Laboratórios de Informática III

Relatório do Projeto

GRUPO 49

Diogo Pinto
Jorge Rodrigues
Mariana Pinto

Conteúdo

Introdução

O projeto atribuído tem como objetivo dar a conhecer ao grupo os obstáculos de realizar uma aplicação de software em média-larga escala, isto é, com grandes volumes de dados, sendo que estes dados estão contidos em 3 ficheiros (rides, drivers e users) no formato *.csv* (Comma separated values). Para a realização do mesmo, usamos a livraria "glib" em C, que contém diversas estruturas de dados e API's das mesmas.

Estruturação

A estruturação do código consistiu em 3 API's base (rides, drivers e users) que tinham como propósito tratar e organizar os dados lidos a partir dos ficheiros dados. Assim, com o módulo de leitura (read.c) é possível traduzir os ficheiros CSV para estruturas de dados organizadas, de modo a facilitar o desenvolvimento e desempenho das funcionalidades pedidas. Também o módulo Catálogo (model.c) a partir dos módulos anteriores falados teria a capacidade de desenvolver as queries pedidas de maneira bastante simples, isto porque os dados tinham sido organizados e as funções disponibilizadas pelos módulos anteriores.

Por fim, o interpretador (interperter.c) é responsável pela interação entre o utilizador e o programa, através de um ficheiro *input* e os respetivos ficheiros de *Output* gerado na diretoria "Resultados". Para a segunda fase do projeto, é agora introduzido um modo interativo que permite ao utilizador escolher o "Data set" juntamente com as instruções em tempo real, e também um novo executável com o propósito de testar as diversas fases da execução do projeto.

Relativamente ao módulo "table.c", implementamos uma estrutura de dados composta um cabeçalho e um conteúdo. Neste módulo estão presentes diversas funções responsáveis pela inicialização, adição de conteúdo e impressão de uma tabela, bem como a conversão de uma tabela numa outra com uma porção do conteúdo da anterior.

Neste sentido, especial atenção, também, para a função init_entries que aloca memória para o número total de linhas que terá o conteúdo, informação que será conhecida nas queries e realizada antes de adicionar cada linha. De seguida, a função add_entry adiciona linha a linha o conteúdo e copia para os valores de cada coluna. Também incrementa o last para termos conhecimento do último elemento que foi posto na coluna.

Módulos

1. Utilizadores (users.c)

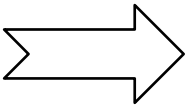
➢ Descrição

Os users, que estão dispostos na respetiva Collection, englobam diferentes parâmetros, como, por exemplo, o username, género, data de nascimento (…).

Relativamente à Users Collection, tem como parâmetro uma GHashTable (tabela de Hash da biblioteca *glib)* em que a key usada para mapeamento de dados é o username e o valor guardado é uma struct user (Users).

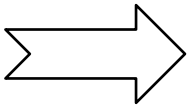
Cada user é armazenado de acordo com a seguinte struct, sendo os parâmetros *score*, *distance_total, total_gasto e num_rides,* obtidos através do ficheiro "rides.csv".

Cada user é armazenado baseando-se nesta struct.



```
struct user {
    char *username;
    char *name;
    char *gender;
    char *birth_date;
    char *account_creation;
    char account_status;
    char *last_ride_date;
    double *score;
    double *distance_total;
    double *total_gasto;
    int*num_rides;
};
```

User collection → os users estão distribuídos nesta collection. A GHashTable contém a key que é o username e o value é uma struct user.



```
struct ucoll {
    GHashTable *table;
};
```

✓ Procedimentos utilizados

- Inits e Frees da Tabela e da Struct User.
- Utilização de Getters e Setters com a função de duplicação (strdup) de forma a garantir o encapsulamento.

2. Condutores (drivers.c)

➢ Descrição

Este módulo agrupa todos os condutores e a respetiva informação contida no ficheiro. Os drivers, que estão dispostos na respetiva Collection, englobam diferentes parâmetros, como, por exemplo, o nome, género, data de nascimento (…).

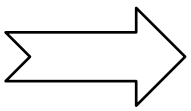
Relativamente à Drivers Collection, tem como parâmetro uma GHashTable (tabela de Hash da biblioteca *glib)* em que a key usada para mapeamento de dados é o id e o valor guardado é uma struct driver (Drivers).

