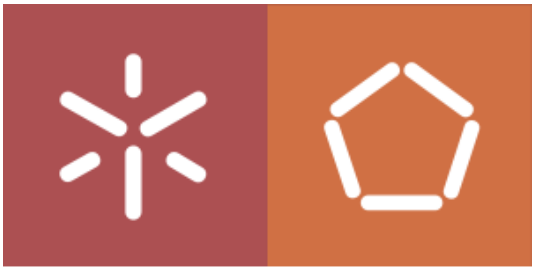


Universidade do Minho
Escola de Engenharia

Sistema de gestão e consulta de recomendações de negócios na plataforma Yelp

Grupo 80

Joana Veiga de Oliveira (A87956)

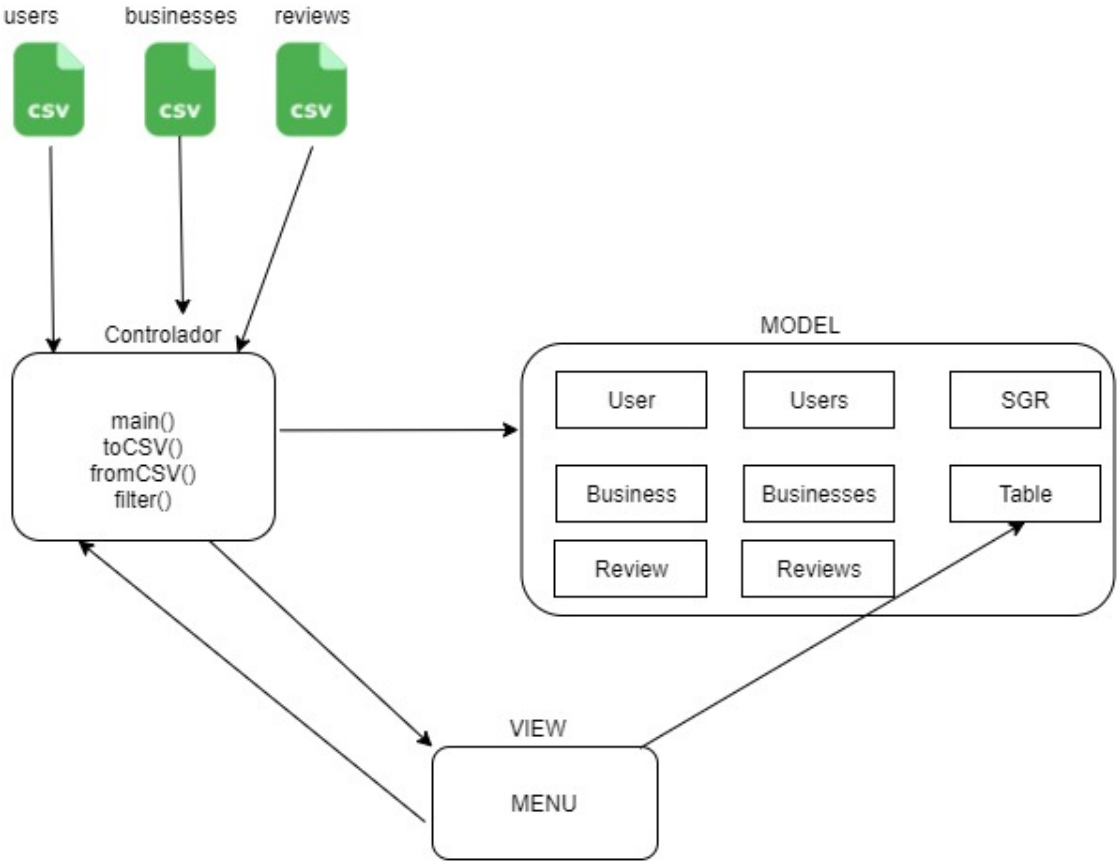
José Miguel Dias Pereira (A89596)

Índice

1. Introdução

No âmbito da unidade curricular de Laboratórios de Informática III, foi-nos proposto desenvolver **um sistema de gestão e consulta de recomendações de negócios na plataforma Yelp**. Este tinha como objetivo trabalhar com grandes volumes de dados, a chamada **programação em grande escala**, onde nos deparamos com conceitos como a modularidade e o encapsulamento.
O maior desafio era conseguir com que a execução do programa fosse o mais rápido possível, uma vez que os ficheiros lidos eram consideravelmente grandes.

2. Arquitetura da aplicação



3. Estruturas de dados

As estruturas de dados escolhidas irão ser faladas mais em detalhe no capítulo a seguir, referente ao *model*. No entanto, é importante referir que, das possíveis escolhas que a biblioteca GLib oferece, escolhemos a HashTable para o armazenamento dos dados, devido à sua eficácia e rapidez.

Pensamos inicialmente na GTree (árvores) mas, após analisar o que as queries pediam, concluímos que ter os dados armazenados por um critério de ordenação não seria assim tão necessário, nem indicado.

