Relatório Fase 2 do Projeto

Laboratórios de informática III

03/02/2022

Gustavo Barros - a100656
Pedro Sousa - a100823
Pedro Viana - a100701
Grupo 15

Índice:

Introdução:

Este projeto foi realizado no âmbito da UC de LI3 do 2º ano do curso de LEI, tendo como principais objetivos a consolidação dos nossos conhecimentos da linguagem C, aplicação dos conceitos de modularidade e encapsulamento, utilização de estruturas dinâmicas de dados, e testagem funcional e de desempenho do código.

Para controlo de versões e hospedagem de código é utilizado um repositório na plataforma GitHub.

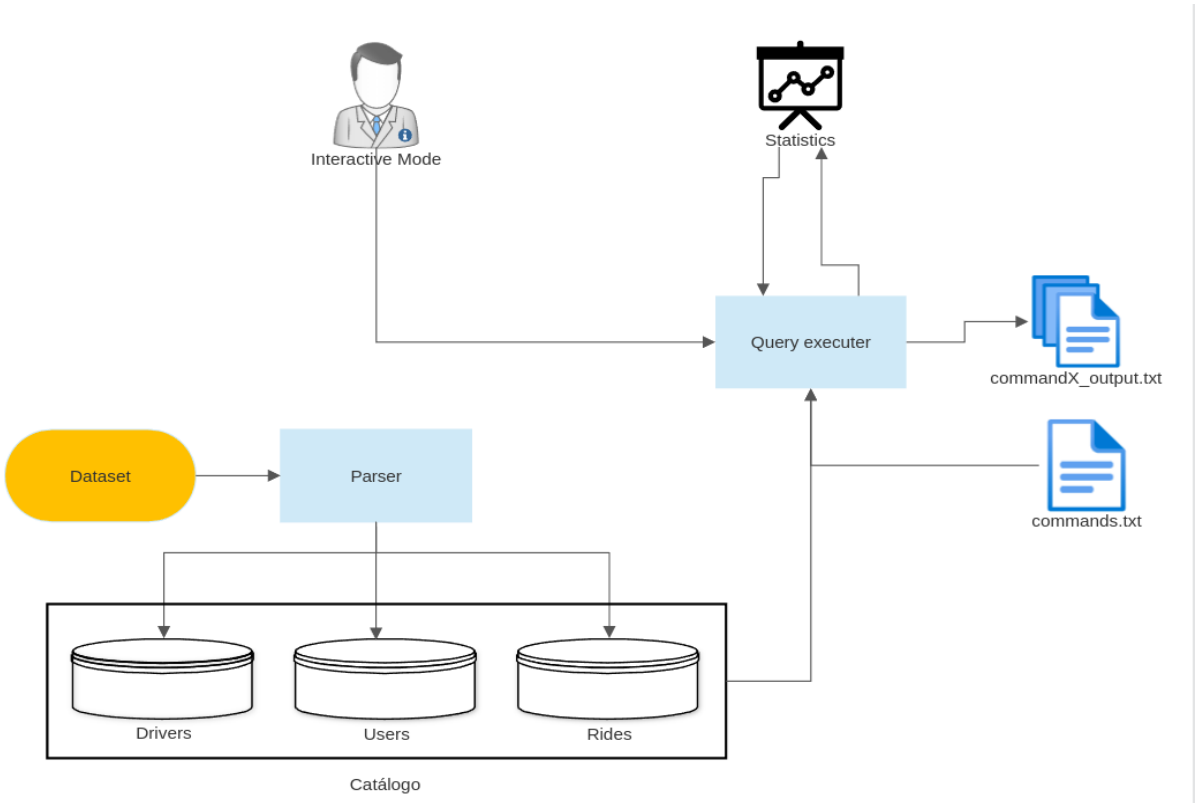
O contexto do projeto consiste em desenvolver uma aplicação, que deverá efetuar o tratamento de uma grande quantidade de dados de uma empresa de transportes guardados em três ficheiros de texto (informações sobre utilizadores, condutores e viagens efetuadas). Estes dados serão então manipulados e armazenados em módulos num catálogo que servirá de base de dados para implementação de 9 *queries*. A aplicação deverá assumir dois modos de execução (*batch* e interativo).

No modo *batch* a aplicação receberá como *input* uma diretoria contendo os três ficheiros de texto(“*users.csv*”, “*drivers.csv*” e “*rides.csv*”) e uma diretoria contendo um ficheiro de texto com as *queries* a ser executadas pela aplicação (“*commands.txt*”). O resultado de cada *query* deverá então ser armazenado num ficheiro (“*commandX_output.txt*”) e armazenado numa pasta “Resultados”.

