Universidade do Minho
Laboratórios de Informática III
Guião II

a96106, Miguel Silva Pinto
a94557, Délio Miguel Lopes Alves
a97613, Pedro Miguel Castilho Martins

1 Introdução

Será desenvolvido um programa, tendo em conta principalmente os conceitos de Encapsulamento e Modularidade.

Este projeto consiste em carregar três ficheiros de dados para memória, para futuramente a informação ser processada, com o objetivo de responder a diversas questões. Sendo divididas as estruturas de dados e as respetivas funções que atuam nessas estruturas , por diversos ficheiros, ficando mais organizado, respeitando as regras de encapsulamento e modularidade.

2 Estrutura de dados

Inicialmente, definimos structs com os correspondentes campos de cada tipo de dados(user, commit, repo).

Para guardar todos os conjuntos de dados provenientes dos ficheiros .csv, definimos três estruturas de dados, diferentes para cada tipo de dados (users, commits, repos), constituídas por um array de apontadores de structs que representam um user/commit/repo, juntamente com um inteiro em que um representa o número de conjuntos de dados inseridos nesse array e um outro inteiro que representa o tamanho a que o array foi alocado em memória, sendo o array realocado para permitir guardar mais valores quando o número de conjuntos de dados inseridos atinge este tamanho máximo.

Escolhemos guardar os dados em arrays, uma vez que é uma estrutura de dados em que nos sentimos mais confortáveis e apresenta uma maior facilidade de acesso aos seus valores, permitindo também uma boa organização dos dados.

De modo a respeitar os conceitos de encapsulamento e modularidade, estas structs foram definidas em diferentes módulos, tendo a sua própria biblioteca de funções que introduzem e acedem aos valores destas estruturas de dados.

3 Parsing

Através das funções que permitem aceder e alterar de forma controlada as estruturas de dados, os dados do ficheiros .csv são transferidos para as diferentes estruturas de dados.

Durante o parsing, é possível obter certos dados úteis para a realização das queries estatísticas, por isso estes valores vão também ser guardados em estruturas de dados para cada tipo de dados, StatsU (stats sobre users), StatsC (estatísticas sobre commits), StatsR (estatísticas sobre repositórios).

Para as StatsU, são guardados o número total de Users que foram inseridos na estrutura, juntamente com o número de utilizadores do tipo Bot, Organization e User. Para as StatsC, são guardados o número total de Commits que foram inseridos na estrutura. Para as StatsR, são guardados o número total de Repos que foram inseridos na estrutura.

Com o objetivo de reduzir o tempo de processamento de dados pelas diferentes queries, após o parsing dos ficheiros .csv que introduziu todos os dados

nestas estruturas, decidimos usar a função qsort da library <stdlib.h>, para ordenar estes conjuntos de dados, relativamente ao valor de um determinado campo.

Foram criadas três funções de comparação de valores para cada tipo de dados que ao serem utilizadas pela função qsort, irão ordenar os catálogos de Users pelo valor do campo "id", os Commits pelo valor do campo "commiter_id" e os Repos pelo valor do campo "id".

Ao organizar os catálogos de dados desta maneira, permitiu-nos obter valores necessários à realização das queries de maneira bastante mais eficaz, o que consequentemente reduz o tempo de execução delas.

No final as funções que realizam o parsing dos três ficheiros, retornarão um tuplo contendo a estrutura com o catálogo do respetivo tipo de dados e uma outra estrutura com os valores das suas estatísticas.

4 Queries

4.1 Query 1

Na query 1, obtemos os valores necessários, através das estatísticas dos Users, sendo simplesmente necessário obter os valores das estrutura de dados **StatsU**, e através das funções de obtenção dos valores do número de bots, users e organizations, fazer print desses valores no determinado ficheiro de output.

4.2 Query 2

Na query 2, precisamos de obter o número de colaboradores e o número total de repositórios. Para saber o número de repositórios, acedemos às estatísticas de **StatsR** e obtemos esse número. Para calcular o número de colaboradores, como organizamos o catálogo de commits relativamente a commiter_id fica mais fácil de conseguir o número total de diferentes commiter_ids, uma vez que os mesmos commiter_ids irão estar em posições contíguas. Obtemos este valor através da função getDifsOfColaboradores que verifica quantos ids diferentes estão envolvidos nos repositórios, ou seja, ids que correspondem ao author de um repositório ou ids de alguém que fez algum commit num repositório.

4.3 Query 3

Na query 3, necessitamos de obter o número de bots colaboradores. Para fazer isso, primeiro criamos um array com os ids de todos os bots no catálogo de users. Com esse array de ids pesquisamos no catálogo de commits se existem ids de committers ou de authors que pertençam ao array de ids dos bots, se encontrarmos um id igual guardamos o id do repositório em que o bot é colaborador num outro array de ids, e impedimos que haja ids repetidos nesse array. A cada novo repositório incrementamos uma variável que será o número de repositórios que contenham um bot como committer ou author (colaborador).

