



UNIVERSIDADE DO MINHO

Mestrado Integrado em Engenharia Informática

Laboratório de Informática III – Projeto C

Grupo 64

Ana Filipa Ribeiro Murta (a93284)

Ana Paula Oliveira Henriques (a93268)

Augusto César Oliveira Campos (a93320)

Ano Letivo 2020/2021

Índice

1. Introdução	3
2. Módulos	3
2.1 Módulo <i>dados</i>	3
2.2 Módulo <i>interpretador</i>	4
2.3 Módulo <i>auxiliares</i>	5
2.4 Módulo <i>paginacao</i>	6
2.5 Módulo <i>sgr</i>	7
3. Conclusão	10

I. Introdução

Neste trabalho, foi-nos proposto a criação de um programa de reviews de negócios que lida com um grande número de dados. Posteriormente, tivemos de executar queries que se relacionam com as funcionalidades esperadas de um programa semelhante ao que nos foi proposto; entre elas, estacam-se procurar um só negócio, procurar todos os negócios começados por uma letra, entre outras. Por isso, tivemos de arquitetar um programa que fosse bastante eficiente na procura de dados de modo a que, se um utilizador necessitasse de usar estas funcionalidades, a aplicação fosse “responsive” e agradável de utilizar.

2. Módulos

2.1. Módulo *dados*

As estruturas de dados representadas em baixo são referentes aos tipos de dados dos ficheiros.csv (users.csv, business.csv, reviews.csv) disponibilizados no BB.

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

```
typedef struct business{
    char *business_id;
    char *name;
    char *city;
    char *state;
    char **categories;
}*BUSINESS;
```

```
typedef struct user{
    char *id;
    char *name;
    char *friends;
}*USER;
```

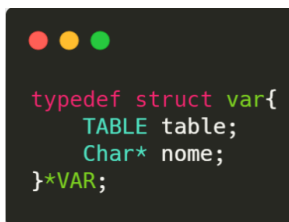
Relativamente à estrutura de dados *REVIEW*, os campos *date* e *text* foram guardados como strings para ser mais fácil fazer o tratamento dos dados. Mais especificamente sobre o campo *date*, nós tínhamos conhecimento sobre a possibilidade de criarmos o nosso próprio tipo de dados *Date* (ou, então, usarmos a biblioteca `<time.h>`), mas, por questões de tempo e de poupança de memória, preferimos investir noutras partes do trabalho. Em relação à estrutura de dados *BUSINESS*, o campo *categories* guardámos como um array de strings e assumimos que todas as categorias encontradas eram válidas, incluindo a falta delas. Já na estrutura de dados *USER*, guardámos todos os campos como strings.

Neste módulo, implementámos toda a API que modifica e cria as estruturas de dados. Uma das funções que faz parte desta API é a *lerFichCsv*, que faz a leitura do ficheiro e retorna a matriz dinâmica de char's. Ora, nesta matriz, cada linha corresponde a uma linha do ficheiro e, como tal, a *lerFichCsv* recebe o input dos ficheiros transformados em strings de forma a que estas sejam usadas na criação das structs.

Para converter strings num apontador *BUSINESS*, *REVIEW* e *USER*, temos as funções *addBusiness*, *addReview* e *addUser*, respetivamente. Em cada uma, é alocado espaço dinamicamente para a respetiva struct e são criados todos os seus campos.

As funções *transStrTo* usam o array de strings criado pela função *lerFichCsv* e, a partir deste, aplicam as funções *add* a cada string do array, que corresponde a uma linha do ficheiro. Deste modo, é criado um array dinâmico de apontadores para as respetivas structs. Posteriormente, este array será usado nas funções *transStructToTable* para a criação das hash tables, facilitando o processamento das query's.

