Relatório da 2ª fase de LI3 – Grupo 70

Os principais problemas que nos foram propostos a resolver durante a apresentação foi a falta de modularidade(tínhamos o Parsing todo num só módulo), um erro que tínhamos no nosso encapsulamento, fazendo com que este fosse basicamente inexistente, o facto de estarmos a ocupar memória com características desnecessárias para as queries e nunca darmos free à memória utilizada. Sendo assim, estes foram os nossos primeiros objetivos para a segunda fase.

1. Evolução da modularidade e encapsulamento

Inicialmente só tínhamos 4 headers mais a main, atualmente temos 16 headers mais a main. Um dos principais problemas da modularidade inicialmente era que antigamente tínhamos na data.c todo o Parsing do projeto. Agora está separado em 6 ficheiros, os de Parsing e os gets para encapsulamento.

No parseusers.c temos o parse de todos users, sendo que temos para um array normal com as informações e outro array de hash para facilitar a execução da query 3 e 8, a criação do array de users, a função que trata da limpeza do array dos users no fim da execução e a verificação dos users.

No parsedrivers.c, temos o parse de todos os drivers, , a criação do array de drivers, a função que trata da limpeza do array dos drivers e a verificação dos drivers.

No parserides.c, além do normal parse que tem dos users e dos drivers (parse das rides, criação do array de rides, função que trata da limpeza do array das rides e a verificação das rides), também passamos a ter uma função putdata que facilitará a comparação de datas, que transforma a data numa struct DATA criada por nós e que também é guardada na struct das rides, uma função scoremedia para calcular a avaliação média que depois facilitará a query 2, que guarda na struct das rides e além disso também calculamos o preço da viagem logo no Parsing para facilitar posteriormente.

Os gets são basicamente a melhor forma de encapsulamento que conseguimos encontrar. Antigamente, estes também se encontravam dentro do data.c, mas agora foram divididos em três ficheiros(getuserdata.c, getdriverdata.c e getridedata.c). O erro que nós tínhamos era que, em vez de mandarmos o conteúdo necessário, mandávamos mesmo o apontador da característica necessária, arriscando a alterar o conteúdo dela. Felizmente nenhuma função alterava o conteúdo e não tivemos erros, mas isto resultava em praticamente nenhum encapsulamento. Sendo assim agora utilizamos strdup, fazendo com que não seja possível mexer no conteúdo.

No getuserdata.c, além de termos funções para ir buscar cada parte do user (exemplo: get_username, faz strdup ao username daquele user em específico e manda para a informação para onde ela foi requisitada, sem arriscar alterar o conteúdo requisitado), agora além desses gets também temos o userisnull. Quando fazemos a validação dos dados, se a linha for inválida, simplesmente pomos naquela posição NULL, para facilitar organização. Sendo assim, sempre que temos de correr o array, necessitamos dessa função de modo a que a execução seja mais rápida, ignorando essas posições.

Quer no getdriverdata.c e no getridedata.c, temos funções parecidas a esta com o mesmo motivo e método. A driverisnull e a rideisnull, respetivamente. O resto desses módulos tambem são semelhantes ao getuserdata.c. Funções para ir buscar cada parte do driver e da ride necessária.

Ainda acerca da modularidade, continuamos a ter o módulo calculos.c onde fazemos cálculos básicos para certas querys e Parsing, o querys.c que agora já tem todas as querys implementadas e o queryscheck.c que antigamente só tinha o query1check para verificar se era user ou driver e agora também tem query6check e query7check que serve para melhorar o tempo de execução destas duas querys, pois temos os arrays das cidades calculados previamente no Parsing.

Além dos novos módulos para o Parsing e para o encapsulamento, também criamos mais alguns e dois headers para facilitar execução. Os dois headers que adicionamos foram o maxs.h e o structsaux.h, o primeiro para ajudar no caso em que os files superam um certo tamanho (2.ª fase) e no caso dos testes estarem ativos, o segundo para ajudar em certas querys que necessitam de structs novas para melhorar a velocidade delas.

Os novos módulos foram o estatisticas.c, o hashing.c, o setup.c, o interativo.c e o tests.c. O estatisticas.c serve para fazer cálculos para certas querys logo após o Parsing e antes de vermos quais vão ser as querys executadas. O hashing.c foi o melhor método que conseguimos arranjar para nos ajudar na query3 e query8, onde temos as funções que servem para definir a key de cada espaço e como será feita a procura. O setup.c foi criado para organizar melhor a main quando criámos o modo interativo. Todo o setup necessário para correr o programa encontra-se definido lá, desde a chamada dos parses até à handle que seleciona a query chamada. O interativo.c é onde está programada a maior parte do nosso Modo Interativo. O último módulo foi testes.c que serve para verificar se as querys estão bem feitas.