Cada driver é armazenado de acordo com a seguinte struct, onde os parâmetros *score*, *total_gasto* e *num_rides* são obtidos através do ficheiro "rides.csv".

✓ **Procedimentos utilizados**

- Utilização de Getters e Setters com a função de duplicação (strdup) de forma a garantir o encapsulamento.
- Inits e Frees da Tabela e da Struct Driver.

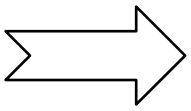
Cada driver é guardado baseando-se nesta struct.



```
struct driver {
    char *id;
    char *name;
    char *birth_date;
    char *gender;
    char *car_class;
    char *account_creation;
    char account_status;
    char *last_ride_date;
    double *score;
    double *total_gasto;
    int*num_rides;
    City list;

};
```

Driver collection → os drivers estão distribuídos nesta collection. A GHashTable contém a key que é o id e o value é uma struct driver



```
struct dcoll {
    GHashTable *table;
};
```

3. Viagens (rides.c)

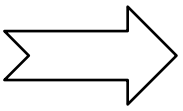
➢ Descrição

Este módulo agrupa todas as viagens e a respetiva informação contida no ficheiro. As rides, que estão dispostos na respetiva Collection, englobam diferentes parâmetros, como, por exemplo, a data, o condutor, a cidade Relativamente à Rides Collection, tem como parâmetro uma GHashTable (tabela de Hash da biblioteca *glib)* em que a key usada para mapeamento de dados é o id e o valor guardado é uma struct ride (Rides).

Cada ride é armazenada de acordo com a seguinte struct, onde todos os parâmetros são obtidos através do ficheiro "rides.csv".

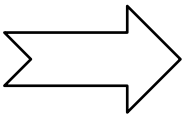
Neste módulo, durante o parsing das rides há certos parâmetros calculados e inseridos no utilizador, como por exemplo, id,utilizador,condutor(...).

Cada ride é guardada baseando-se nesta struct.



```c
// uma entrada ride
struct ride {
    char *id;
    char *date;
    char *driver;
    char *user;
    char *city;
    int distance;
    double tip;
};
```

Rides collection → as rides estão distribuídas nesta collection. A GHashTable contém a key que é o id e o value é uma struct ride.(…).



```c
struct rcoll {
    GHashTable *table;
};
```

✓ **Procedimentos utilizados**

- Utilização de Getters e Setters com a função de duplicação (strdup) de forma a garantir o encapsulamento.
- Inits e Frees da Tabela e da Struct Rides.

4. Leitura (read.c)

➢ Descrição

Nesta secção, os ficheiros .csv “Data sets” são lidos e convertidos nas estruturas de dados adequadas utilizadas no projeto, como por exemplo, os users e a respetiva tabela, rides e respetiva tabela e drivers e respetiva tabela.

❖ Sequencia de procedimentos:

- Verificação de linhas.
- Carregar um ficheiro para a respetiva estrutura de dados
- Converter uma linha na respetiva estrutura de dados
- Estatística.

Na função rides para além de ser feito o parsing e ser criada a tabela de rides também são atualizadas algumas componentes como, por exemplo, o “score” para o respetivo utilizador e driver.

5. Catálogo (model.c)

➢ Descrição

A estrutura de dados contém uma Collection correspondente a uma junção das coleções dos ficheiros anteriores, nomeadamente, Rides_collection, Users_collection, Drivers_collection. Para além disso, possui três “arrays” principais com as “keys” correspondentes às estruturas Users_collection,Drivers_collection e Rides_collection. Os restantes “arrays” são auxiliares às queries, sendo que alguns têm como prepósito ser uma seleção de “keys”, e outro com intuito de melhorar a performance das queries.

Para além disso, conta com uma lista ligada que visa otimizar a querie 4.

```
struct model {
    Rides_Collection rides; //
    Drivers_Collection drivers; //
    char** keyArray_d; //
    char** keyArray_d_2; //
    char** keyArray_u;//
    char** keyArray_r;//
    int *r_size;//
    int *d_size;//
    int *d_size2;//
    int *u_size;//
    char *lastcity;//
    char **starray; //
    Users_Collection users; //
    Q4 backuplist; //
    int starray_size; //
    int how_many_starray; //
    char isIn; //
};
```

Queries

Q1 →

Faz um resumo de um perfil registado no serviço através do ID. É uma querie implementada de forma simples que retorna os dados x, dependendo do input, ou seja, caso seja um id de um driver ou o username de um utilizador.

