Introdução

Quando estamos perante uma situação na qual temos de lidar com a manipulação de grandes quantidades de dados existem sempre algumas opções que devemos preferir adotar e outras que, pelo contrário, devemos evitar a todo o custo, portanto, ao longo deste relatório iremos explicar as estratégias que utilizámos para abordar este problema, e porque razão optámos por uma estratégia em específico e não outra qualquer.

Posto isto, importa também realçar que nas estruturas que iremos exemplificar mais à frente tivemos em conta a natureza dos dados com os quais estávamos a lidar e as questões que tinham de ser resolvidas em tempo útil, assim sendo, caso levássemos em conta outros quaisquer fatores, certamente a abordagem adotada teria sido diferente.

Catálogos

Antes de mais, a primeira etapa do projeto diz respeito ao armazenamento dos dados, ou seja, tínhamos de guardar toda a informação útil numa determinada estrutura, portanto, como reparámos que existem três entidades fundamentais *(users, drivers, rides)*, decidimos criar uma estrutura de dados para cada uma destas entidades, e mais uma para as cidades, visto que em várias *queries* as cidades são um dos parâmetros visados.

Utilizadores

```c
/*ficheiro users.c*/
                                    |/*ficheiro users.h*/
                                    |
struct user{                        |#define CAP_USER 10
    char *username;                 |#define BUCKET 8000
    char *name;                     |
    char *gender;                   |typedef struct user* USER;
    char *birth_date;               |
    char *account_creation;         |typedef USER* USERS;
    char *pay_method;               |
    char *account_status;           |
    int sp;                         |
    int size;                       |
    int *positions;                 |
    struct user *prox;              |
};                                  |
```

Estrutura de dados dos utilizadores

Tal como é possível observar pela estrutura presente no ficheiro *users.c*, um utilizador é um elemento de uma lista ligada, visto que um dos campos corresponde a um apontador para outro **`struct user`**. Além disso, existe mais um campo em particular para além daqueles que, como estão presentes no ficheiro *users.csv*, teriam de estar aqui obrigatoriamente, falamos nem mais nem menos do campo **`positions`**. Este corresponde a um *array* dinâmico que começa com um tamanho definido por **`CAP_USER`**, e conforme o valor do **`sp`** (número de elementos do *array*) iguala o do **`size`**, este último multiplica por dois, sendo que os elementos do *array* equivalem a todas as posições em que um determinado utilizador aparece no *array* das viagens.

Passando para o ficheiro *users.h*, percebe-se de que forma os utilizadores estão organizados, neste caso, os utilizadores são guardados numa tabela de *hash*, onde cada utilizador é mapeado para uma determinada posição conforme o seu **`username`**, e acrescentado no início

de uma lista ligada para poupar o máximo de tempo possível. Assim, a estrutura **USERS** corresponde a uma tabela de *hash* com 8000 *buckets*, onde cada utilizador pode ser acedido em tempo praticamente constante, não sendo constante pois é necessário percorrer um *bucket*.

Vantagens

1. Ao utilizar uma tabela de *hash* podemos encontrar um utilizador mais facilmente, uma vez que a função de *hash* mapeia apenas para um determinado *bucket*.
2. Com a utilização de uma lista ligada em cada um dos *buckets* é possível inserir um utilizador no início da lista em tempo constante, algo que não seria possível num *array*.
3. Ao usufruir de uma lista ligada apenas vamos alocar a quantidade exata de memória necessária para representar os nossos dados em cada um dos *buckets*.

Desvantagens

1. A utilização de listas ligadas não tira partido da localidade espacial, dado que os apontadores para os próximos elementos da lista estão dispersos na memória, como tal, a *miss rate* aumenta, e portanto demoramos mais tempo a aceder ao elemento seguinte.
2. Para libertar toda a memória que uma tabela de *hash* ocupa é necessário despender algum tempo, visto que temos que percorrer as listas ligadas presentes em cada um dos *buckets*.

