Relatório - LI3

Grupo 27

David Perreira Alves (A93274)
Rui Miguel Borges Braga (A93228)
Tiago Lucas Alves (A93261)

5 May 2021

Conteúdo

1 Introduction

Neste projeto foi nos proposto criar sistema de gestão e consulta de recomendaçõesde negócios na plataforma Yelp usando a linguagem de programação c.

2 Divisão de Tarefas

2.1 David Perreira Alves (A93274)

Catálogo Reviews;
Query 4,5,9;
Implementação do controlador;

2.2 Rui Miguel Borges Braga (A93228)

Catálogo de Users;
Query 6,7;
Implementação de TABLE;

2.3 Tiago Lucas Alves (A93261)

Catálogo Business;
Query 1,2,3,8;
Implementação do Model(sgr);

3 Estrutura do Trabalho

3.1 Estrutura Geral

Geral

Model - Todas as estruturas de dados e funcionalidades do programa.

Controller - Controla a ligação entre o utilizador e o programa.

View - Construção de Tables para serem apresentadas pelo Controller.

Api dos modulos:

contrHandler.h - Funções que invocam todas as querys do modulo sgr e validam o input do programa

controller.h - Módulo que controla o input do utilizador e comunica entre os modulos para gerar o output esperado.

AuxStructs.h - Módulo no qual existem estruturas de dados auxiliares para otimizar o tempo de execução das querys.

business network.h - Módulo onde se manipula o catalogo de negócios.

business.h - Módulo onde se constroi a estrutura negócio.

catalogoR.h - Módulo onde se manipula o catalogo de reviews.

reviews.h - Módulo onde se constroi a estrutura review.

catalogoU.h - Módulo onde se manipula o catalogo de users.

data.h - Módulo em que se define um tipo data(dia,mês,ano) para ser colocado no modulo das reviews. sgr.h - Módulo em que se define todas querys utilizadas.

glibWarningAvoid.h - Modulo de warnings da biblioteca glib.

interface.h - Módulo de IO que apresenta ao utilizador todo as funcionalidades disponíveis.

table.h - Módulo no qual é implementado a definicão de tables.

3.2 Estruturas de dados para cada modulo

3.2.1 Businesses

Neste modulo foram utilizadas como estruturas de dados Balanced binary trees (da biblioteca glib). A motivação para o uso de àrvores foram as rápidas pesquisas na àrvore dos negócios e fácil de manter e ser alterada e percorrida.

3.2.2 Reviews

Neste modulo foram utilizadas como estruturas de dados hashtables (da biblioteca glib). A motivação para o uso de hash tables foi a inserção e remoção em tempo costante das reviews e a facilidade de implementação das funcionalidades do glib.

3.2.3 Users

Neste modulo foram utilizadas como estruturas de dados hashtables (da biblioteca glib). A motivação para o uso de hash tables foi a inserção e remoção em tempo costante dos users e a facilidade de implementação das funcionalidades do glib.
Quanto à leitura do parâmetro "friends", devido a este não ser utilizado em nenhuma funcionalidade do programa, decidimos armazená-lo numa string, para poupar espaço precioso em memória. Sendo que, caso seja necessário a sua utilização, fizemos uma função que transforma essa string numa lista.

3.3 Estruturas Auxiliares

3.3.1 RV

Estrutura de dados que armazena o business id, o nome do business, número de estrelas total e numero total de reviews.

Intenção de capturar toda a informação miníma necessária para identificar e qualificar um negócio, tentando comprimir ao máximo, de forma a melhor gerir o espaço ocupado em memória.

struct reviewedBUSINESS
char *id;
char *name;
int reviews;
float stars;

3.3.2 RVA

Array estático criado pelos membros do grupo que constroi um array de RV.

3.3.3 RVarray

Array de RV utlizando as funcionalidades do GArray do glib.

3.3.4 StrArray - Array de strings

Array de strings(char*) usando as funcionalidades do GArray do glib.

3.3.5 HASH RV

Hash table de RV utilizando as funcionalidades do Ghashtable do glib.

3.3.6 TREE RV

Binary balanced tree de RV utilizando as funcionalidades do GTree do glib.