Q2 →

É responsável por listar os N condutores com maior avaliação média. Em caso de empate, os condutores com a viagem mais recente surgem primeiro. Em caso de novo empate, o id do condutor, por ordem crescente, é o fator de desempate.

Nesta querie, recorremos ao "array de keys", que, juntamente com as funções de *quick sort* do ficheiro "drivers.c", permitiram colocar as respetivas "keys" dos condutores com maior avaliação por ordem decrescente no "array". A ação de ordenação é acima de tudo realizada no *load* dos dados, o que, de certa forma, deixa a função "q2" responsável apenas pela output.

Q3 →

Listar os N utilizadores com maior distância viajada. Em caso de empate, o resultado deverá ser ordenado de forma a que os utilizadores com a viagem mais recente surjam primeiro.

O funcionamento desta querie é bastante semelhante ao da "q2", no sentido em que a ordenação é realizada logo após ao "parsing" dos dados terminar e a função "q3" fica responsável por dar print á output.

Q4 →

Preço médio das viagens (sem considerar gorjetas) numa determinada cidade.

A invocação da função "q4" faz uma travessia sobre a coleção das rides realizando os cálculos necessários. Como forma de otimização, é guardado sob a forma de lista ligada (Estrutura Q4) o preço médio das viagens numa cidade específica. Assim, evitamos que seja necessário iterar mais do que uma vez pela coleção por cada cidade. Para além disso, ainda existe uma lista das cidades existentes, o que por usa vez, caso a input seja uma cidade que não tem nenhuma viagem, não realize nenhuma iteração pela coleção de rides.

Q5 →

Preço médio das viagens (sem considerar gorjetas) num dado intervalo de tempo.

A invocação da função "q5" faz uma travessia sobre a coleção das rides realizando os cálculos necessários. Para isto, é utilizada uma função de comparação de datas para ver se uma determinada viagem que pertença à coleção deve ou não ser contabilizada para o resultado.

Q6 →

Preço médio das viagens (sem considerar gorjetas) num dado intervalo de tempo e numa determinada cidade.

A invocação da função “q6” faz uma travessia sobre a coleção das rides realizando os cálculos necessários. Para isto, é utilizada uma função de comparação de datas e da respetiva cidade para ver se uma determinada viagem que pertença à coleção deve ou não ser contabilizada para o resultado. Como otimização, a função “q6” (à semelhança da “q4”) também confere a lista de cidades existentes para não executar travessias desnecessárias.

Q7 →

Top N condutores numa determinada cidade, ordenado pela avaliação média do condutor. Em caso de empate, o resultado deverá ser ordenado através do id do condutor, de forma decrescente.

Para a realização desta querie, é realizado um “quick-sort” baseado na componente “City” de cada driver, que consiste numa lista ligada com as avaliações médias do condutor pela cidade. Como otimização e de forma a evitar operações desnecessárias (à semelhança da “q4” e da “q6”), é inicialmente efetuada uma procura pela existência da cidade na lista de cidades existentes.

Q8 →

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, com o output ordenado pela conta mais antiga de condutor e, se necessário, pela conta do utilizador.

A solução presente no projeto consiste, primeiramente, em colocar num “array“ as “keys” das rides que vão de encontro às condições referidas anteriormente e, de seguida, ordenar o “array “ de acordo com as regras também antes impostas.

Q9 →

Listar as viagens nas quais o passageiro deu gorjeta, no determinado intervalo de tempo, ordenadas por ordem decrescente de distância percorrida. Em caso de empate, as viagens mais recentes deverão aparecer primeiro. Primeiramente, à semelhança da “q8”, são colocadas num “array” as “keys” das rides que vão de encontro às condições referidas. Posteriormente, é realizado um quick sort de acordo com as condições do enunciado.

6. Interpretador (interpreter.c)

- ✓ Descrição

O interpretador é responsável pela interação entre o utilizador e o programa através de um ficheiro *input* e os respetivos ficheiros de *Output* gerado na diretoria "Resultados", ou em outros casos, realizar testes de performance sobre o funcionamento do programa e recorrer ao modo interativo como forma de correspondência entre usuário e máquina.

