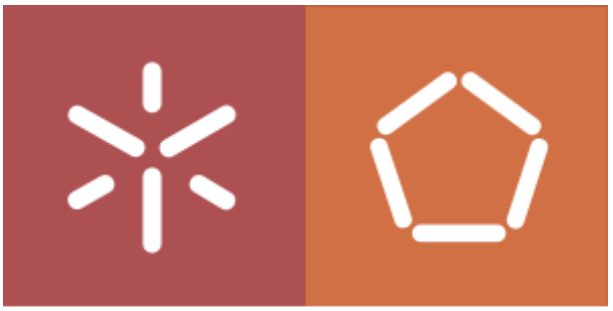

Universidade do Minho

Departamento de Informática

Projeto de Laboratórios de Informática III
Grupo 6

Gonçalo Pereira (a93168) Maria Gomes (a93314)

Gonçalo Afonso (a93178)

18 de Maio de 2021

Introdução

1.1 Motivação

O presente relatório foi elaborado no âmbito da unidade curricular de *Laboratórios de Informática III* (LI3), do Mestrado Integrado em Engenharia Informática da Universidade do Minho e tem como objetivo descrever o desenvolvimento de um sistema de processamento e gestão de informações contidas em ficheiros CSV, utilizando a linguagem de programação C.

1.2 Apresentação do problema

Os ficheiros CSV a serem processados contêm informações referentes à plataforma Yelp, plataforma de críticas e recomendação de estabelecimentos e negócios. É esperada a leitura de grandes quantidades de dados para memória. Além disso, pretendia-se a implementação de um conjunto de queries capazes de extrair informação necessária de uma forma eficiente e em tempo útil para o utilizador, assim como manipulação destes dados através de um interpretador.

1.3 Estratégia implementada

Para a realização deste projeto, após alguma deliberação o grupo concluiu que a solução mais óbvia para guardar os dados dos ficheiros lidos, seria uma hashTable, usando como chave o próprio ID referente à estrutura gravada. Esta implementação, apesar de ter um maior gasto de memória, tem um tempo de procura constante o que poupa muito tempo de execução visto que as queries têm que fazer extensas procuras. Para utilizar hashtables dispusemos da biblioteca glib. Realizamos a leitura dos ficheiros linha a linha usando a função **getline** pois esta aloca dinamicamente o tamanho necessário para cada linha lida. Os dados lidos são validados e adicionados às respectivas hashTables. De forma a responder às queries que foram requisitadas usamos funções disponíveis da biblioteca glib para iterar e procurar, **g_hash_table_foreach** e **g_hash_table_lookup** respectivamente, além disso também usufruímos destas para filtrar dados das hashTables previamente mapeadas com os dados dos ficheiros.

Organização dos dados

2.1 Principais estruturas de dados

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

A estrutura de dados **user** contém o ID do utilizador, o seu nome, e o seu conjunto de amigos identificados pelo ID de utilizador e colocados no campo friends separado pelo delimitador " ; ".

```
struct business{
   char * business id;
   char * name;
   char * city;
   char * state;
   char * categories
};
```

A estrutura de dados **business** contém o ID do negócio, o seu nome, a cidade a que pertence , o seu distrito/estado, e o conjunto de categorias que o caracteriza colocadas no campo categories. Estes campos estão separados pelo delimitador " ; ".

```
struct reviews{
   char * review_id;
   char * user_id;
   char * business id;
   float stars;
   int useful;
   int funny;
   int cool;
   char * date;
   char * text;
};
```

A estrutura de dados **reviews** contém o ID da crítica, o ID do utilizador que a fez, o ID do negócio ao qual a crítica se dirige, número de estrelas atribuídas, o

número de utilizadores que consideram a crítica útil, engraçada ou fixe, o dia em que foi feita e a crítica feita.

```
struct sgr{
   GHashTable * hashT_users;
   GHashTable * hashT_businesses;
   GHashTable * hashT_reviews;
};
```