Quanto à função de Hash, usou-se a predefinida pela Glib.

4. Model

Esta camada da aplicação é onde se encontra a principal lógica, desde o carregamento dos dados para memória até ao processamento das queries.

A estrutura dos modelos foi evoluindo conforme o desenvolvimento do projeto, de modo a satisfazer as necessidades das queries. Os modelos criados são os seguintes.

4.1. Business

```
struct business{
    char *business_id;
    char *name;
    char *city;
    char* state;
    GHashTable* categories;
    GHashTable* reviews_ids;
};
```

Contém a estrutura básica referente a um negócio como o seu ID, nome, cidade, estado e categorias.

O campo **categories** é uma tabela de hash pelo simples facto de, as funções que a GLib oferece, facilitarem imenso a gestão de memória quando se trata de grande quantidade de dados.

O campo **reviews_id** foi algo que implementámos quando desenvolvemos a query 6. Este é preenchido no momento de leitura dos reviews. Isto é, a cada review que se lê do ficheiro CSV, associamos o seu review_id ao respetivo negócio, para que, quando na query 6 se procure o top dos negócios, apenas tenhamos que percorrer a estrutura dos businesses.

4.2. User

```
struct user {
    char *user_id;
    char *name;
    GHashTable *friends;
    GHashTable *states;
};
```

Para cada utilizador temos o seu ID (user_id) e o seu nome (name) como simples strings.

Para os seus amigos (**friends**), temos uma tabela de hash que é preenchida na leitura do ficheiros dos users. No entanto este campo nunca chega a ser usado em qualquer query.

O campo **states** surgiu no desenvolvimento da query 7 e é preenchido no momento de leitura dos reviews. Para cada review, obtemos o seu negócio (**business**) e, desse business, obtemos o seu estado (**state**) e associamos ao **user** (dono da **review)**. O facto de ser uma hashtable permite que não haja states repetidos e, assim, permite mais tarde obter quantos states únicos cada user visitou de forma praticamente imediata.

4.3. Review

```
struct review {
    char *review_id;
    char *user_id;
    char *business_id;
    float stars;
    int useful;
    int funny;
    int cool;
    DATA date;
    char *text;
};
```

O modelo review, embora seja o que mais campos contém, é o mais simples. Isto deve-se ao facto de não termos nenhum tipo de estrutura complexa, uma vez que, cada campo da review basicamente corresponde a cada campo de cada linha do ficheiro CSV.

4.4. Businesses, Users e Reviews

```
struct businesses{
    GHashTable *businesses;
};
```

```
struct users {
    GHashTable *users;
};
```

```
struct reviews {
    GHashTable *reviews;
};
```

De uma forma muito resumida, estas 3 estruturas são o agrupamento das estruturas acima referidas, em tabelas de hash. A inserção e procura são feitas através das funções disponibilizadas pela GLib.

4.5. SGR

```
struct sgr
{
    USERS users;
    BUSINESSES businesses;
    REVIEWS reviews;
};
```

A estrutura SGR tem como principal objetivo armazenar os dados para depois preencher a TABLE. Assim, a forma mais organizada de representá-la é tendo apenas 3 campos, referentes aos 3 tipos de ficheiros, que estão estruturados segundos os modelos acima referidos.

4.6. Leitura dos ficheiros originais

Na função da leitura de ficheiros é dada a opção de receber como parâmetro a diretoria dos ficheiros. Caso isso não aconteça, são lidas as que estão na diretoria definida estaticamente no ficheiro sgr.c

A leitura dos ficheiros **Business** e ficheiros **Users** é relativamente simples e pode ser descrita da seguinte forma:

1. Abrir ficheiro
2. Ler linha a linha (usando a função fgets)
3. Criar user/business a partir dessa linha (strstep para delimitar)
4. Adicionar user/business criado à hashtable respetiva

A leitura dos **Reviews** segue a mesma lógica mas com a adição dos dois passos referidos acima. Para cada *review*, adicionamos ao respetivo *user* o *state* do *business* do qual a *review* se trata (como auxílio à query 7). Além disso, adiciona-se também o *review_id* ao *business* correspondente (auxílio query 6).

5. View

Módulo responsável por apresentar ao utilizador o resultado das funções invocadas.
É composto por um menu, que contém informação sobre os vários comandos disponíveis e a sua sintaxe, e a função ***show*** do interpretador, que irá ser falada com mais detalhe já em seguida.

5.1. Show

Permite visualizar, sob a forma de tabela, o valor de uma variável. Decidimos, de maneira a uniformizar esta funcionalidade, que todas as variáveis no ***interpretador*** são do tipo TABLE. Assim sendo, e tal como em funções anteriores, também aqui é feita um procura pelos campos preenchidos da estrutura e em seguida, é mostrado o conteúdo em concordância com tal.

