PROJETO DE LI3

-Grupo 32-

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**

Para fazer a leitura dos ficheiros *.csv* decidimos dividi-lo inicialmente por três ficheiros. Cada um deles corresponde ao parsing de um ficheiro (*drivers.csv*/*rides.csv*/*users.csv*). Todos os ficheiros têm o mesmo papel e utilizam as mesmas funções, a única coisa que varia são os parâmetros estabelecidos nos vários ficheiros.

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.

Infelizmente, mesmo tentando várias estratégias, o parsing ainda continua um pouco lento. Não está tão rápido quanto gostaríamos, mas faz o seu trabalho. Na segunda fase tentaremos melhorar o código relativo ao mesmo de modo a ter um programa com menos custo computacional e mais rápido.

- **Catálogo de dados**

Decidimos catalogar os dados utilizando hash tables. Decidimos utilizá-las, pelo facto que 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. Mais tarde, temos o objetivo de migrar para essa biblioteca, visto que as hash tables são mais rápidas e mais eficientes.

Criamos uma hash table que contém todos as linhas do ficheiro drivers e outra que contém as linhas do ficheiro users. Decidimos criá-las, porque nas

queries são muitas vezes pedidas informações sobre um user/driver específico, e como já referi, as hash tables encontram-nos muito rapidamente, poupando tempo de execução do programa.

Para além destas criamos mais três hash tables. Note que não criamos nenhuma hash table com o ficheiro de rides, visto ser um processo muito extensivo e pouco eficiente. Decidimos então criar duas hash tables para resolver a querie 1 e uma hash table responsável pela querie 4.

- **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**

Como já referimos anteriormente, as queries que conseguimos fazer foram as queries 1 e 4. Estas conseguem rodar qualquer teste que lhes é colocado. Sabemos isto, porque já fizemos bastantes testes com vários inputs e as queries respondem como deviam. Também mencionamos a querie 6, na qual tentamos fazê-la, mas não tivemos sucesso. Vamos falar melhor de cada uma delas:

-<u>Querie 1</u>

Decidimos abordar esta querie do seguinte modo. Inicialmente, abrimos a pasta Resultados, visto que é la que iremos criar o ficheiro de output de cada uma das queries. Em seguida verificamos, através das hash tables se o utilizador ou driver 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 está.

Nesta querie desenvolvemos duas funções relativamente às rides. Estas funções armazenam nas hash tables todos os id/usernames do seguinte modo. Cada vez que um username/id e lido no parsing, verificamos se esse já existe na hash table. Se não existir, então adicionamo-lo à hash, caso contrário procuramos a linha da hash em que se encontra e atualizamos os dados. Ou seja, somamos ao valor existente o total gasto/ total auferido por exemplo.

Este método é útil na medida em que não existe necessidade de criar uma hash table de um milhão de linhas, apenas temos de criar uma com dez mil linhas e outra com cem mil. Não só isso, mas também ajuda na rapidez da função procura, visto que só precisa de identificar uma key e não múltiplas. Se isso acontecesse teríamos de iterar pelas linhas todas à procura da informação o que não é recomendável.

-Querie 4

Já esta querie não exigiu tanto esforço como a primeira, visto ser mais simples de implementar. Nesta querie apenas criamos uma hash table que contém a informação de todas as cidades. Fizemo-la da seguinte forma:

Inicialmente (quando efetuamos o parsing) inserimos as várias linhas das rides nessas hash table, mas verificamos primeiro se essa cidade já existe na hash table. Se existir, vamos à linha onde esta se encontra e atualizamos a sua informação, ou seja, acrescentamos o total gasto, por exemplo ao que já existia previamente. Se a cidade não existir na hash table, então adicionamo-la. Este método é altamente eficaz, visto que não necessitamos de criar uma hash com um milhão de linhas. Apenas temos de criar uma hash com mil linhas que contém toda a informação necessária. No final, procuramos na hash table a cidade do input e damos o output de acordo com o pedido.

-Querie 6

Gostaríamos também de mencionar a querie 6 que, infelizmente, não conseguimos terminar. Achamos que isso aconteceu devido a limitações da nossa estrutura de dados. Tentamos implementá-la de várias formas e acabamos por ter sempre problemas nas suas execuções. Originalmente criamos uma hash table que continha todas as linhas do ficheiro *rides.csv*, mas achamos que não é uma boa opção devido à sua lenta execução e gasto computacional, para além disso o nosso makefile não dava output, ficava durante vários minutos a processar e não terminava. Logo em seguida, pensamos fazer o parsing de novo, mas não conseguimos abrir o ficheiro rides.csv por um motivo que não conseguimos descodificar.

O maior problema desta querie foi o facto de a começarmos a fazer demasiado tarde. Demoramos bastante tempo a corrigir bugs de outras funções e acabamos por não a conseguir finalizar.

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.

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.
- Os ficheiros *.h* não incluem instruções.
- A não utilização de 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*.

Nota:

Uma ferramenta que usamos bastante e achamos pertinente falar dela foi o debbuger (gdb). Esta ajudou bastante a descobrir onde estavam os erros, e que valores eram passados como inputs em certas funções. Para além disso, também conseguimos observar em que locais os loops estavam a funcionar mal e/ou a nunca acabar devido a erros dos contadores. Esta ferramenta facilitou bastante a elaboração do projeto pela rapidez em que encontramos os bugs poupando-nos bastante tempo.

Observações gerais:

Analisando o projeto, de forma geral, que fizemos nesta primeira fase, estamos contentes na medida em que nos esforçamos e conseguimos fazer duas queries com sucesso e também conseguimos colocar o programa a funcionar. Por outro lado, gostaríamos de melhorar em alguns aspetos.

Em primeiro lugar, cometemos o erro de fazer as nossas próprias hash tables, deveríamos ter usado o Glib visto que as hash tables são mais otimizadas, mais rápidas, têm um custo computacional menor e são mais fáceis de implementar. Não o utilizamos, porque quando nos apercebemos dessa ferramenta, já era demasiado tarde para mudar o código tendo em conta o prazo de entrega da primeira fase. Sendo assim, ficamos com um código que demora 10 a 12 segundos a executar.

Para além disso, teremos de descobrir outra forma mais sofisticada de fazer as queries, visto que não podemos, do ponto de vista de rapidez do programa, fazer uma hash table por cada querie. Teremos de conseguir encontrar um método no qual apenas será necessário três ou quatro hash tables no total. Por fim iremos tentar também tornar o programa mais rápido no geral e não só nas hash tables. Como já tinha referido acredito que o próprio parsing poderá ser feito com funções mais dedicadas a isso para que possamos reduzir o tempo de execução.

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, 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 muitas vezes juntos e esforçamo-nos de forma igual para este projeto. Normalmente distribuíamos tarefas pequenas para cada um fazer e no final juntávamos tudo de forma a termos um progresso mais rápido e melhor. Em suma, acho que fizemos um bom trabalho, tendo em conta o tempo que nos foi dado e o número de pessoas no grupo a trabalhar.

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