4.4 Query 4

Na query 4, obtemos os valores do número total de commits e do número total de utilizadores, através dos valores guardados nas **StatsC** e **StatsU** respetivamente, e procedemos a calcular a média arredondada a duas casas decimais.

4.5 Query 5

Na query 5, para obter os top N utilizadores mais ativos num determinado intervalo de datas, percorremos o catálogo de commits à procura dos commits feitos dentro das datas recebidas. Utilizando a função **validDate** conseguimos verificar se uma data é superior ou inferior a outra, modificando o valor de **minMax** para 0 ou 1 respetivamente. Com os commits que foram feitos dentro dessa data guardamos os ids do committer num array. Esse array criado irá conter todos os ids dos users que fizeram commits naquele espaço de tempo. Para fazer uma contagem dos top users criamos uma lista ligada que armazena cada id do array e o número de vezes que ele aparece no array. A **struct idx** da query.c foi utilizada para isso. Através da função **howManyx** criamos a lista com o respetivo número de ocorrências. Com a lista criada, ordenamos a mesma através do número de ocorrências de cada id. Com essa lista ordenada temos os Top N users com mais commits dentro de um certo período de tempo e imprimimos no ficheiro os N primeiros dessa lista ordenada.

4.6 Query 6

Na query 6, para obter o top N de utilizadores com mais commits de uma determinada linguagem, começamos por pesquisar o catálogo dos repos e obter um array com todos os ids dos repositórios que usam essa linguagem, através da função **getNRepoLanguage**. De seguida, com a função **getCommiterLanguage** retiramos um array de ids de utilizadores que fizeram commits nestes repositórios. Esta função insere tantas vezes um determinado id de um utilizador no array a devolver, quantas as vezes esse utilizador fez um commit num repositório com essa linguagem (por exemplo: se um utilizador fizer 3 commits num repositório com essa linguagem, a função vai inserir 3 vezes o seu id no array a devolver. [...,id1, id1, id1,...]). Seguidamente o array retornado pela última função vai ser tratado por uma outra função **howManyX** que vai processar os ids do array e retornar uma lista ligada, que contém o id de um utilizador, associado com o número de vezes que fez um commit, ou seja, o número de vezes que aparece no array. Como o catálogo de commits está ordenado, os ids que forem iguais, vão aparecer em posições contíguas do array, sendo menos pesado estar à procura de elementos iguais neste array. Após este processamento, a lista vai ser ordenada por ordem decrescente relativamente ao número de vezes que um utilizador fez um commit. Com esta lista ligada ordenada, apenas é necessário percorrer N (argumento da query) elementos da lista e escrever os seus ids no ficheiro. O respetivo "login" de um id, necessário para escrever no ficheiro, é obtido através da função **returnLoginWithId** que faz uma procura

binária no catálogo de Users, uma vez que estão ordenados, até encontrar o id e obter o "login" associado a esse mesmo id.

4.7 Query 7

Na query 7, para determinar quais os repositórios inativos a partir de uma determinada data, precisamos de ir ao catálogo de commits e identificar os repositórios que não têm nenhum commit para além da data de argumento. Primeiramente, fazemos parsing da data argumento e colocamos os valores correspondentes ao ano, mês e dia em variáveis de inteiros que vão ser usados posteriormente na função **getAboveDate**. Esta função vai retornar um array ordenado de ids de repositórios que têm pelo menos um commit após a data argumento através duma função auxiliar **validDate** que verifica se a data de um commit é maior que a data argumento. Com os valores dos ids que são válidos após esta data, verificamos os ids dos repositórios acedendo aos valores no catálogo de repositórios. Faz-se uma procura binária uma vez que esse array foi ordenado e se um id desse catálogo não se encontrar no array de ids válidos, significa que o repositório com esse id é invalido para aquela data argumento e é adicionado ao array que vai ser devolvido contendo os ids dos repositórios que estão inativos após aquela data argumento. Com este array com os valores de ids de repositórios inativos, só falta percorrê-lo e printar no ficheiro output esses ids, juntamente com a sua descrição que é obtida através da função **returnDescriptionWithId** que faz uma procura binária do id no catálogo de Repos, retornando a respetiva descrição quando encontrado esse id.