No modo interativo, o utilizador será apresentado a uma versão interativa do programa, onde poderá escolher o *data-set* usado, *queries* a executar, que informações consultar, etc, tudo dentro de um programa dinâmico.

Estrutura do programa:

O programa assenta numa estrutura complexa conforme a apresentada pelo diagrama abaixo:



Estruturas de dados:

Foram nos fornecidos 3 ficheiros com dados para trabalharmos e manipularmos, nomeadamente: “*drivers.csv*”, “*users.csv*” e “*rides.csv*”.

Para facilitar a manipulação dos dados dos ficheiros foram criadas *structs* para cada tipo de dados, respetivamente as *structs* “*drivers*”, “*users*” e “*rides*”.

Criámos uma *struct* aparte para a manipulação da data, “date”.

Quanto às *queries,* foram criadas *structs* quando achámos necessário, e uma *struct* geral para ajudar na sua execução.

Foi utilizado ainda um catálogo para disponibilizar os dados para as funções, o qual tem também uma *struct* própria “*catalog*”.

Modularidade e encapsulamento:

Durante o projeto, tentamos aplicar os conceitos da modularidade, colocando a definição do código num *header file*, e a implementação do mesmo no respetivo *source file*, e os conceitos do encapsulamento ao tentar agrupar os dados e os métodos que operam nesses dados numa unidade, de forma a ocultar informações desnecessárias a certas partes do programa, e através de funções responsáveis por manipular e interagir com esses dados, como no caso dos catálogos com as *structs*, *user*, *drivers* e *rides*.

Parsing:

A aplicação começa por criar uma base de dados (que o grupo decidiu chamar catálogo) através da função "*catalog_new*", este catálogo consiste numa *struct* composta por três apontadores para cada um dos ficheiros de input ("*users*", "*drivers*" e "*rides*"), um apontador para o ficheiro que contém as *queries* a ser executadas, uma *GHashTable* (*hashtable* da biblioteca "GLib") que serve de módulo para armazenar os dados de cada *user* e utiliza o *username* como *key*, uma outra *GHashTable* com a mesma funcionalidade mas de cada *driver*, utilizando o *id* dos mesmos como *key*, uma *GList* (lista ligada da biblioteca "GLib") para armazenar os dados de cada *ride* e uma outra *GList* para armazenar todas as *queries* que serão executadas.

Ao criar o catálogo é necessário fornecer os 4 apontadores dos *FILES* (previamente associados ao texto de *input*). Todas as outras estruturas de dados começam vazias (*hashtables* e listas ligadas).

A razão pela qual o grupo optou por uma *Hashtable* para armazenar a informação de cada *user* e de cada *driver* foi porque desta forma é muito mais fácil procurar um determinado indivíduo através da sua *key*. Já nas *rides*, concluímos que a maioria das *queries* pede informações presentes numa grande quantidade de *rides*, logo seria muito mais eficiente utilizar uma lista ligada, visto que é percorrida com maior rapidez. O mesmo pensamento foi utilizado para armazenar as *queries*.

Após criar a estrutura da base de dados, o próximo passo seria encher a base de dados com a informação presente nos ficheiros de *input*. Para isso é utilizada a função "*catalog_load*" que é responsável por fazer o *parsing* dos três ficheiros de *input*. O *parsing* é feito de forma bastante simples, dividindo cada linha dos ficheiros (através dos caracteres ';') num *array* de *tokens* mais pequenos. O *array* terá um tamanho variável

dependendo de qual dos 3 ficheiros está a ser tratado, este tamanho será passado como argumento à função *parser*.

Cada *array* é então associado a uma *struct(user, driver ou ride)* e cada parâmetro da mesma é preenchido com o conteúdo do *array*. No caso dos *users* e dos *drivers* a *struct* é inserida na respetiva *hashtable*. Já nas *rides*, a *struct* é colocada no final da lista ligada.

Fica apenas a faltar o *parsing* do *file* que contém as queries a ser executadas. A função "*parse_queries*" é responsável por isto. A estratégia do grupo foi definir uma *struct* para armazenar cada *query*, esta *struct* seria composta por um inteiro que representaria o seu tipo, e por um *void array* que guardaria os argumentos da *query*.

Com este formato, cada linha do ficheiro das *queries* é então associada a uma *struct* e seguidamente introduzido à lista ligada presente no catálogo.

Tendo então a base de dados carregada, temos o necessário para executar as queries.

Queries:

Para cada *query* criámos uma *struct* diferente para armazenar as estatísticas necessárias para resolver a query.

Query 1:

Esta *query* tem como objetivo listar um resumo de um perfil através do seu identificador (fornecido como argumento) este resumo poderá assumir duas formas, pois o identificador poderá pertencer a um *user* ou a um *driver*.