7. Estatístico (stats.c)

- ✓ Descrição

O módulo estatístico serve como auxiliar aos restantes módulos e visa melhorar a reutilização do código pelo projeto. Alguns exemplos de funções presentes neste módulo são, por exemplo, a função "more_recent", que realiza comparações entre datas, e a função "get_preco" que realiza os cálculos para o preço de uma determinada viagem.

8. Main (main.c)

- ✓ Descrição

No caso em que a main recebe o número de argumentos necessários ao modo batch, chama o interpretador respetivo a este modo. Caso contrário, chama o interpretador interativo.

O segundo ficheiro "main.c" presente na diretoria "src-teste" só aceita o modo batch, chamando uma outra função que realiza testes de performance.

9. Table (table.c)

- ✓ Estrutura de dados

A struct table presente contém um array de strings (char *) correspondente ao header e uma matriz de strings correspondente ao content.

```
struct table {
    int ncols;
    int last;
    char **headers;
    char ***content;
};
```

✓ Cabeçalho

De acordo com o mencionado na estruturação, a função *init_table* permite a inicialização de uma tabela e dos respetivos campos do cabeçalho, dando *strsep* aos campos separados por espaços.

Número de colunas → Número de campos que o cabeçalho tiver

Last → Último elemento a ser posto na tabela do conteúdo → começa a 0 visto que a tabela só começa com os cabeçalhos iniciados e o conteúdo vazio.

✓ Conteúdo

De acordo com o mencionado na estruturação, o processo abordado é realizado nas queries para todas as linhas de output possíveis.

1. Conversão de table's

Este processo de conversão consiste na cópia de um determinado número de entradas de uma tabela delimitadas por um índice mínimo e um índice máximo para uma nova tabela. Este foi o método utilizado para implementar a paginação.

2. Show (imprime uma tabela)

Função implementada para a impressão de qualquer tipo de tabela, independentemente não só do seu número de colunas bem como do número de linhas pertencentes ao seu conteúdo. A impressão da tabela pode ser feita tanto no terminal como num ficheiro à escolha.

10. Interative.c

Este módulo tem como função criar um interpretador interativo de leitura e execução de comandos.

Neste sentido, criamos uma função (interpretador_interativo) que permite ao utilizador carregar data sets de forma a iniciar os dados do model, bem como realizar queries com o respetivo modelo e obter os seus resultados.

A partir da função (show_func) é possível visualizar os resultados obtidos por cada querie em forma de uma tabela. Esta função possibilita a implementação da paginação na visualização dos resultados e permite ao utilizador navegar as várias páginas da tabela, bem como imprimir o conteúdo total da tabela para um ficheiro à escolha do utilizador.

Aspetos a Melhorar

1. Rides Collection

Apesar das queries apresentarem os resultados esperados, é uma falha no design da aplicação usar uma Tabela de "Hash" como estrutura de dados para o caso das rides, uma vez que seria muito benéfico a existência de uma certa ordem pela data de cada viagem. Assim, seria melhor o uso de uma estrutura como uma "AVL Tree" que proporcionaria travessias com um melhor tempo médio.

2. Utilização de Memória

Apesar de não terem sido ultrapassados os limites definidos para cada "Data set", nós consideramos que a memória utilizada é elevada face ás dimensões dos respetivos dados. Através de ferramentas como o "Valgrind", sabemos que não se trata de nenhuma fuga de memória, concluindo assim que as estruturas de dados utilizadas poderiam estar mais bem otimizadas, como por exemplo, a Rides Collection.

Aspetos melhorados na Segunda Fase

1. Maior eficiência das queries

Relativamente a este aspeto, e devido a mudanças nos algoritmos de sort, as queries estão mais eficientes.

2. Maior eficiência de certas estruturas

As estruturas users, drivers e rides estão mais eficientes, dado que foram apagados campos que não tinham utilidade para as queries e também pela redução do uso de memória de outros campos necessários, como, por exemplo, o account status.