A estrutura de dados **sgr** é constituída por três HashTables onde em cada uma é guardado o conjunto de dados de utilizadores, negócios e críticas lidos dos ficheiros CSV.

```
struct table{
  char ** tab;
  int entries;
};
```

De forma a tratar os resultados das queries de uma forma forma genérica criamos a estrutura de dados **table** é constituída por uma lista de strings e um indicador do número de entradas da mesma. A informação é guardada linha a linha e os campos de cada linha são separados pelo delimitador " ; ", a primeira linha contém sempre o nome dos campos/colunas guardadas. Desta forma, sabemos sempre a estrutura das strings caso seja necessário a sua visualização (facilita a divisão da string nos seus vários campos, ex.: user_name;user_id;user_friends).

Implementação

2.1 Modularidade

- main.c - main do programa
- Leitura
    - reading.c - funções necessárias à leitura dos ficheiros
- Modelo
    - user.c - funções necessárias à estrutura de dados users
    - business.c- funções necessárias à estrutura de dados business
    - reviews.c- funções necessárias à estrutura de dados reviews
- Visualização
    - show.c - funções que permitem a visualização dos dados pelo utilizador
- Controlo
    - interpretador.c - todas as funções que permitem o funcionamento do interpretador de comandos do programa

- sgr.c - contém as funções necessárias para a estrutura de dados sgr, assim como a implementação das querys
- table.c - contém as funções necessárias para o manuseamento da estrutura de dados table, assim como, implementação de comandos utilizados pelo interpretador
- structs.c - contém funções úteis para o manuseamento da struct GHashTable

2.2 Queries

Query 2: *businesses_started_by_letter(SGR sgr, char letter)*

De forma a solucionar este problema, utilizamos a seguinte struct auxiliar:

```
typedef struct query2 {
    char** result;
    char letter;
    int line;
}*Query2;
```

Nesta struct guardaremos em 'result' os resultados finais da procura, em 'letter' a letra dada como comparador à função, e em line a quantidade total de businesses cujo nome começa pela letra dada.

Percorrendo a GHashTable dos businesses atraves do uso de g_hash_table_foreach com o iterador query2_iterator e user_data uma struct query2, procuramos businesses cujo nome comece pela letra guardada em 'letter' seguindo uma estratégia case insensitive, considerando por exemplo 'm' e 'M' como iguais. Caso um business cumpra esta condição é guardado numa nova linha em result.

Acabando a procura é criado uma nova TABLE cujo char ** t será result e entries será igual a line.

Query 3: *business_info(SGR sgr, char* business_id)*

Estruturas utilizadas neste problema, reutilizadas da query 6:

```
typedef struct query3 {
        GHashTable * h_reviews_info;
 }*Query3;
```

```
typedef struct bstar {
        char* b_id;
        char* b_name;
        int n_reviews;
        float total;
}*B_STARS;
```

Procura na hash table do business da estrutura sgr este negócio com a função *g_hash_table_lookup* a estrutura deste negócio, depois com a hash table das reviews através da *g_hash_table_foreach* tendo como iterador a *reviews3_info* que cria a hash table *h_reviews_info* que tem como key o business id e como valor uma estrutura (*struct bstar*) esta estrutura guarda o número de reviews e o número total de stars de volta à função principal basta calcular o número médio de estrelas, dividindo o número total de estrelas pelo o número de reviews. Toda esta informação é colocada num array através da função *sprintf* e guardada na *TABLE*.

Query 4: *businesses_reviewed(SGR sgr, char* user_id)*

De forma a criar esta query, utilizamos a seguinte struct auxiliar:

```
typedef struct query4 {
  char** result;
  char* user_id;
  GHashTable * hashT_businesses;
  int line;
}*Query4;
```

Nesta struct guardaremos em 'result' os resultados finais da procura, user_id será o user cujas reviews estamos à procura, a GHashTable de business será utilizada pelo iterador auxiliar para encontrar o nome de um business dado o seu id, por fim, line será o número de entradas em result.