2. Melhorias na memória

Inicialmente, reparamos que estávamos realmente a guardar partes dos CSV desnecessárias para as queries, porque simplesmente nunca eram utilizadas, ou seja, era memória ocupada que nunca chegava a ser útil para a execução em si.

Sendo assim, começamos por deixar de as guardar e simplesmente quando deteta que está nessa parte do CSV ignora-a e passa para a próxima parte importante que será guardada na struct.

No parse dos users deixarmos de guardar o paymethod, visto que não é necessário para nenhuma query. Nos drivers deixamos de guardar a license plate. Nas rides deixamos de guardar o comment.

Outro problema que tínhamos era que tínhamos um memory leak enorme, cerca de 17 MB, visto que não utilizávamos frees, ou seja, quer no final do programa, quer a meio, se utilizávamos alguma memória e a deixávamos de utilizar ela simplesmente continuava a existir apesar de nunca mais ser utilizada. O maior leak devia-se ao facto de no final do programa não libertar-mos a memória dos arrays dos users, drivers e rides, logo essa enorme quantidade de memória era toda leaked.

Para resolvermos estes problemas, começamos a fazer free sempre que requisitávamos algo e depois não era mais necessário (exemplo: sempre que usávamos um gets que usava o strdup) e no final do programa damos free a todas as structs auxiliares que utilizaram o malloc. Para libertar os arrays dos users, dos drivers e das rides foi mais complicado, porque dávamos malloc a cada posição, logo tivemos de criar uma função de limpeza dos arrays que já foi mencionada previamente. Para os users é a freeuserarray que se encontra no parseusers.c que sempre que vê que uma posição não é NULL dá-lhe free. Para os drivers é a freedriverarray que está no parsedrivers.c e para os rides é a freeridearray que está no parserides.c, estas funcionam com o mesmo método.

TESTE 149(Regular dataset, without invalid entries)

TESTE 482(Regular dataset, without invalid entries)

Dá para ver que apesar de agora termos mais duas queries feitas e muitas outras reescritas com structs auxiliares, conseguimos diminuir o memory leak em cerca de 71.67%.

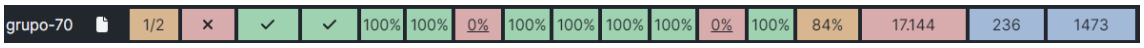
A memória máxima ocupada também diminuiu cerca de 77.60%, devido aos frees e a um melhor management dela.

No modo interativo, usamos goto para certificar que também damos free a tudo.

3. Melhoria do tempo de execução

Um dos problemas que nos deparamos no início da segunda fase foi que teríamos de utilizar ficheiros aproximadamente dez vezes maiores que os que utilizávamos até agora. Devido a esse facto, o nosso programa passaria a demorar muito mais tempo do que o tempo máximo autorizado, visto que já nos ficheiros menores ele demorava quase o tempo máximo autorizado (300s), com o aumento dos ficheiros seria impossível o nosso programa correr a tempo.

TESTE 149(Regular dataset, without invalid entries)



Como dá para ver, o programa mesmo com duas queries a menos demorava 236 segundos, sendo que o máximo era 300 segundos, logo só praticamente um minuto a menos. Para resolver isso aplicamos várias estratégias.

Uma das estratégias apresentadas pelo professores era ter mais cálculos realizados logo após o parsing e antes de chamar as querys. Para isso criamos as estatísticas.c. Nesse módulo fazemos praticamente os cálculos todos de 4 queries previamente.

A função query2est faz o cálculo todo necessário para a query2 pois este já calcula a avaliação média de todos os drivers e faz o sort deles, logo quando é chamada a query 2 ele simplesmente vai pegar nos N últimos drivers do array que foram pedidos.

A função query3est é parecida com a query2est pois deixa a query3 também toda calculada quando for chamada, visto que as duas calculam um Top N de um certo parâmetro, logo simplesmente calculamos tudo e damos sort ao array e depois quando a query é chamada podemos dar imediatamente os resultados. Esta query fez com que tenhamos de criar as funções de hashing.