Testes

Specs Máquina 1	Specs Máquina 2	Specs Máquina 3
Virtual Machine running Linux Ubuntu 22.04, Software UTM	macOS 12.5	Virtual Machine running Linux Ubuntu 22.04.1
8 GB de RAM	16 GB de RAM	8 GB de RAM
Processador arm64 (mac m1)	Processador arm64 (Apple M1 Pro)	Processador AMD Ryzen 7 4800h

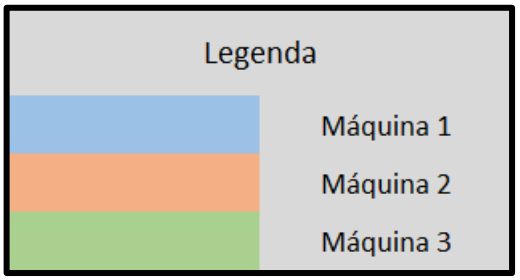
Média do data-set da primeira fase			
Querie 1	0.00000625	0.00000465556	0.00000497
Querie 2	0.000038	0.0000392	0.0000398
Querie 3	0.000061	0.0000442	0.0000492
Querie 4	0.197332	0.1161379	0.16713
Querie 5	0.477900	0.365689	0.49804652
Querie 6	0.1825505	0.14376684	0.193097
Querie 7	0.0156703	0.01999	0.0373664
Querie 8	0.777353	0.64892	0.743658
Querie 9	0.56713	0.4454674	0.36696124
load	3.937486	2.4630272	4.538784
queries	12.423378	9.7342622	11.4290894
Mem usage	332.39	244	360.25

Média tiradas do data-large			
Querie 1	0.000005	0.00001025	0.00000505
Querie 2	0.001575	0.001638	0.0011892
Querie 3	0.002732	0.0019704	0.001676
Querie 4	3.147483	1.422089	1.7901056
Querie 5	5.333999	4.2002378	3.3303417
Querie 6	2.724516	1.4681719	1.7055138
Querie 7	1.436828	0.6375388	0.7890014
Querie 8	11.9733	8.2788746	8.0513309
Querie 9	5.46381	4.1588033	3.3253024
load	57.048675	36.0522582	38.850229
queries	58.682520	39.7050166	37.1993204
Mem usage	3232.75	2450	3437.48

Média tirada do regular-erros			
Querie 1	0.000033	0.00000645	0.00000765
Querie 2	0.000041	0.0000368	0.0000378
Querie 3	0.000572	0.00004	0.0000472
Querie 4	0.261356	0.1342707	0.225235
Querie 5	0.627381	0.4301819	0.5573887
Querie 6	0.184982	0.176136	0.2673518
Querie 7	0.0329312	0.0096171	0.0178315
Querie 8	0.863621	0.6346194	0.8612832
Querie 9	0.568489	0.4196789	0.5610528
load	3.894859	2.3453248	4.305526
queries	4.910207	3.6153314	4.9824524
Mem usage	298.84	201	314.41

Média tiradas do data-large-erros			
Querie 1	0.00006	0.00000755	0.0000051
Querie 2	0.001929	0.0016436	0.0013294
Querie 3	0.002807	0.001963	0.0016316
Querie 4	3.086342	1.3711952	1.7376987
Querie 5	4.95313	4.4213394	3.2158026
Querie 6	2.570580	1.5088103	1.6410585
Querie 7	1.386654	0.7012466	0.8492328
Querie 8	11.181766	8.4958496	7.7641894
Querie 9	5.0532	4.4027155	3.2286473
load	56.1900	35.0279482	37.5669358
queries	55.075	41.112023	36.0293224
Mem usage	3133.14	2390	3358.75

Legenda →



Discussão de resultados dos testes

Após analisar estes resultados, verificamos que todas as queries ficaram abaixo do tempo útil (10 segundos) exceto a querie 8 na máquina 1. De facto, esta mesma querie é mais lenta, por ser obrigada a percorrer as rides todas e a dar sort.

Concluímos, também, que o facto de ter passado do tempo útil na máquina mencionada deve-se ao software da máquina virtual não ser o mais otimizado.

Conclusão

Em suma, entendemos que a estruturação do projeto em 3 API's diferentes foi bem-sucedida, na medida em que consideramos que a modularidade e o encapsulamento foram bem cumpridos. Neste sentido julgamos que cumprimos com as instruções propostas pela equipa docente e que o projeto foi completado com sucesso. Contudo, consideramos que existem melhores abordagens ao problema que conseguiriam otimizar o tempo de execução do programa principal e o respetivo custo da memória.