta para a direita do teclado.
  - Andar uma página para trás através da seta para a esquerda do teclado.
  - Ir para a primeira página através da tecla F do teclado.
  - Ir para a última página através da tecla L do teclado.
  - Voltar para o menu utilizando a tecla C do teclado.
- Sair. Este botão apaga todas as hash tables e termina a execução do programa.

Este modo foi ,inicialmente, um pouco difícil de implementar, mas, após nos familiarizarmos com a biblioteca, tornou-se muito mais fácil a sua implementação e utilização.

A parte mais difícil de implementar foi, sem dúvida, os mecanismos de páginas. Tentamos várias abordagens até conseguir fazer com que o programa funcionasse na perfeição com os outputs corretos.

Modularidade e reutilização de código:

A modularidade e reutilização de código, que são muito importantes quer para o desenvolvimento do trabalho, quer para a avaliação dele, foram conseguidos através dos seguintes parâmetros:

- O código dos programas foi dividido em unidades mais pequenas e específicas de modo a não só ser mais fácil de navegar pelos ficheiros, mas também permite a possibilidade de essas "subprogramas" serem chamados mais vezes. Isto ajuda bastante porque é a forma ideal de ter um código mais compacto e fácil de perceber.
- Criamos também um ficheiro apenas dedicado às funções auxiliares que são usadas várias vezes, de modo que se torne mais fácil a sua utilização.
- Os ficheiros *.h* não incluem instruções.
- Utilizamos muito poucas variáveis globais.
- Os ficheiros *.h* são sempre incluídos no respetivo ficheiro.

Encapsulamento e abstração de dados:

Também temos outros processos importantes de implementar, as estratégias de encapsulamento e abstração de dados. A primeira mudança no código que fizemos para implementar estas estratégias foi a implementação do *#define*. Este é importante, para que o user não tenha acesso a valores desnecessários e ter a possibilidade de os mudar. Dessa forma temos a garantia de a front-end está mais segura, visto que os *#define* estão definidos nos ficheiros .h que não são acessíveis ao user.

Para além disso, nos nossos ficheiros *.h* declaramos de forma abstrata as funções, ou seja, nele não fazemos cálculos nem leituras, apenas possuem informações necessárias à execução do programa. Também se encontram declaradas de forma abstrata, ao contrário dos ficheiros *.c*. Por fim os ficheiros *.c* contêm a implementação concreta e completa do código. Um exemplo relevante é nas structs e nos typedef, no qual o typedef deve estar definido no *.h* e a struct no *.c* e em seguida ser acessado pelos outros ficheiros através de funções get.

Por último, uma coisa que não tínhamos feito na primeira fase, mas fizemos na segunda foi que nas funções de procura das hash tables em vez de retornar a linha da hash correspondente, que viola o princípio do encapsulamento, retornar uma cópia da linha da hash com funções de *get*. Isto é recomendável, pois assim, os ficheiros que chama essas hash tables apenas têm acesso a uma cópia da linha correspondente não a podendo alterar.

Memória e performance:

Este é um tópico que achamos pertinente discutir na medida que ambos os parâmetros são importantes na execução e desempenho do programa.

No que toca à memória, infelizmente, não conseguimos baixar muito a memória do programa desde o início, ocupando o nosso programa no *dataset without invalid entries* cerca de 3606 MB de memória. Isto acontece porque nós utilizamos bastantes hash tables (8 para ser preciso) e isso faz com que necessitamos de ocupar muita memória para construir e completar essas hash tables devidamente.

No que toca à performance conseguimos melhorá-la significativamente a cada dia. Para o fazer mudamos o mecanismo de *parsing*, a forma como ordenamos as hash tables e por último a utilização da *flag* -O3 que otimiza a performance geral do código.

Nota:

Uma ferramenta que usamos bastante e achamos pertinente falar dela foi o debbuger (gdb). Esta foi útil a descobrir a localização dos erros, e que valores eram passados como inputs em certas funções. Para além disso, também conseguimos observar em que locais os ciclos estavam a funcionar mal e/ou a nunca acabavam devido a erros dos contadores, etc.

Para além disso também utilizamos bastante a ferramenta valgrind com as flags *–memcheck* e *–tool-check=true* para verificar a quantidade de memory leaks que o nosso programa tinha. A vantagem de utilizar este método é que ele indica todas as linhas ondem existiam memory leaks fazendo com que fosse mais fácil corrigir esses erros.

Fase 1 Projeto vs Fase 2 Projeto

Na fase 2 do projeto, conseguimos fazer tudo de forma muito mais fluída do que na fase 1. Na primeira, demoramos muito tempo a perceber exatamente como fazer as coisas e acabamos por não conseguir terminar nem otimizar tudo, coisa que não aconteceu nesta fase. Começamos a trabalhar na segunda fase cerca de 2 semanas depois da entrega da entrega da primeira e tivemos logo um grande progresso. Nos primeiros 2 dias mudamos totalmente a forma como executávamos as queries e conseguimos praticamente faze-las todas nesse tempo, dando-nos assim muito mais tempo para passar nos problemas seguintes.

Para além disso, também gostaria de referir que da primeira vez que tivemos todas as queries a fucniomar o nosso programa rodava em cerca de 45 segundos, mas devido a optimizações e ideias diferentes conseguimos reduzir esse tempo em cerca de 90%. Facilitando o que se seguia.

Observações gerais:

Analisando o projeto, de forma geral, estamos contentes na medida em que nos esforçamos e conseguimos fazer tudo o que foi pedido com sucesso. Para além disso também temos um programa com excelente desempenho e com uma boa gestão de memória.

Em primeiro lugar, gostamos bastante do facto de termos feito as nossas hash tables. Na primeira fase ficamos um pouco reticentes e indecisos quanto à mudança para o Glib, mas nesta segunda fase isso não foi necessário visto que conseguimos baixar drasticamente o tempo e a memória usada através de várias otimizações e mudança de certos algoritmos. Sendo assim ficamos com um código a rodar o dataset regular without invalid entries em 5 segundos o que é excelente principalmente tendo em consideração que na primeira fase implementamos apenas 2 queries e que essas demoravam cerca de 12 segundos a executar.

Avaliação do grupo:

Para finalizar, gostaria de mencionar como foi feita a distribuição e execução de tarefas entre o grupo para efetuar o projeto. Infelizmente, o nosso terceiro membro do grupo não nos ajudou no trabalho, nem na primeira fase nem na segunda, mesmo após o termos chamado para vir trabalhar muitas vezes. Sendo assim, apenas dois alunos do grupo trabalharam sendo eles Tomás e Diogo.

Nós trabalhamos a maioria das vezes juntos e esforçamo-nos de forma igual para este projeto. Tínhamos dois métodos de trabalho, um deles em que distribuíamos tarefas pequenas para cada um fazer e no final juntávamos tudo ou trabalhávamos os dois ao mesmo tempo no mesmo ficheiro. Em suma, acho que fizemos um excelente trabalho, principalmente na fase 2 concluindo com sucesso todas as tarefas que nos foram apresentadas.

Grupo 32
- Diogo Araújo | a100544
- Tomás Oliveira | a100657