Começamos por percorrer a GHashTable * reviews em busca de reviews realizadas pelo user_id dado à query. Para cada elemento desta hash será, através da função auxiliar query4_iterator, verificado se a review contem o user_id procurado, em caso positivo, será buscado o business_id da review e consequentemente procura-se na GHashTable * businesses por este business_id, para obter o nome desse business e posteriormente guardar numa nova linha de result o business_id e o business_name.

Acabando a procura é criado uma nova TABLE cujo char** t será result e entries será igual a line.

Query 5: *businesses_with_stars_and_city(SGR sgr, float stars, char* city)*

Estruturas utilizadas neste problema, reutilizadas da query 6:

```
typedef struct query5 {
        TABLE t;
        char* city;
        float stars;
        GHashTable * hashT_business;
        GHashTable * h_reviews info;
 }*Query5;
```

```
typedef struct bstar {
    char* b_id;
     char* b_name;
    int n_reviews;
     float total;
  }*B_STARS;
```

Primeiramente, inicializa a estrutura *struct query5*, ou seja, a *TABLE* onde será guardada a resposta, de seguida torna a city em letras minúsculas e guarda-se, coloca-se a hashtable do business da sgr e guarda o número de stars dado. Com *a g_hash_table_foreach* aplicada à hashtable das reviews tendo como iterador a *reviews5_info* que cria a hash table *h_reviews_info* que tem como key o business id e como valor uma estrutura (*struct bstar*) e esta estrutura guarda o número de reviews e o número total de stars de cada negócio.

Por fim, com a função *g_hash_table_foreach* aplicada à hashtable das *h_review_info* tendo como iterador a função *query5_iterator* que calcula o número médio de stars e depois compara esse número como o número de stars da estrutura *struct query5* se for maior ou igual vai procurar na hashtable do business este business id e obtém a city com a get_city e torna-a em letras minúsculas e verifica se é igual à dada. Sendo igual é colocada na TABLE o seu business id e o seu name com a função *setNewLine*.

Query 6: *top_business_by_city(SGR sgr, int top)*

Para esta questão utilizamos as seguintes structs:

```
typedef struct city {
    char* name;
    char** top;
    int entries;
    float low_score;
}*CITY;
```

```
typedef struct b_stars{
    char* b_id;
    char* b_name;
    int n_reviews;
    float total;
}*B_STARS;
```

```
typedef struct b_average_stars{
    SGR sgr;
    GHashTable * b_same;
    GHashTable * cities;
    int top;
    char* condition;
    char** result;
    int entries;
    float lowScore;
}*B_AVERAGE_STARS;
```

A struct principal para esta query será B_AVERAGE_STARS que por sua vez gera uma GHashTable de elementos CITY, que servirá para distinguir as diferentes cidades no documento dos business, unindo cidades com mesmo nome na mesma chave, e guardando nesta struct os 'top' businesses de cada cidade, e outra GHashTable de elementos B_STARS, onde cada chave será um business_id, e lá se guardará o número de reviews festa neste negócio e o total acumulado de estrelas dados.

Na resolução primeiro pretendeu-se criar a GHashTable das cidades percorrendo a hash table dos businesses utilizando como iterador a função auxiliar city_hash, de modo a criar o ambiente onde guardar o top para cada cidade, e, ao mesmo tempo descobrir o numero de tops maximo que serao necessarios calcular.

De seguida percorre-se a hashtable das reviews de modo a criar a GHashTable de structs B_STARS, registando para cada business o número de reviews dele feito e o acumular das estrelas dessas reviews.

Neste ponto, tendo feito uma tabela para guardar o top de cada cidade e calculado o número médio de estrelas de cada business, percorre-se novamente a hash table dos businesses, verificando se para a key (business_id) existe um B_STAR correspondente (foram feitas reviews), e de seguida utilizando o algoritmo de função top_city, atualiza-se o top da cidade desse business.

Por último, é criado uma nova TABLE com o número de linhas igual ao número de cidades vezes o 'top' e através de uma passagem pela hashtable de CITY, é processado o 'char **top' de cada CITY para a TABLE, ordenado pela função auxiliar sort_top(), que utiliza um algoritmo adaptado do quicksort para ordenar segundo as 'stars' de cada linha, de forma crescente.