A função query7est já é mais complexa pois apesar de se chamar query7est também ajuda nos cálculos da query6. Como ambas estas funções são sobre determinadas cidades, nós simplesmente procedemos a dividir as rides todas por cidades logo no Parsing e assim os cálculos serão muito mais rápidos posteriormente.

Nas estatísticas.c também temos a função hashsearchuserdist que é uma das funções que nos ajudou no hashing, que foi necessário implementar para a query 3 e query 8. Além desta função, também temos o módulo previamente mencionado, o hashing.c. Neste módulo tem duas funções, hashkey e hashsearch. A função hashsearchuserdist é simplesmente uma implementação da função hashsearch para a struct USERDIST. A função hashkey tem dois valores arbitrados (o p foi encontrado por tentativa e erro e foi o que teve melhor aleatoriedade, o m foi o maior número primo que encontramos que podia ser utilizado) e serve para atribuir uma key a cada user específico.

Antes estávamos a tentar usar strcmp, mas era muito lento por isso decidimos utilizar este tipo de encriptação para a query3 e a query8. A hashsearch serve simplesmente para encontrar a posição daquela key que é pedida que nos irá levar para o user pedido.

Outro problema que encontramos foi que a comparação de datas fazia o programa perder muito tempo nessas comparações, por causa do strtok especificamente. Sendo assim, criamos a struct DATA que tem três parâmetros: o dia, o mês e o ano. As datas passaram a ser guardadas assim logo no Parsing e deu logo para reparar numa maior rapidez na comparação das datas em certos casos.

Uma parte do código que ajudou significativamente no tempo de execução foi o uso de structs auxiliares.

```c
typedef struct
{
    int id;
    char *nome;
    char *datarecente;
    double somascore;
    int nviagens;
    double avmedia;
    char *activity;
} DRIVERMEDIA; // query2

typedef struct
{
    char *username;
    char *nome;
    char *viagemrecente;
    int dist;
} USERDIST; // query3

typedef struct
{
    int id;
    char *nome;
    double avmedia;
    char *activity;
    int nviagens;
} CITYMEDIA; //query7

typedef struct
{
    int id;
    RIDE a;
} RIDE2; //query7 query9
```

DriverMedia: esta struct é utilizada para a query2. Esta struct guarda o número de viagens e a soma dos scores, que serve para depois calcular a avaliação média de cada driver.

UserDist: struct utilizada na query3. Serve para guardar a viagem mais recente e a distância percorrida do user e para depois na utilização da função hashsearchuserdist não ter de ir ao userisnull

CityMedia: cada cidade tem a sua citymedia para facilitar os cálculos na query7, pois assim os cálculos só são focados naquela cidade em específico

Ride2: serve para facilitar o acesso ao id da viagem no sort

Uma pequena alteração que resolvemos fazer foi também adicionar o -O3 ao Makefile, que melhora o programa principalmente na execução de ciclos.

TESTE 420(Large dataset, without invalid entries)

grupo-70	1/2	×	✓	✓	100%	100%	10%	100%	100%	100%	100%	60%	100%	98%	-	713	3225

TESTE 424(Large dataset, without invalid entries)

grupo-70	1/2	×	✓	✓	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	-	658	3791

Apesar de termos adicionados duas queries, o tempo diminui cerca 7.71%.

TESTE 482(Regular dataset, without invalid entries)

grupo-70	1/2	✓	✓	✓	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	4.857	10	330

Dá para ver que houve uma redução drástica do tempo de execução, cerca de 95.76% no regular dataset.

4. Mudanças para a 2ª fase

As principais mudanças do projeto para a segunda fase são: o aumento do tamanho dos dados utilizados, a validação dos dados utilizados e a criação dum modo interativo e de testes.

A nossa primeira preocupação foi o alargamento dos dados, pois o nosso programa no estado em que estava não conseguia corre-los, pois além de ser muito lento, nem tinha capacidade para tais ficheiros, porque tínhamos definido um máximo para cada ficheiro. Sendo assim, como sabíamos que o novo data large era aproximadamente dez vezes maior criamos duas funções para o caso em que não era o data normal.

A função countlines corre o ficheiro dos users e se o tamanho dele for maior que o maxuser atual, que se encontra definido no maxs.h, então a função setdimensions aumenta o tamanho de todos os maxs. Os maxs encontra-se definidos num header à parte para todos os módulos terem acesso a esses maxs, por isso utilizamos o extern.

Além disso, também temos nos maxs.h o extern int interativo e o extern int testes que serve para o programa saber se se encontra em modo interativo, modo batch ou modo de testes.