4 Implementação Tables

4.1 Implementação em c

```
struct table
    int ls alloc;
    int ls;
    int cs;
    char ***matrix;
    int *sizes;
    int *isNum;
    char **header;
```

4.2 Lógica da Implementação

Para armazenar a informação dentro das tables decidimos simplificar e generalizar, colocando tudo numa matriz de strings. Ou seja, nos casos em que a informação correspondesse a um número, precisávamos de uma componente na estrutura da TABLE que nos informasse disso mesmo, e para isso usámos o parâmetro isNum que é um array de int, com um valor para cada coluna, sinalizando com 1 quando a informação dentro da coluna são números (0 caso contrário).
Os campos ls e cs, dizem respeito às linhas e colunas, respetivamente. O número de colunas numa TABLE é imutável depois do momento da alocação (pelo menos, não fizemos nenhuma função que o alterasse), mas o número de linhas não, ficando, então os campos ls alloc e ls a servir o mesmo efeito que o tamanho e o stack pointer servem numa stack. Na medida em que o ls indica, aonde é que se deve inserir novo conteúdo na TABLE e se o seu valor for igual ao de ls alloc (linhas alocadas), temos que aumentar o espaço alocado.
De resto, temos também um array de strings que contêm um nome descritivo para o conteúdo de cada coluna (**header) e um array de int (*size) para formatar as linhas no momento em que são impressas no terminal.

5 Estratégias Usadas nas Querys

Definição de sgr:
struct sgr
    B NETWORK business;
    catUser users;
    catReview reviews;

5.1 Query 1

Junção de todas as funções que carregam as estruturas de dados dos ficheiros.

5.2 Query 2

A estratégia utilizada foi percorrer a àrvore de businesses e preencher um StrArray com o nome de negócios da letra dada.
Transformação do array de Strings para a table.
Optou-se por esta estratégia, uma vez que, a medida que se percorre a àrvore temos a possibilidade de preencher o uma estrutura de dados auxiliar já com a informação pretendida para a Table em tempo linear(O(n)).

5.3 Query 3

Estratégia utilizada foi percorrer a àrvore de negócios e encontrar o business pretendido.
De seguida, percorrer a hash table das reviews, calcular o número estrelas e número de reviews guardando na struct RV do negócio pretendido.
colocação de toda a informação do negócio numa TABLE.
Optou-se por esta estratégia em que consegue-se a informação do negócio em tempo logarítmico O(log(n)) e em tempo linear o número de estrelas e reviews feitas ao percorrer as reviews para todos os negócios. Aqui vê-se a vantagem da struct RV uma vez que a medida que percorremos as reviews podemos armazenar tanto o número de estrelas como de reviews em paralelo.

5.4 Query 4

Para esta querie era necessário, recebendo um user id, listar todos os negocios avaliados por esse user.Para tal, percorri o catalogo das reviews pois estas contêm tanto o user id como o business id. Para cada review, verifica-se o user id com o que pretendiamos e , caso isto se verificasse, procura-se no catalogo dos negocios o negocio correspondente ao business id daquela review. Optou-se por esta opção ,porque apesar de ter de percorrer todas as reviews (tempo linear), a procura nos businesses faz-se em tempo quase constante o que ajuda muito o tempo de execuçao.

5.5 Query 5

Nesta querie, recebiamos um número de estrelas e uma cidade e tinhamos de verificar quais os negocios dessa cidade com uma média de estrelas nas suas reviews que fosse superior ao número de estrelas fornecidas. Para resolver este problema, eu utilizei uma estrutura auxiliar que armazenava o business id,o número de estrelas total e o número de reviews do negocio. Armazenei esta estrutura numa hashTable em que a chave era o seu business id e inseri-a um valor sempre que a cidade do negocio era a mesma que a dada.Depois percorria esta hashTable e calculava a média de cada valor.Se a média fosse maior que a dada, eu ia buscar esse negócio ao catalogo e inseria a sua informaçao na tabela. Como temos de percorrer o catalogo das reviews, esta função vai correr em tempo linear.

5.6 Query 6

Para determinar os "n"melhores negócios de uma cidade, voltamos a percorrer o catálogo das reviews e, para cada review, transformamos o negócio que lhe corresponde numa estrutura RV. A partir daí, usando a cidade do negócio como chave, criamos uma hash table, cujo valor são árvores que possuem negócios em formato RV como valor e os seus ids como chave. Para cada tratamento da review que vem do catálogo, verificamos se a cidade já se encontra na tabela de hash, caso contrário é adicionada aí e também a uma outra hash table que também possui a cidade como chave, e é criado um array com "n"RV's vazios para facilitar depois no final ao fazer addSort. Nas situações em que a cidade existe na 1ª tabela de hash e o negócio existe na sua árvore, a informação com do RV que se encontra na árvore é complementada com a informação do RV recém criado na iteração atual.
Então, percorre-se a 1ª hash table e para cada árvore, executa-se o addSort de cada RV nos arrays de RV's da outra tabela na respetiva cidade e, no final, percorre-se esta 2ª tabela, adicionado os seus elementos que contém informação à TABLE.