2.2. Módulo *interpretador*



```
typedef struct var{
    TABLE table;
    Char* nome;
}*VAR;
```

Esta estrutura de dados foi criada como auxílio na leitura de uma variável, dada pelo utilizador, para se poder guardar o nome da mesma e associá-la a uma *TABLE*. Assim, sempre que for utilizada esta variável na execução de outra função, saberemos a *TABLE* que foi criada pelo utilizador.

A função *show* permite visualizar uma *TABLE* e ainda possibilita ao utilizador saber em que página se encontra e o número total de páginas. A estratégia foi usar um ciclo que termina quando o usuário assim o pretender, uma vez que o caso de paragem é determinado pela variável *r*, terminado quando esta tiver o valor 1 e continuando a visualizar páginas quando tiver o valor 0. Para além disto, a *show* também usa a função *printPagina* para imprimir as páginas.

Na construção da *filter*, decidimos recorrer à função auxiliar *compare* para podermos comparar o conteúdo de duas strings. Consoante o operador dado como argumento, a *compare* retorna 0 se a comparação entre os dois valores verificar a condição do operador. Assim, a *filter* verifica uma linha de cada vez se a *compare* tem sucesso e, se tal acontecer, é guardada essa linha numa nova *TABLE*, que será a retornada pela dita função.

Já a *maxOrMin*, um comando adicional que fizemos, imprime o extremo de uma determinada coluna dependendo do operador dado como argumento. Se o operador for "LT", imprime o mínimo dessa coluna; se for "GT", imprime o máximo.

Neste módulo, a função principal é a *interpretador* uma vez que é a partir desta que executamos o que o utilizador pedir. Ao fazer esta função, deparámo-nos com um problema: como iríamos guardar os nomes das variáveis dadas pelo usuário; isto porque, são estas variáveis que guardam os resultados das query's. Assim, sempre que o utilizador quisesse executar algum comando, dando como input uma das variáveis inicializadas anteriormente, seria necessário aceder à *TABLE* associada a essa variável. A estratégia para resolver isto foi, portanto, o uso da função *regex*, que faz o parsing da string dada como input, ou seja, que guarda, num array dinâmico de strings, o que o

usuário pretende executar. Por exemplo, se for `x = businesses_started_by_letter(sgr, 'A')`, então iremos guardar no array o `"x"`, `"="`, `"businesses_started_by_letter"`, `"sgr"` e `"A"`. Deste modo, iremos guardar o nome da variável, `"x"`, e associá-la à *TABLE* produzida. Para além disto, a *regex* também permitiu, como já dito previamente, guardar o nome da função que o usuário chamou, bem como os seus parâmetros. Nesta função, outra estratégia utilizada para correr estes comandos que utilizam *TABLE's* já criadas foi verificar se a variável já existia, ou seja, se o nome da variável que o utilizador passou como parâmetro para esse comando era válido ou não. No entanto, após a data de entrega, enquanto revíamos o trabalho, apercebemo-nos de que testávamos se a variável já existia para os comandos deste módulo (*show*, *filter*, *proj*, entre outros), mas não o fazíamos para as queries. Portanto, o que está a acontecer é que estão a ser guardadas, no array das variáveis, variáveis com o mesmo nome, porém associadas a *TABLE's* diferentes.

2.3. Módulo *auxiliaries*

Este módulo foi criado para definir funções auxiliares que não estavam diretamente ligadas com o objetivo do projeto, mas que simplificavam a construção de funções importantes e que, em situações de corrigir erros, facilitavam a reescrita do código. Estas funções acabaram por ser usadas em mais do que um módulo, daí estarem num ficheiro à parte. Por exemplo, a função *compare* foi uma grande ajuda sempre que quisemos comparar o conteúdo de duas strings e, também, possibilitou que o código da função *filter* não ficasse tão "pesado", ou seja, difícil de ler. Outro exemplo é a *ordenaDecresc*, que foi chamada em duas queries, permitindo, deste modo, que não houvesse a repetição de código em duas funções distintas.