Uma mudança que resolvemos fazer na 2.ª fase foi alterar a main para facilitar a diferenciação entre o modo interativo e o modo batch. Sendo assim criou-se o setup.c. É no setup.c que se encontram a countlines e a setdimensions previamente mencionadas. Além dessas duas funções, também tem o pathsetup que serve para termos o path de onde estão os ficheiros CSV, o setuparraysaux que inicializa os arrays auxiliares das structs auxiliares, o parsing que faz o parse dos ficheiros (manda cada ficheiro para a sua respetiva função de parse) e a handle que depois de receber o queryid envia-a para o seu respetivo caso (queryid=6, vai para o query6check com os respetivos ficheiros necessários).

Na handle em si há casos que não manda imediatamente para a função da query em si, mas sim para os querychecks que estão no queryscheck.c. Como dito previamente, agora além de ter querycheck para a query1, também tem para a query6 e 7. Enquanto que para a query1 era só diferenciar entre user ou driver, para a query6 e 7 tem o trabalho muito mais importante. No parsing, nós fazemos estatísticas para a query7, dividindo as rides pelas diferentes cidades e tendo os cálculos feitos para cada cidade. Na 1.ª fase, só teríamos de verificar se a cidade pedida era Lisboa, Braga, Porto, Faro ou Setúbal. Agora na 2.ª fase tivemos de adicionar as cidades Coimbra e Vila Real. A adição dos querychecks para estas duas querys tornou-as extremamente rápidas.

Depois de feitas estas alterações na main, procedemos à criação do modo interativo.

5. Modo interativo

O nosso programa para detetar que se encontra em modo interativo, vê se o argc é somente igual a 1, significando que só tem um parâmetro. Se isso for verdade, ele dá início ao modo interativo.

A primeira coisa que ele pede é o caminho para os files, por exemplo: `Caminho: /home/pedro/Desktop/LI3/dados1`

Ele dá scan ao caminho dado e depois faz o pathsetup normal como no modo batch. Dá uma mensagem a dizer que está a carregar os ficheiros de dados e começa a abrir os ficheiros e faz setdimensions para ver se não é o data large e definir o seu tamanho caso seja. Faz os mallocs das structs auxiliares, o seu setup e o parsing, informa que vai fazer as estatísticas, faz as estatísticas e depois mostra o menu.

No nosso menu, é possível escolher entre apresentação das instruções para cada query ou se não. Depois dessa escolha basta escrever a query que se quer fazer e o resultado aparece na consola. Logo depois de aparecer o resultado, o programa requisita novamente e podemos fazer outra query.

Quando quisermos terminar o programa, basta escrever 0 em vez de por uma query e o modo interativo termina.

Antes de o programa fechar oficialmente, temos um goto para frees para passar à frente partes do modo batch e libertar toda a memória para não ser leaked.

```
Bem-vindo ao modo interativo.

Antes de começar introduza um caminho para a pasta de dados:
Caminho: /home/pedro/Desktop/LI3/dados1

A carregar ficheiros de dados...
A criar estatisticas de dados...

Menu:

Ver instruções de execução(1)
Executar querys(2)

1

Instruções:

Query 1: 1 'ID de condutor' ou 1 'Username de utilizador' // Apresenta um resumo do perfil de um condutor ou utilizador.
Query 2: 2 'N' // Sendo N um número, lista os N condutores com maior avaliação média.
Query 3: 3 'N' // Sendo N um número, lista os N condutores com maior distância viajada.
Query 4: 4 'Cidade' // Dada uma cidade, apresenta o preço médio de uma viagem nessa cidade.
Query 5: 5 'data1' 'data2' // Dadas duas datas, apresenta o preço médio das viagens nesse intervalo de tempo.
Query 6: 6 'Cidade' 'data1' 'data2' // Dada uma cidade e duas datas apresenta o preço medio das viagens nessa cidade nesse intervalo de tempo.
Query 7: 7 'N' 'Cidade' // Dado um número e uma cidade apresenta o top N condutores com maior avaliação média nessa cidade.
Query 8: 8 'Género' 'N' // Dado um género M ou F e um número, apresenta todas as viagens em que o condutor e o utilizador são desse género e tem contas com N ou mais anos.
Query 9: 9 'data1' 'data2' // Dadas duas datas, apresenta todas as viagens em que o utilizador deu gorjeta, nesse periodo de tempo.


Introduza query ou 0 para sair:
2 10

000000008899;Rafaela de Barros;3.564
000000007405;Kevin Assunção;3.535
000000007060;Filipe Melo;3.476
000000007082;Camila Pinheiro;3.468
000000000219;Lúcia Santos;3.457
000000007816;Vitória Amorim;3.452
000000003392;Renato do Cruz;3.441
000000006941;Vera Batista;3.436
000000006657;Diego Oliveira;3.435
000000003682;Gonçalo Maia;3.432
```