Condutores

```
/*ficheiro drivers.c*/                |/*ficheiro drivers.h*/
                                      |
struct driver{                        |#define CAP_DRIVER 100
    int id;                           |#define CAP_DRIVERS 5000
    char *name;                       |
    char *birth_date;                 |typedef struct driver *DRIVERS;
    char *gender;                     |
    char *car_class;                  |
    char *license_plate;              |
    char *city;                       |
    char *account_creation;           |
    char *account_status;             |
    int sp;                           |
    int size;                         |
    int *positions;                   |
};                                    |
```

Estrutura de dados dos condutores

Agora na estrutura dos condutores, a estratégia adotada foi muito parecida à dos utilizadores, mais uma vez, todos os campos que estão presentes no ficheiro *drivers.csv* também estão aqui presentes, e, além disso, também existe outro *array* dinâmico **`positions`**, que a exemplo do anterior possui um tamanho inicial definido por **`CAP_DRIVER`**, e tem como elementos todas as posições em que um determinado condutor aparece no *array* das viagens.

De seguida, passando ao ficheiro *drivers.h*, vemos que a estrutura **`DRIVERS`** é um *array* dinâmico de tamanho inicial definido por **`CAP_DRIVERS`**, onde cada elemento é um condutor. Neste caso, a forma de mapeamento utilizada para colocar um determinado condutor numa posição do *array* foi o **`id`** do mesmo, ou seja, um condutor com um **`id`** correspondente a **`000000000162`**, será mapeado na posição de índice **`162`** do *array* **`DRIVERS`**.

Vantagens

1. A utilização de um *array* tira partido da localidade espacial, o que aumenta o *hit rate*, e consequentemente leva a um ganho de tempo, caso pretendamos percorrer o *array*.
2. Com um *array* dinâmico nunca corremos o risco de alocar uma quantidade de memória muito maior do que aquela que é extremamente necessária, no pior dos casos alocamos o dobro daquela que seria pretendida.
3. Quando pretendermos libertar a memória do *array* **`DRIVERS`** não vai ser necessário percorrer o *array*, ao contrário da tabela de *hash*, neste caso a operação de libertar memória será feita em tempo constante.
4. Se pretendermos aceder a um determinado utilizador, é possível fazer essa operação em tempo constante desde que saibamos, previamente, qual o **`id`** do condutor em questão.

Desvantagens

1. A menos que saibamos qual o índice de um determinado condutor, teremos de percorrer o *array* **`DRIVERS`** até encontrar o condutor que pretendemos.
2. Ao usufruir desta forma de mapeamento, a posição de índice zero estará sempre vazia, uma vez que nenhum condutor tem um **`id`** equivalente a zero.

Viagens

```
/*ficheiro rides.c*/                |/*ficheiro rides.h*/
                                     |
struct ride{                         |#define CAP_RIDES 100000
    int id;                          |
    char *date;                      |typedef struct ride *RIDES;
    int driver;                      |
    char *user;                      |
    char *city;                      |
    short int distance;              |
    short int score_user;            |
    short int score_driver;          |
    float tip;                       |
};                                   |
```

Estrutura de dados das viagens

Passando agora à estrutura das viagens, vemos que uma viagem possui apenas os campos presentes no ficheiro *rides.csv*, e ao contrário das estruturas anteriores, não possui nenhum *array* de inteiros.

Assim, a estrutura **`RIDES`** está implementada como sendo um *array* dinâmico com tamanho inicial definido por **`CAP_RIDES`**, onde cada elemento é uma viagem. E, novamente, a forma de mapeamento utilizada para determinar qual a posição de uma viagem, é o **`id`** da mesma, ou seja, uma viagem com um **`id`** correspondente a **`000000008439`** é inserida na posição de índice **`8439`** do *array* **`RIDES`**.

Vantagens

- Quando for pretendido percorrer o *array* **`RIDES`**, o programa vai beneficiar da localidade espacial, ou seja, será necessário menos tempo para percorrer as suas posições, dado que os endereços estarão em *cache*.
- Se pretendermos aceder a uma determinada viagem, essa operação pode ser feita em tempo constante, caso saibamos previamente qual o **`id`** da viagem em questão, uma vez que as viagem são mapeadas conforme o seu **`id`**.
- Para libertar a memória alocada pelo *array* **`RIDES`** não é necessário percorrer o mesmo.
- Como o *array* **`RIDES`** é dinâmico, no pior dos casos é alocada o dobro da memória considerada necessária.