No decorrer do nosso projeto, sentimos a necessidade de utilizar *regular expressions (regex)* para fazer o parsing de strings, usando um padrão e criando todos os *tokens* que coincidissem com esse padrão. Estes tokens foram guardados num array dinâmico de strings. Para este efeito, implementámos a *regex* da biblioteca *glib*, construindo, então, duas funções: uma que cria apenas os *tokens* que fazem match com o padrão e outra que cria os *tokens* que fazem match sem repetição; isto não esquecendo que um *token* aparece uma e uma só vez.

2.4. Módulo *paginacao*

```
typedef struct table{
    char ***variaveis;
    int numLin;
    int numLinTotal;
}*TABLE;
```

Neste módulo, definimos a estrutura de dados TABLE com três campos. O campo *variaveis* é um array de array de strings, isto é, é uma matriz, que guarda o conteúdo da TABLE por linhas e colunas de maneira a facilitar a consulta de um valor numa coluna, numa determinada linha. Já os campos *numLin* e *numLinTotal* são inteiros que guardam o número da linha atual (para a visualização) e o número de linhas total, respetivamente. Estes dois campos foram criados para podermos imprimir uma página, avançar ou recuar de página e, ainda, definir quantas linhas uma página tem.

Como referido anteriormente, neste módulo uma das suas funcionalidades é avançar e recuar de página, sendo que isto é nos possibilitado pela função *acao*.

Para este efeito, a estratégia utilizada foi a leitura de uma específica tecla dada pelo utilizador. As teclas às quais foram atribuídas funcionalidades são: a “k”, para avançar de página, a “j” para recuar de página e a “q” para parar de visualizar páginas. Para isto, uma vez que uma página tem, no máximo, 10 linhas, a tática foi mexer com o número de linha atual. Isto é, se o utilizador procurar avançar de página, então incrementamos 10 à linha atual e, deste modo, estamos a colocar a linha atual na primeira linha da página a seguir. No entanto, caso a página atual não contenha 10 linhas, a *acao* não irá alterar o número da linha atual. Se quiser recuar de página, então é decrementado 10 à linha atual colocando, assim, a linha atual a apontar para a primeira linha da página a que queremos recuar. Há que realçar que, no entanto, se a página atual for a primeira, o comando “j” não efetua nenhuma ação. A função retornará 0 se o utilizador ainda pretender visualizar mais páginas e 1 caso contrário.

Por questões de modularidade, as funções *load_table*, *init_table*, *init_linha*, *add_linha*, *add_palavra* foram criadas para permitir o acesso as TABLES pelos outros módulos e para facilitar a criação das mesmas. Como podemos ver pela função *proj*, que se localiza no interpretador, para podermos criar a TABLE pedida, utilizamos estas funções para inicializarmos uma TABLE e, à medida que a função é executada, criar linhas com os valores pretendidos.

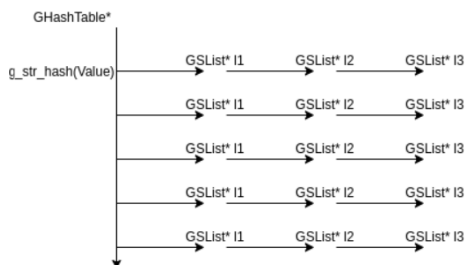
2.5. Módulo *sgr*

```
typedef struct sgr{
    BUSINESS *bus;
    REVIEW *rev;
    USER *use;
    GHashTable* business;
    GHashTable* businessByCity;
    GHashTable* businessByInicial;
    GHashTable* businessByCategory;
    GHashTable* review;
    GHashTable* reviewByText;
    GHashTable* reviewBusId;
    GHashTable* reviewUserId;
    GHashTable* user;
}*SGR;
```

A estrutura de dados SGR foi criada à medida que se foram construindo as query's, isto é, consoante houvesse a necessidade de criar campos que facilitassem a procura de dados específicos relativos às structs *BUSINESS*, *REVIEW* e *USER*, mudando entre elas as keys que se associam ao apontador das structs.