4.8 Query 8

Na query 8, para obter o top N de linguagens mais utilizadas a partir de uma determinada data, foi necessário ir ao catálogo de repositórios e verificar quais dos repositórios foram criados após a data recebida como argumento. Primeiramente, fazemos o parsing da data argumento e colocamos os valores correspondentes ao ano, mês e dia em variáveis de inteiros que vão ser usados posteriormente na função **langCounter**. Essa função é responsável por criar um array de strings que irão ser as linguagens que são utilizadas a partir daquela data. É também criado ao mesmo tempo, um array de inteiros que será o número de ocorrências de cada linguagem, e é usado o mesmo índice para identificar a linguagem e o seu número de ocorrências, assim conseguimos saber quantas vezes cada linguagem é utilizada. Com esses 2 arrays criamos uma lista utilizando a **struct ids** que é uma estrutura de dados que guarda uma string e um inteiro. Nessa lista ligada inserimos todas as linguagens e números de ocorrências que temos guardadas nos arrays **words** e **count** utilizando a função **insertL** e depois organizamos essa lista com a função **sortListS**. Agora com a função ordenada pelo número de ocorrências de cada linguagem podemos criar o ficheiro com as N linguagens mais utilizadas a partir daquela data.

4.9 Query 9

Na query 9, para obter o top N de utilizadores com mais commits em repositórios cujo owner é um amigo seu, começamos por criar um array com os ids dos utilizadores com commits em repositórios de amigos. Pesquisamos o catálogo dos commits retirando todos os author_ids (criadores do repositório onde o commit foi feito) e todos os commiter_ids (utilizadores que fizeram o commit). Para verificar se 2 utilizadores são amigos utilizamos a função **isFriend** que verifica se 2 utilizadores são amigos ou não. A função **isFriend** de início verifica se os 2 ids recebidos são iguais e se forem retorna 0, para impedir que seja feito trabalho desnecessário porque vários commits são feitos para repositórios do próprio criador. Se os ids recebidos forem diferentes é feita uma procura binária do catálogo de users pelo índice do utilizador em questão. Com o índice do utilizador encontrado podemos obter o número de **followers** e **following** desse mesmo utilizador. Se o número de followers não for 0 criamos um array com os ids dos seguidores desse utilizador **(follower_list)** e verificamos se o author está nessa lista. Se o author pertencer a essa lista então fazemos o mesmo na **following_list**. Se o author estiver nas duas listas então a função retorna 1 indicando assim que os 2 utilizadores são amigos. Quando encontramos commiters com commits em repositórios de amigos colocamos esses mesmos ids num array. Com o array de todos os ids criamos uma lista usando a **struct idx** e a função **howManyX** para criar uma lista ligada com o número de ocorrências de cada id no array criado pela função **getCommiterFriends** e ordenamos essa lista com a função **sortListX**. Com a lista ordenada pelo número de ocorrências de cada id podemos criar o ficheiro com os N primeiros elementos e os respetivos logins dessa lista.

4.10 Query 10

Na query 10, para obter o top N de utilizadores, pelo tamanho da maior mensagem de commit, em cada repositório, começamos por criar dois arrays, um com o tamanho message e outro com os author_id, dos primeiros N que aparecem no ficheiro commits-g2.csv. Depois, organizamos os dois arrays, por ordem decrescente em relação ao tamanho da message. Em seguida, comparamos o tamanho do último elemento do array com o tamanho das restantes message do ficheiro, caso seja maior, o último elemento do array é substituído e o array é ordenado novamente. No final obtemos um array com author_id ordenado em relação ao tamanho da message, com a função **idsMessage**. É feito o mesmo para obter o repo_id, com a função **repoMessage** e o size, com a função **tamMessage**. Para obter o login fazemos uma pesquisa no ficheiro users-g2.csv com o author_id, com a função **returnLoginWithId**. Porém, a query não responde corretamente ao que é pedido, porque não dá as maiores mensagens de commit por repositório, mas sim as maiores mensagens de commit, repetindo o mesmo repositório, não conseguindo ter implementado 100% corretamente.

5 Conlusão

Durante a interpretação das queries, sentimos alguma dificuldade em ter certezas sobre o que o guião nos pedia, obrigando a certas alterações de estratégias ao longo do desenvolvimento do trabalho.

Em termos de desempenho, as nossas queries, em média, executam em tempo que consideramos aceitável, apercebendo-nos que a query 5 tem uma maior exigência de processamento relativamente ao resto das queries devido à necessidade de comparação das datas de todos os commits com as data recebidas como argumento.

Sobre as estruturas de dados que utilizamos, reparamos que algumas das nossas estratégias podem ser ligeiramente menos eficazes relativamente à utilização de outras eventuais estruturas de dados (ex: Hash Tables, árvores binárias...), mas estamos satisfeitos com a escolha.

Concluindo, este trabalho incentivou-nos a termos uma melhor compreensão sobre detalhes da linguagem C, mais concretamente sobre pointers, gestão de memória e sobre os conceitos de modularidade e encapsulamento, e acreditamos que nos tornou mais aptos a lidar com este tipo de conceitos no futuro.