Portanto, o grupo resolveu armazenar o identificador do perfil, a soma de todas as avaliações, o total de *rides* efetuadas, e o valor total gasto/auferido.

O identificador é então associado a um *user* ou um *driver* dependendo do seu formato. Este *user/driver* tem o seu status verificado na respectiva *hashtable* para garantir que a sua conta está ativa.

Para obter os restantes dados necessários, é efetuada uma única travessia da lista ligada que contém a totalidade das *rides*, incrementando os parâmetros da *struct* que armazena as estatísticas caso uma *ride* do identificador seja detectada.

O resultado final é então impresso e guardado no formato pedido.

Query 2:

Nesta *query* é pedida alguma informação sobre os "*N*" condutores com melhor avaliação média (nome, idade e avaliação média). Em caso de empate, seguem-se os critérios de desempate: viagem mais recente, e *id* do condutor (crescentemente).

O grupo resolveu armazenar o *id*, nome, *score* total, número de *rides* efetuadas, e a data da *ride* mais recente numa *struct*.

É então criada uma *struct* para todos os *drivers* com conta ativa e preenchidos os parâmetros do *id* e nome. Cada *struct* é então inserida numa lista ligada. De seguida é efetuada uma única travessia da lista ligada das *rides* e recolhidas as estatísticas necessárias para preencher os restantes parâmetros da *struct*.

No final é utilizada uma função de ordenação (obedecendo aos critérios acima referidos) para ordenar a lista. A informação dos "N" condutores do topo é então impressa e guardada no formato pedido.

Query 3*:*

Esta *query* é semelhante à *query* anterior, sendo que a única diferença é que ao invés de retornar os "N" condutores com melhor avaliação média, teria de retornar os "N" utilizadores com maior distância viajada. Portanto o grupo decidiu adotar uma estratégia muito semelhante.

Armazenando o *username*, nome, *score* total, número de *rides* efetuadas, a data da *ride* mais recente e o total de km viajados numa *struct*.

Em caso de empate os critérios de desempate são: viagem mais recente e ordem alfabética do *username.*

Para obter o resultado da *query* é criada então uma *struct* para todos os utilizadores com conta ativa, inserindo-a numa lista ligada, é efetuada uma travessia da lista das *rides*, incrementando todas as estatísticas necessárias.

Uma função de ordenação (que obedece aos critérios de desempate) é então utilizada para ordenar a lista resultante. A informação dos "N" utilizadores do topo da lista é então impressa e guardada no formato pedido.

Query 4:

Nesta *query* é pedida informação sobre o preço médio das viagens realizadas numa determinada cidade, neste caso teria de retornar o preço médio das viagens na cidade.

Decidimos não usar uma *struct* nesta *query* de forma a poupar memória e por também considerarmos não ser necessário, sendo que podemos obter todos os dados necessários para realizar a query do catálogo.

Começamos por fazer uma função de cálculo do preço de uma *ride*, que consoante o serviço prestado, no caso, a distância e a classe do carro, calcula o preço da *ride*.

Para obter os valores desejados na *query*, usamos dois contadores, que nos forneciam o dinheiro total gasto nas *rides* nessa cidade e o número de *rides*.

No fim imprimimos a média desses dois valores no ficheiro *output*.

Query 5:

Em semelhança à *query* anterior, nesta é pedida a informação sobre o preço médio das viagens realizadas num determinado intervalo de tempo, neste caso teria de retornar o preço médio das viagens nesse intervalo.

Em semelhança com a *query* anterior, pelos mesmos motivos, decidimos não usar uma *struct*.

Utilizámos a função de cálculo do preço de uma *ride*, para obter o dinheiro gasto na mesma.

Adotamos a mesma estratégia para obter o total de *rides* e o preço total das mesmas com os contadores e no fim imprimimos a sua média.

Query 6:

Esta *query* funciona como se fosse a junção da *query* 4 e 5, pois é nos pedido o preço médio das viagens num certo intervalo de tempo numa cidade, deste modo o processo da sua resolução é muito semelhante ao anterior.

Também não foi usada uma *struct* para armazenar dados devido a conseguirmos obter tudo diretamente através do catálogo.

Foi usada novamente a função de cálculo do preço de uma *ride*, e os contadores para armazenar os valores totais de rides e do preço das mesmas.

No fim imprimimos a média dos dois valores no ficheiro *output*.

Query 7:

Nesta *query* é nos pedido os *top* “N” condutores numa cidade, o que é indicado pela sua avaliação, neste caso terá de retornar os condutores (id e

nome) e as suas avaliações médias, ordenados pela avaliação média, em caso de empate ordenar por id de condutor, de forma crescente.