Por exemplo, a *businessByCity* é uma hashtable em que cada key é uma cidade e cada value é uma lista ligada de *BUSINESS* (pointer de business). Se procurarmos uma cidade nesta table, irá ser retornado a lista ligada de negócios da cidade em questão.

Este é o diagrama de uma hash tables:



QUERY 1

```
STARTED LOADING SGR...
LOADING Table Business...
LOADING Table Reviews by User...
FINISHED Table Business.

LOADING Table Business por Cidade...
FINISHED Table Business por Cidade.

LOADING Table Business por letra inicial...
FINISHED Table Business por letra inicial.

LOADING Table Business por categoria...
FINISHED Table Business por categoria.

FINISHED Table Business by User.

LOADING Table Reviews by Business...
FINISHED Table Business by Business.

Tempo de execucao da QUERY (segundos): 3.38804
FINISHED LOADING SGR!
```

Nesta query, criámos a todas as hash tables necessárias para a execução do programa exceto a da query 9, pois se o tamanho do input for grande, o load time do programa tornar-se muito longo. A criação da hash table da query 9 leva cerca de 45 segundos e a criação das hash tables de todas as outras queries leva cerca de 5 segundos, sendo estes valores bastante destoantes.

QUERY 2

```
# > x = businesses_started_by_letter(sgr,'a');
Tempo de execucao da QUERY (segundos): 0.012353
```

Para construir esta query, começámos por procurar a lista com todos os negócios iniciados pela letra dada como argumento, na hashtable *businessByInicial*. A seguir, caso exista, é calculado o número de negócios começados por essa letra e é, também, construído o cabeçalho da *TABLE* inicializada e retornada por esta query. Por último, esses negócios são iterados um por um de forma a escrever, nessa *TABLE*, a identificação e o nome de cada um deles.

QUERY 3

```
# > e = business_info(sgr,"pCFTvC1v0B9Wilm8XixKDw");  
Tempo de execucao da QUERY (segundos): 2.8e-05
```

Nesta query, criámos uma lista com o negócio identificado pelo *bus_id* dado

como parâmetro, guardando este negócio numa variável do tipo *BUSINESS*. Em seguida, procurámos esse *bus_id* na hashtable *reviewByBusId* de forma a podermos calcular o número médio de stars de todas as reviews feitas a esse negócio. Construímos, então, uma *TABLE* com uma linha para o cabeçalho e outra linha com o nome, a cidade, o state, o número médio de stars e o número de reviews desse negócio.

QUERY 4

```
# > y = businesses_reviewed(sgr,"vQ8XF1c495SYzKVQyuTDFA");  
Tempo de execucao da QUERY (segundos): 1.1e-05
```

Com o objetivo de determinar a lista de negócios aos quais um

determinado user fez reviews, verificámos, primeiramente, se esse *user_id* existe na hashtable *reviewByUserId*. Se existir, cria-se, então, uma lista com todas as reviews feitas por esse user e, ainda, é inicializada uma *TABLE*. Assim, à medida que as reviews são iteradas, é criada outra lista com o negócio associado a essa review, guardando-se a identificação e o nome desse negócio na tal *TABLE*.

QUERY 5

```
y = businesses_with_stars_and_city(sgr,4.5,"Columbus");  
Tempo de execucao da QUERY (segundos): 0.010661
```

Pretende-se produzir uma *TABLE* com a lista de negócios de uma cidade

com *n* ou mais stars. Para este efeito, procurámos essa cidade na hashtable *businessByCity*. Assim que encontrada, são iterados todos os negócios dessa cidade e, posteriormente, é calculado o número médio de stars de cada um desses negócios. Se este valor médio for maior que o número de stars passado como argumento, então produz-se uma *TABLE* onde será adicionada uma linha com a identificação, o nome e o número médio de stars desse negócio que verifica aquela condição.