5.2. Paginação

Tendo em conta a quantidade de resultados que uma simples query pode gerar, foi necessário apresentar os mesmos de uma forma mais organizada, simples e intuitiva. Por isso, e tal como pedido no enunciado do projeto, este módulo permite mostrá-los sobre a forma de páginas e suporta funcionalidades tais como avançar e recuar de página, bem como voltar para o menu principal.

6. Controllers

Este é o módulo responsável pela interação com o utilizador, ou seja, a nossa **main**.
É, inicialmente, apresentado um menu através da ***view*** e, após isso, é dada a oportunidade ao utilizador de escrever uma sequência de comandos. Estes comandos seguem uma certa sintaxe, definida no enunciado do projeto. São descodificados e passados à parte do ***interpretador*** que contém as funções, ou ao ***model*,** no caso de o comando inserido invocar uma query.

6.1. Interpretador

O módulo interpretador está dividido em duas partes distintas. A primeira, presente na main, que descodifica os comandos passados pelo utilizador e invoca as respetivas funções. E a segunda, onde se encontram essas mesmas funções definidas e que são responsáveis por atender os pedidos do utilizador.

6.1.1. Escrita da Table nos ficheiros (toCSV)

```
void toCSV(TABLE x, char *path, char* delim)
```

A escrita do resultado das queries em ficheiro foi feita de modo a facilitar a leitura dos mesmos, isto é, na chamada à função ***fromCSV***. Na escrita do cabeçalho pomos, em cada campo, um prefixo indicando o número da query. Por exemplo, supondo que se quer escrever em ficheiro o resultado da query 6, onde os seus campos são: City_id, City_name, City_stars. O cabeçalho do ficheiro resultante da escrita ficaria:

6.City_id; 6.City_name; 6.City_stars

Forma que pensamos facilitar e até acelerar a leitura dos mesmos.

6.1.2. Leitura dos ficheiros resultado para Table (fromCSV)

```
TABLE fromCSV(char *path, char* delim)
```

Guarda numa TABLE, o conteúdo de um ficheiro CSV previamente criado com o comando ***toCSV***. Para tal, é lido o cabeçalho e, conforme o número lá presente, é guardada a leitura no campo correto da nossa estrutura.
Exemplo: 2.Business_id; 2.Name, corresponde a preencher o campo correspondente à query 2.

6.1.3. Atribuição do valor de queries

Guarda, numa das posições de um array de 25 TABLEs, o resultado de uma query. O índice na qual este é guardado, depende da sua variável e é calculado através do seu código ASCII menos 97.

Exemplo: 'a' – 0, 'b' – 1, 'c' – 2, etc...

6.1.4. Filtragem de resultados

Filtra dados de uma tabela, dada uma coluna, um valor de comparação e um operador de comparação. Assim, conforme o nome da coluna é selecionado um conjunto de campos da TABLE que o contêm e, após isso, é visto qual deles se encontra preenchido. Só posteriormente é que é feita a filtragem, que, no caso de **ids** e **nomes** é feita de acordo com a **ordem alfabética** dos mesmos.

6.1.5. Indexação

Retorna o conteúdo contido numa dada **linha** e **coluna** da TABLE. Para esse fim são, mais uma vez, percorridos os campos da table e é visto qual deles se encontra com conteúdo. Posto isto, é preenchido na nova TABLE apenas o campo correspondente com a informação contida no índice da linha indicado.

6.1.6. Terminação

Sai do programa com a invocação do comando ***quit;***. Foi, para isso, implementado na *main* um ciclo *do while* que sai da execução quando este é inserido.

7. Testes de performance

7.1. Leitura de ficheiros para memória (query 1)

```
A ler businesses....
Lidos 160585 businesses
Tempo execucao leitura Business.csv: 0.515625 segundos

A ler users....
Lidos 2189457 users
Tempo execucao leitura Users.csv: 31.953125 segundos

A ler reviews....
Lidos 1000000 reviews
Tempo execucao leitura Reviews.csv: 23.312500 segundos
```

7.2. Restantes queries

```
Tempo execucao Query 2: 0.109375 segundos
Tempo execucao Query 3: 0.359375 segundos
Tempo execucao Query 4: 0.500000 segundos
Tempo execucao Query 5: 0.578125 segundos
Tempo execucao Query 6: 0.093750 segundos
Tempo execucao Query 7: 0.906250 segundos
Tempo execucao Query 8: 0.609375 segundos
Tempo execucao Query 9: 4.484375 segundos
```

8. Conclusão

Para finalizar, este projeto permitiu-nos aprender e pôr em prática conceitos como encapsulamento, modularidade, MVC, bem como o uso da biblioteca Glib. Estamos, de um modo geral, satisfeitos com o resultado final. No entanto, e apesar de o nosso programa ser capaz de responder à maior parte dos requisitos, havia, ainda algum trabalho a fazer. Não nos foi possível concluir, por exemplo, a opção de projeção das colunas na parte do interpretador. Além disso, existem também alguns erros de *segmentation fault na* interação com o *menu,* provavelmente devido a *memory leaks,* que gostaríamos de ter resolvido.