Desvantagens

- Sem saber qual o **`id`** de uma viagem é necessário percorrer o *array* **`RIDES`** para encontrar aquilo que pretendemos.
- A exemplo do que acontece no *array* **`DRIVERS`**, também o *array* **`RIDES`** possui a primeira posição vazia, ou seja, no índice zero não existe qualquer elemento.

Cidades

```
1 /*ficheiro cities.c*/                |/*ficheiro cities.h*/
2                                       |
3 struct city{                          |#define CAP_CITIES 2
4     char *city;                       |#define CAP_CITY 100
5     int sp;                           |
6     int size;                         |typedef struct city *CITIES;
7     int *positions;                   |
8 };                                    |
```

Estrutura de dados das cidades

Este módulo foi criado devido ao facto de o parâmetro "cidade" ser requerido em várias *queries*, como tal, ao fazer uma busca de uma determinada cidade no *array* **`RIDES`** seria necessário percorrer o *array* todo, algo que levaria imenso tempo dado o elevadíssimo tamanho do mesmo.

Portanto, para compensar tal debilidade, foi criada a seguinte estrutura, onde uma cidade possui um *array* com todas as posições das suas ocorrências no *array* **`RIDES`**, assim quando quisermos fazer uma busca, já sabemos previamente quais as posições que devemos analisar.

Ao verificar o código, vemos que o *array* **`positions`** é um *array* dinâmico de tamanho inicial definido por **`CAP_CITY`**, e que possui um total de elementos definido pelo valor que o **`sp`** toma. Por fim, o *array* **`CITIES`** segue uma estrutura semelhante, onde **`CAP_CITIES`** define o seu tamanho inicial e cada um dos seus elementos é uma **`struct city`**.

Nota final

Perante esta estrutura de dados certamente fica mais fácil responder às *queries* em tempo útil, contudo uma das grandes desvantagens deve-se ao facto de esta estratégia ocupar demasiado espaço em memória, pois tanto os utilizadores, condutores e cidades armazenam todas as posições correspondentes às suas aparições no *array* **`RIDES`**.

Queries

Durante a resolução das diversas consultas, procurámos sempre fazer tais operações no menor tempo possível, aproveitando ao máximo a estrutura de dados que tínhamos implementado, todavia a nossa implementação não se revelou a melhor para algumas das consultas, e portanto, enquanto algumas *queries* podem ser facilmente resolvidas em tempo linear, outras são resolvidas em tempo quadrático, o que não é ótimo dado o elevado número de elementos que cada estrutura possui.

Query 1

A *query* 1 consiste fundamentalmente em recolher a informação de um determinado utilizador ou condutor, isto conforme o primeiro parâmetro do *input* seja uma *string* de inteiros ou um *username*. Assim sendo, a primeira fase da resolução deste problema passa por identificar qual o tipo de *input* com que estamos a lidar, depois disso, podemos tirar partido do *array* **`positions`** definido anteriormente, e a partir daí obtemos toda a informação necessária.

Primeira fase

Nesta fase temos de saber distinguir se um *input* é referente a um utilizador ou condutor, como tal, decidimos aproveitar ao máximo a função **`atoi()`**, ou seja, como esta transforma uma *string* num inteiro, se a *string* apenas contiver inteiros vai ser devolvido o inteiro correspondente, por outro lado, se o primeiro carácter for uma letra do alfabeto a função vai retornar zero. Assim, desta forma, é possível distinguir um utilizador de um condutor.

Segunda fase

Como já sabemos qual a entidade em questão, resta-nos colocar o **`username`** do utilizador na função de *hash* e obter o índice do *bucket* para sabermos qual a lista que devemos percorrer. Assim, quando estivermos na célula correta, apenas teremos de guardar o *array* **`positions`**, uma vez que este contem todas as posições do utilizador no *array* **`RIDES`**.

Caso estejamos perante um condutor, muda o facto de podermos aceder ao mesmo em tempo constante, visto que este é mapeado conforme o seu **`id`**.