Query 7: *international_users (SGR sgr)*

Estrutura utilizada neste problema:

```
typedef struct query7 {
    TABLE t;
    GHahTable * h_user_and_businesses;
    GHashTable * hashT_business;
    GHashTable * h_state;
}*Query7;
```

Inicializa a estrutura *struct query7*. Com *a g_hash_table_foreach* aplicada à hashtable das *reviews da sgr* tendo como iterador a função *query7_iterator* que vai inserir na *h_user_and_businesses* como *key o user_id da review* e associado a este uma lista com todos business id que este user fez review. Depois percorre a *h_user_and_businesses* com a *g_hash_table_foreach* com a função *check_state,* que verifica se a lista associada a cada user é maior que dois e se for depois vai-se verificar se são de estados diferentes procurando na

hashT_business com a função *g_hash_table_lookup,* nos casos de ser diferente é colocada na *TABLE* o *user_id.*

Query 8: *top_business_by_city(SGR sgr,int top, char* category)*

Em muitos aspectos esta query é similar à query 6 tendo portanto reutilizado para esta as structs:

```
typedef struct b_stars{
    char* b_id;
    char* b_name;
    int n_reviews;
    float total;
}*B_STARS;
```

```
typedef struct b_average_stars{
    SGR sgr;
    GHashTable * b_same;
    GHashTable * cities;
     int top;
    char* condition;
  char** result;
    int entries;
      float lowScore;
  }*B_AVERAGE_STARS;
```

No caso da query 8 utilizamos 'char** result' para guardar os top business.

Similarmente à query 6, cria-se uma hashtable de B_STARS que contém o número de estrelas médio de cada business, porém neste caso, apenas daqueles que validem a condição da categoria passada à query, guardada em 'char* condition', para fazer esta verificação utiliza-se a função auxiliar cmp_category, que segue uma estratégia case insensitive.

Em seguida percorrer-se-á esta hashtable com o iterador top_category que, com um algoritmo similar ao de top_city (auxiliar da query 6), irá atualizar 'char** result'.

Por fim cria-se uma struct TABLE cujo número de entradas será igual a 'top' e 'char** t' será um char** com a mesma informação de 'char** result', porém passada pelo sort_top() (reutilizada da query 6), que ordena top de acordo de forma crescente.

Query 9: *reviews_with_word (SGR sgr,char * word)*

Inicializa uma "struct query9" que contém uma TABLE e uma string com a palavra dada como argumento.Posteriormente a hashTable de reviews é percorrido com recurso à função *g_hash_table_foreach*, que para todos os valores da hashTable vai executar a função *query9_iterator* que para uma dada struct review percorre o campo "text" da mesma e procura pela ocorrência da palavra word.Caso essa ocorrência se verifique o "review_id " correspondente é adicionado à TABLE.

2.3 Interpretador de comandos

```
typedef struct variavel {
    TABLE t;
    char* variavel;
}*VARIAVEL;
```

```
typedef struct variaveis {
        VARIAVEL* variaveis;
    int entries;
    int max;
}*VARIAVEIS;
```

Comando: atribuição do valor das querys a uma váriavel

Para podermos associar o valor das querys a variáveis criamos a estrutura *struct variveis* que permite guardar várias atribuições como por exemplo: *x = business_info(sgr); u = international_users(sgr);* Antes de ser guardada uma variável é verificado se já existe alguma variável com o mesmo nome, nesse caso, a variável é atualizada. Para não limitar o número de variáveis guardadas usamos os inteiros entries e max como controladores, por exemplo se se quiser inserir uma nova variável e se entries for igual a max é feito o realloc e o max aumenta para o dobro.

Comando: *toCSV (TABLE t,char* delim, char* filepath)*

Temos um ciclo que percorre todas as linhas da TABLE e nesta a informação está separada ponto e vírgula, para ler cada linha temos um ciclo que usa esse separador na função strsep e com a função fprintf escrevemos no ficheiro esse primeiro pedaço depois verificamos se é o último e se não for escrevemos no ficheiro o delimitado delim.