QUERY 6

```
table = top_businesses_by_city(sgr, 10);
```

```
Tempo de execucao da QUERY (segundos): 16.1959
```

Para construir esta query, criámos uma lista com todas as cidades que existem.

Deste modo, percorre-se uma cidade de cada vez e, para cada uma, forma-se outra lista com todos os seus negócios. A seguir, calculámos o número médio de stars de cada um deles e, para tal, procurámos os seus *bus_id*'s na hashtable *reviewsByBusId*. Entretanto, como se pretende retornar uma *TABLE* com os top n negócios de cada cidade, criámos um array auxiliar para o qual fomos realocando espaço dinamicamente à medida que se iteravam esses negócios. Nesse array, guardou-se as identificações, os nomes e os números médios de stars de todos os negócios de cada cidade. No final, este array é ordenado decrescentemente para se poder escrever os top primeiros negócios desse array numa *TABLE*. Assim que escritos, passa-se para a próxima cidade e o array auxiliar volta a ser inicializado a NULL.

QUERY 7

```
# > x = international_users(sgr);
```

```
Tempo de execucao da QUERY (segundos): 0.581854
```

Nesta query, começámos por obter a lista de listas das reviews organizadas por users. Para isso,

obtemos a lista de values da table *reviewByUserId* e percorremos essa lista de listas de reviews. Para cada lista de reviews, verificamos se os businesses que foram reviewed são de estados diferentes; se forem, adicionamos o user a table de output da query.

QUERY 8

```
# > e = top_businesses_with_category(sgr, 10, "Food");
```

```
Tempo de execucao da QUERY (segundos): 35.8473
```

Nesta query, criámos uma lista com todos os negócios com a

categoria dada como argumento. Assim, percorre-se um negócio de cada vez e, para cada um, forma-se outra lista com todas as suas reviews. Para tal, procurámos o seu *bus_id* na hashtable *reviewsByBusId* e calculámos o número médio de stars do negócio. Como se pretende retornar uma *TABLE* com os top n negócios que pertencem a uma determinada categoria, criámos um array auxiliar para o qual fomos realocando espaço dinamicamente à medida que se iteravam esses negócios. Nesse array, guardou-se as identificações, os nomes e os números médios de stars de todos os negócios da categoria dada. Por fim, este array é ordenado decrescentemente para se poder escrever os top primeiros negócios desse array numa *TABLE*.

QUERY 9

Nesta query, usámos uma hash table em que cada key é uma palavra que está no campo text de uma review e o value é a lista de reviews que tem a palavra, depois simplesmente procuramos por essa palavra na hash table e criámos uma *TABLE* de visualização com a lista obtida.

```
# > b = reviews with word(sgr, 10, "bad");  
LOADING Table Reviews by Words...  
This may take a while but only occurs once  
  
FINISHED Table Reviews by Words!!!  
  
Tempo de execucao da QUERY (segundos): 48.1405  
  
# > c = reviews_with_word(sgr, 10, "bad");  
  
Tempo de execucao da QUERY (segundos): 0.032861
```

3. Conclusão

Este projeto foi bastante enriquecedor para todos os elementos do grupo. Isto porque nunca nenhum de nós tinha prestado especial atenção ao quão exponenciais certos trechos de código podem ser. Por vezes, escrevemos “lazy code” que acaba por passar despercebido pelos nossos olhos, mas a verdade é que esse código não é “scalable”. Com este trabalho, isso foi bastante óbvio na execução das queries: algo que, se fosse feito uma vez, levava 0.25 segundos e não era notado, passava a demorar segundos ou minutos, o que não era aceitável para a implementação que queríamos fazer para a proposta dos professores.

Para além disso, o trabalho com a biblioteca da GLIB, bem como o esforço constante para a modularidade do nosso próprio código, ajudou-nos a compreender melhor o quão importante é o nosso código ser entendido, não apenas por nós, mas também por todos os colegas da área que possam ter de utilizar o nosso código sem ter de perceber nada sobre os in’s and out’s do mesmo.