1ª hash table
key: cidade
value: tree key: business id
    value: RV

2ª hash table
key: cidade
value: array de RV's

5.7 Query 7

Para implementar esta funcionalidade no programa, temos que percorrer o catálogo de reviews e, na tentativa de fazer apenas uma vez esta travessia, decidimos fazer duas hash tables auxiliares. Uma que para cada review, adicionava o id do user que a fez como chave e o estado do negócio era adicionado numa lista (valor da hash table) com todos os estados onde este user fez review, se ainda não estivesse na lista. Depois, para cada review, testava-se se o user tinha visitado mais que um estado, se assim fosse, era adicionado o seu id à outra hash table auxiliar (valor não era necessário, por isso ficava a NULL), e caso, não pertencesse a esta 2ª tabela, era também adicionado à TABLE que irá ser retornada no final da querie.

5.8 Query 8

A estratégia foi percorrer a àrvore de businesses e armazenar num array todos os negócios que tem a categoria pedida. Transformação do RVarray para HashRV de modo a ser mais rapido percorrer as reviews para adicionar as estrelas e o numero de reviews ao rv. De seguida, criação de uma àrvore em que o conceito de ordem é dado pelo número de stars. De seguida faz se uma travessia inorder à arvore transformando num RVarray(ordenado pela média de estrelas). Finalmente pega-se nas ultimas n posições pedidas pela query(melhores negócios) e coloca-se numa table.
Optou-se por esta estratégia, uma vez que percorrer a àrvore de business cria-se uma array de RV com todos os negócios da categoria pretendida. De seguida passagem para uma hash de Rv do array em que a key é o business id. Seguidamente, percorrer as reviews(tempo linear), ver se ela existe na HashRV(consulta em tempo constante) e caso exista armazena-se as estrelas e reviews. Após ter todas as RV preenchidas transforma-se a HashRV em arvóre(estratégia utilizada uma vez que algoritmos de ordenação de array mostraram-se muito pouco eficientes neste problema) em tempo linear. De seguida após uma travessia inorder em tempo linear permite ter um array ordenado onde depois se retiram as ultimas n posições em tempo constante.

5.9 Query 9

Esta querie requer que ,dada uma palavra, indique os negocios cujos texto contenha essa palavra. Para isto tinha de percorrer todas as reviews e verificar para cada texto se a palavra existia naquele texto. Para tal, criou-se uma funçao utilizando o auxilio da funçao ispunct() , verificando se a palavra estava separado por pontuaçoes ou espaços das outras palavras. Depois de implementar esta funçao , percorreu-se todo o catálogo das reviews e para cada review verificar se a palavra estava no texto. Apesar de funcional, acho que esta querie era capaz de ser melhorada com uma melhor implementaçao da funçao que verifica se a palavra existe no texto.

6 Testes de performance

Utilizando a biblioteca time.h

Query 1: 8.66s;
Query 1 com friends: 14.19s;
Query 2: 0.08s -> business started by letter(sgr,'A');
Query 3: 0.91s -> business info(sgr, 6iYb2HFDywm3zjuRg0shjw);
Query 4: 0.99s -> business reviewed(sgr, q QQ5KBBwlCc1s4NVK3g);
Query 5: 3.29s -> business with stars and city(sgr,3,Austin);
Query 6: 4.24s -> top business by city(sgr,3);
Query 7: 4.20s -> international users(sgr);
Query 8: 1.31s -> top business with category(sgr,2,food);
Query 9: 3.74 -> reviews with word(sgr, food);

7 Conclusão

Os principais desafios que encontramos neste trabalho estavam relacionados com a gestão de memória, programação a uma escala maior do que estamos habituados, abstração, modularidade e com a escolha dos melhores algoritmos para implementar as queries com a melhor performance possível.
O código não está perfeitamente modular em alguns aspetos e com código reutilizável para as várias querys.
O próximo desafio será no trabalho de java corrigir as falhas feitas neste trabalho de c.

7.1 Aspetos a melhorar:

Para o catalogo de Business usar hash table em vez de binary search tree. As funções do glib não são práticas e a ordenação da àrvore não compensa, pois maior parte das querys só precisam de saber a informação de um negócio, o que seria feito em tempo constante(O(1)) se fosse utilizada uma hash table.

Código mais modular. Separamos as tarefas entre todos e não comunicamos muito para usarmos todos as mesmas estruturas auxiliares fazendo com que haja estruturas auxiliares parecidas entre querys que podiam ser as mesmas.