6. Validação de dados

Uma importante parte da 2.ª fase era a validação dos dados, a qual implementamos cada uma no seu respetivo parsing, ou seja, a verificação dos users era feito no parse dos users, a verificação dos drivers era feitos no drivers e a verificação das rides era feita nas rides.

Sendo assim, criamos três funções: verificacaousers, verificacaodrivers e verificacaorides.

Na verificacaousers, verifica inicialmente se tem username, se não tiver devolve 0 e naquele posição fica um NULL e liberta-se o espaço que tinha sido criado para aquela posição. Será sempre assim quando é inválido. Verifica se tem name, gender, se a birth_date e a account_creation estão bem estruturadas, se tem pay method e se tem account status.

Se tudo estiver certo devolve 1 e a linha é considerada válida, vai ser igual para os drivers e as rides.

Na verificacaodrivers, verifica se tem id e nome, se o birth_day está bem escrito, verifica se tem gender, car_class, licence_plate, city, se a account_creation está bem escrita e se tem account_status

Na verificacaorides, verifica se tem id, driver e user , se a date está bem escrita, se tem cidade, se a distância, o score_user e o score_driver são inteiros maior que zero e se a tip é um decimal maior ou igual a zero.

7. Teste funcionais

Para correr o nosso programa de testes, em vez de correr ./programa-principal com os paths para os CSV e o input, corre-se com ./programa-testes CSV input testes, onde os testes é a pasta onde estão os resultados corretos.

Depois de o programa identificar que está num contexto de testes, corre o programa de maneira normal, até que depois de executar a query, abre-se novamente o output, após este estar preenchido.

A função compare, que se encontra no tests.c, recebe o output criado por nós, a pasta onde estão os testes, o nome do teste em questão e o número do teste em que estamos. Depois de encontrar o ficheiro de testes, envia para a função checkresults (que também está no tests.c) o nosso output, o output de testes e o número do teste.

Na checkresults, comparamos os files com o fgetc e, se nenhum carater for diferente e se nenhum dos ficheiros fechar antes do outro, passamos no teste.

8. Mudança nas queries para a 2ª fase

Todas as queries sofreram alterações pelo menos para correr o modo interativo, pois antes só escrevia no file e agora também pode escrever na consola dependendo do modo em que estamos.

A query1 encontra-se praticamente igual exceto que agora damos free depois de utilizarmos as características que necessitaram de strdup.

A query2, como se encontra agora praticamente toda feita no Parsing, está reduzida à escrita dos top N condutores pedidos que já estão organizados no array por maior avaliação média. De notar que começamos o array pelo fim, pois este está organizado do menor ao maior, justificando assim a utilização do sizequery2 na main.

A query3, tal como a query2, também é maioritariamente feita no parse, logo também é praticamente só a escrita dos top N users por maior distância viajada.

A query4 e query5 ficaram inalteradas, exceto na parte interativa.

A query6 foi extremamente alterada, mas a maioria dos seus cálculos também se encontram feitas no Parsing, mais especificamente no query7est, pois como antes mencionado, este divide as rides por cidade, logo quando esta query é chamada só verifica as rides daquela cidade em específico.

A query7 também se encontra praticamente feita no parse, sendo que depois só recebe a cidade em específico e os top N condutores pedidos, fazendo somente a escrita necessária.

Para a query8, utilizamos uma estratégia de dividir as rides por gender, diminuindo assim as rides para praticamente metade quando forem necessárias. Esta função dá uso ao hashing para facilitar a busca dos users. Tinhamos testado o strcmp, mas era demasiado lento e tivemos de implementar o hashing.

A query9 dá uso à struct Ride2 para facilitar a busca das rides necessárias que têm tip no intervalo de datas requisitado. Esta foi uma das funções mais beneficiadas pelo uso da nova struct DATA.

9. Conclusão

Conseguimos melhorar o tempo de execução do projeto, diminuímos as memory leaks, apesar de não termos conseguido feito totalmente, a memória foi melhor utilizada, tendo notado uma diminuição drástica nos data regular, e o projeto em si consegue sustentar os data large.