Nesta *query* decidimos guardar dados numa *struct*, destinada a ser preenchida por informações de cada *driver* e as suas *rides* na respectiva cidade, assim como a avaliação de cada *ride*.

A avaliação média é calculada através do número total de *score* a dividir pelo número total de *rides*. Por fim a função é ordenada utilizando uma função feita exclusivamente para esta *query*.

No fim imprimimos os N melhores condutores da cidade(o seu id e nome) e a sua avaliação média.

Query 8:

Nesta *query* é nos pedido todas as *rides* dos condutores do mesmo género e cujas contas têm mais de “X” anos, neste caso terá de retornar por viagem de condutor, o seu id e nome, e ainda o *username* e nome do passageiro. O *output* da *query* deverá ser ordenado, de modo a que apareçam primeiro as contas de condutor mais antigas, em caso de empate, as contas de utilizador mais antigas e em caso de empate, deverá ser ordenado por id de rides de forma crescente.

Decidimos usar uma *struct* para guardar alguns dados pertinentes para a realização desta *query*, com objetivos de calcular a idade da conta e realizar a sua ordenação.

Usámos a função geral que definimos para ser usada em qualquer *query* que fosse necessária, no caso para obter a idade entre duas datas ex: (20/10/2021 - 20/10/2000 = 21 anos)

Usámos ainda uma função de comparação, própria da *query* 8, para comparar e depois devolver um número para a função “*g_list_sort*” fazer a ordenação dos resultados.

Por fim usamos a função própria da *query* 8, para obtermos o *output* num ficheiro à parte, na forma de: id_condutor, nome_condutor, *username*, nome_utilizador.

Query 9:

Nesta *query* é nos pedido todas as *rides* num certo intervalo de tempo e em que o condutor recebe gorjeta, neste caso deverá retornar as *rides*, a distância percorrida, a cidade, a data e o valor da gorjeta. O *output* da *query* deverá ser ordenado por distância percorrida em ordem decrescente, em caso de empate pelas viagens mais recentes e em caso de empate pelo id de viagem em ordem decrescente.

Decidimos usar uma *struct* para guardar alguns dados pertinentes para a realização da *query* e facilitar a sua ordenação.

Para a resolução desta *query* percorremos um *array* de *rides*, e guardámos na *struct* se estas estiverem no dado intervalo de tempo(determinado usando uma função geral que definimos utilizada para manipulação de datas) e tiverem registada gorjeta.

Depois de termos os dados necessários na *struct*, começamos o processo de ordenação usando a função própria da *query*.

Finalmente usamos a função para obter o *output* ,própria também da *query*, num ficheiro, na forma de: id_viagem, data_viagem, distancia, cidade, valor_gorjeta.

Testes:

Foi criado um programa com o objetivo de testar o código escrito, tanto na parte da correção como na parte do desempenho. Este programa leva como argumentos o caminho relativo para a diretoria que contém o *dataset* e o caminho relativo para a diretoria do teste que pretendemos executar.

Testes de correção:

Para testar a correção do programa, o executável corre um certo número de *queries* e compara-as com os resultados esperados que se encontram na pasta de testes. Se o resultado não for o esperado, imprime as linhas onde falhou a correção.

Testes de desempenho:

Para testar o desempenho do programa durante a sua execução, os testes são bastante semelhantes aos testes de correção, a única diferença é que não existem resultados esperados, o programa imprime o tempo de *load* do catálogo, o tempo de execução de cada *query* e no final o tempo de execução total do programa. Para controlar a memória usada no final do programa, o programa também imprime a mesma no final da sua execução.

Conclusão:

Concluindo, conseguimos realizar todas as *queries*, apesar de as últimas duas *queries* apresentarem erros em *datasets* diferentes, obtivemos bons tempos de execução no programa e nas *queries*. Contudo o nosso programa ocupa demasiada memória para *datasets* maiores, visto que excedia a memória para o "*Large dataset*".

Tentámos fazer uma reestruturação do projeto, com ideias como por exemplo implementar uma tabela de *stats* para poupar memória, o que provavelmente iria resolver o problema de excesso de memória gasta no programa, mas devido à escassez de tempo, e a outros problemas que surgiram no desenvolvimento do trabalho não foi possível finalizar esta solução.

Se porventura fossemos fazer alterações futuras, teriamos o objetivo de diminuir o consumo de memória, de modo a que conseguíssemos correr *datasets* maiores com um consumo de memória favorável, corrigirmos os erros nas duas últimas *queries* e diminuirmos os *memory leaks*.

Webgrafia:

- https://docs.gtk.org/glib/