Comando: *fromCSV (filepath,delim)*

Primeiramente, se não conseguirmos abrir o ficheiro dado devolvemos uma table vazia, caso contrário vamos contar quantas linhas tem o ficheiro e criamos uma table com esse número de entradas. Para o tratamento do ficheiro temos uma função, getLine_file, a quem é passada o ficheiro. Esta lê apenas uma linha e escreve-a para um array que se tiver um tamanho insuficiente é aumentado de forma dinâmica. Previamente a ser colocada a linha na table é testada no array aux para ver se tem espaço suficiente para colocar o nosso delimitador. A divisão da linha é feita com o strsep e colocada na aux com o strcat. Para os casos em que o último pedaço a ser escrito ou o delimitar delim é uma string é necessário analisar se se coloca o nosso delimitador ou não .

Todos os outros comandos seguem implementações semelhantes.

Tempos de execução e conclusão

Os seguintes testes foram realizados utilizando os input files fornecidos pela equipa docente de LI3, numa máquina com as seguintes especificações:

Laptop;
Memória RAM: 16 Gb;
Processador: Intel i5-9300H CPU 2.4GHz Quad-core;
Sistema Operativo: Windows (utilizando ubuntu da microsoft store);

Query 1: 11.5 - 11.7 segundos (s)

Para os próximos testes será descontado o tempo de execução de processos auxiliares, como o load_sgr.

Query 2:
TABLE t = businesses_started_by_letter(sgr,'M'); 0.03125 s
TABLE t = businesses_started_by_letter(sgr,'m'); 0.03125 s
TABLE t = businesses_started_by_letter(sgr,'A'); 0.03125 s

Query 3:
(4 testes)TABLE t = business_info(sgr, "-l5w8_vwKDSUlpr9FSQoqA"); 0.359375 - 0.406250 s
(4 testes)TABLE t = business_info(sgr, "6fT0lYr_UgWSCZs_w1PBTQ"); 0.359375 s

Query 4:
(4testes)TABLE t = businesses_reviewed(sgr, "RNm_RWkcd02Li2mKPRe7Eg"); 0.234375-0.25s
(4testes)TABLE t = businesses_reviewed(sgr, "bUHweiErUJ36WGeNrPmEbA"); 0.234375-0.25s

Query 5:
(4 testes)TABLE t = businesses_with_stars_and_city(sgr,1.0,"Portland"); 0.375-0.390625 s
(4 testes)TABLE t = businesses_with_stars_and_city(sgr,4.0,"Austin"); 0.359375-0.5 s

Query 6:
(4 testes)TABLE t = top_businesses_by_city(sgr,1); 0.734375-0.765625 s
(4 testes)TABLE t = top_businesses_by_city(sgr,10); 0.75-0.78125 s
(4 testes)TABLE t = top_businesses_by_city(sgr,100); 0.875-0.906250 s

Query 7:
(4 testes)TABLE t = international_users(sgr); 1.296875-1.42187 s

Query 8:
(4 testes)TABLE t = top_businesses_with_category(sgr,1,"Chinese"); 0.609375-0.65625 s
(4 testes)TABLE t = top_businesses_with_category(sgr,10,"Chinese"); 0.625 s
(4 testes)TABLE t = top_businesses_with_category(sgr,100,"Chinese"); 0.625-0.703125 s

Query 9:
(4 testes)TABLE t = reviews_with_word(sgr,"seem"); 3.109375-3.140625 s
(4 testes)TABLE t = reviews_with_word(sgr,"is"); 2.203125-2.234375 s

Conclusão

Com a realização deste trabalho ficamos a entender como funciona o processamento de grandes volumes de dados. Conseguimos realizar todas as tarefas que nos foram propostas. Apesar disso, havia ainda algumas melhorias possíveis que gostaríamos de ter feito, tal como funções mais modulares e testes de performance. Quanto às nossas estruturas de dados são HashTable, o que permite procurar e inserir elementos com uma complexidade de O(1), o que é bastante bom tendo em conta outras como por exemplo árvores binárias.