Terceira fase

Depois de obtermos o *array* **`positions`** já sabemos quais as posições que devemos analisar no *array* **`RIDES`**, portanto, a partir daí basta recolher todos os dados necessários, quer para um utilizador, quer para um condutor.

Análise de complexidade

Esta consulta tem uma complexidade pertencente a $\theta(N)$, sendo que N corresponde ao número de viagens realizadas por um condutor ou utilizador.

Query 3

Nesta busca, o objetivo consistia em listar um determinado número de utilizadores conforme a distância que estes tinham percorrido, como tal, é necessário calcular a distância total que cada um dos utilizadores percorreu para depois poder fazer uma análise comparativa.

Primeira fase

Como os utilizadores estão numa tabela de *hash* é necessário percorrer cada uma das listas ligadas para obter todos os dados, assim a primeira fase consiste em ir retornando os utilizadores conforme vamos percorrendo a tabela.

Segunda fase

À medida que recebemos os utilizadores, podemos obter o *array* **`positions`**. Daí segue que conseguimos fazer uma análise tendo apenas em conta os elementos do *array* recebido, portanto basta aceder às posições indicadas e calcular o total acumulado das distâncias efetuadas nas viagens.

Terceira fase

Assim que soubermos a distância percorrida por um utilizador, basta colocar esse mesmo utilizador e a sua distância num *array*, a meio de fazermos uma ordenação e assim listar apenas o *top* N utilizadores.

Análise de complexidade

Esta consulta possui uma complexidade que pertencente a $\theta(N^2)$, sendo que N simboliza o total de viagens que um utilizador fez.

Query 4

Nesta consulta surge o desafio de calcular o preço médio das viagens numa dada cidade, algo que se torna bastante fácil dada a existência do módulo **`cities`**, uma vez que este contem os índices das viagens realizadas em cada uma das cidades.

Primeira fase

Como nos é dada uma determinada cidade, basta procurar essa cidade no *array* **`CITIES`** e retornar o *array* **`positions`** correspondente, assim, desta forma, iremos aceder às posições exatas do *array* **`RIDES`**.

Segunda fase

Assim que obtivermos as posições que devemos analisar, basta aceder a essas posições e calcular o total acumulado do preço, para depois dividir pelo número de elementos do *array* **`positions`**, assim descobrimos o preço médio das viagens numa determinada cidade.

Análise de complexidade

Esta consulta possui uma complexidade pertencente a $\theta(N)$, onde N representa o número de viagens realizadas numa dada cidade.

Nota Final

Durante a primeira fase do projeto o nosso grupo conseguiu realizar mais algumas *queries*, contudo por questões de extensão do documento não nos é possível explicar as restantes consultas. Além disso, algumas delas, apesar de resolvidas, não estão do nosso agrado, e portanto também não faria sentido colocá-las aqui.

De qualquer modo, na segunda fase do projeto iremos focar a nossa análise nas *queries* que não foram aqui mencionadas, a meio de fazermos um tratamento mais rigoroso do desempenho do programa.

Testes de Desempenho

# Comando	Comando	Tempo de execução (s)
1	1 PetrPacheco	0.000050
2	1 ÂngVieira	0.000040
3	1 LeTavares103	0.000021
4	1 NoMaia	0.000021
5	1 ErPinheiro7	0.000052
6	1 FrederAraújo	0.000046
7	1 PeFernandes144	0.000042
8	1 ASousa7	0.000026
9	1 RReis26	0.000022
10	1 TerLima	0.000023
11	1 000000002639	0.000012
12	1 000000008561	0.000027
13	1 000000004987	0.000026
14	1 000000001936	0.000027
15	1 000000003518	0.000043
16	1 000000002581	0.000030
17	1 000000004456	0.000027
18	1 000000000879	0.000026
19	1 000000009721	0.000032
20	1 000000003558	0.000025
21	3 50	0.253828
22	4 Faro	0.018316
23	4 Setúbal	0.013000

Tempos para recolha de dados (s)		
Utilizadores	Condutores	Viagens/Cidades
0.068262	0.022621	2.002561