Projeto de Laboratórios de Informática 3
Grupo 21

Diogo Braga A82547 João Silva A82005
Ricardo Caçador A81064

3 de Maio de 2018

Resumo

Este documento apresenta o projeto de Laboratórios de Informática 3 (LI3), do curso de Engenharia Informática da Universidade do Minho.

O projeto baseia-se na criação de um sistema de análise de ficheiros XML que possuem informações do Stack Overflow, um website de perguntas e respostas sobre programação de computadores.

Conteúdo

Lista de Figuras

1 Introdução

Este trabalho tem por base o parse de elementos de ficheiros XML relacionados com a informação do website Stack Overflow, de forma a seguidamente responder a uma série de questões relacionadas com posts, utilizadores e tags do mesmo website. Aliado a tal está o desafio de procurar sempre o melhor algoritmo de resolução das queries, de forma a tornar o código mais eficiente e o mais rápido possível.

A Secção 2 apresenta as estruturas de dados utilizadas no projeto, a Secção 3 aborda a modularidade, a Secção 4 aborda o encapsulamento, a Secção 5 aborda a abstração de dados e a Secção 6 indica as estratégias usadas para resolver as questões apresentadas. O relatório termina com conclusões na Secção 7, onde é também apresentada uma análise crítica dos resultados obtidos.

2 Estruturas de Dados

Este trabalho tem por base uma estrutura principal denominada **TCD**, sendo que depois é utilizada esta mesma mas com abstração de dados, a **TAD**.

Esta estrutura possui:

- Uma **struct utilizador** que contêm os atributos do utilizador e está definida como ***GHashTable****;
- Uma **struct posts** que contêm os atributos de um post e está definida como ***GHashTable****;
- Uma **struct date_posts** que contêm os atributos de um post está definida como ***GList****, sendo esta ordenada por data;
- Uma **struct tags** que contêm os atributos de uma tag e está definida como ***GHashTable****.

```
struct TCD_community{
  GHashTable* utilizador;
  GHashTable* posts;
  GList* date_posts;
  GHashTable* tag;
};
```

Figura 1: Estrutura *TCD*

```
struct utilizador{
        gint key_id;
        gchar* nome;
        gchar* bio;
        GList* posts_frequentados;
        GList* posts_perguntas;
        gint posts_u;
        gint reputacao;
};
```

Figura 2: Estrutura *Utilizador*

```
struct posts{
    gint key_id_post;
    Date data;
    gchar* data_string;
    gint score;
    gint owner_user_id;
    gchar* title;
    gchar* body;
    gint post_type_id;
    gint parent_id;
    GList* tags;
    gint answer_count;
    gint comment_count;
};
```

Figura 3: Estrutura *Posts*

```
struct tag{
        gchar* key_tag_name;
        gint id_tag;
};
```

Figura 4: Estrutura *Tag*

3 Modularidade

Modularidade, por definição, é a divisão do código fonte em unidades separadas coerentes. É algo que temos em conta no nosso projeto, existindo, por base, o ficheiro **main.c**, e depois todos os ficheiros individuais relacionados com as **estruturas de dados** usadas e as **queries** propostas. Estes possuem os ficheiros **.c** que contêm o código fonte e os **.h/header files** que definem o que é invocável do exterior.

Modularidade torna-se portanto fundamental para lidar com a complexidade do código, de tal forma que o código dos programas deve ser dividido por unidades modulares pequenas e autônomas, devendo-se ter em especial atenção a criação dos módulos que representam **abstração de dados**.

4 Encapsulamento

Encapsulamento baseia-se na garantia de **protecção** e **acessos controlados aos dados**. É mais um aspeto que temos em conta no nosso projeto de forma a que exista uma divisão entre as operações que são públicas e aquelas que são internas ao módulo. Estas são privadas e, portanto, são apenas acessíveis do exterior atráves das funções disponibilizadas na **API**. Algo vísivel na figura 5, neste caso abordando as **tags**.

Desta forma um tipo de dados permite ter várias instanciações, visto que os módulos das estruturas e das queries se tornam mais genéricos.

De referir que o encapsulamento não é total, isto é, só são feitos clones no caso das strings, atráves da função **mystrdup**. Ao invês disso utilizamos uma abordagem *shallow*, por uma questão de eficiência tendo sempre o cuidado de não alterar valores das estruturas.

```
struct posts{
    (...)
    GList* tags;
    (...)
};

POSTS create_posts(){
    POSTS p = malloc(sizeof(struct posts));
    (...)
    p->tags = NULL;
    (...)
    return p;
}

GList* get_tags(POSTS p){
    return p->tags;
}

void set_tags(POSTS p, char* str){
    char* cp = mystrdup(str);
    p->tags = g_list_prepend(p->tags, (gpointer) cp);
}

void free_posts(POSTS p){
    if(p){
        (...)
        while(p->tags != NULL){
            char* cp = (char*) (p->tags)->data;
            (p->tags)->data = NULL;
            free(cp);
            p->tags = g_list_remove(p->tags, (p->tags)->data);
        }
        g_list_free (p->tags);
        free(p);
}
```

Figura 5: Exemplo Encapsulamento *Tags*

5 Abstração de Dados

A declaração abstrata duma estrutura esconde dos utilizadores do módulo a implementação concreta, não tendo desta forma acesso à implementação da mesma. Por isso mesmo, previamente, temos a declaração abstrata da estrutura **utilizador**, denominada de **UTILIZADOR**, no *header file*, como mostra o exemplo 6.

```
typedef struct utilizador * UTILIZADOR;
```

Figura 6: Exemplo Declaração Abstrata *utilizador*

6 Estratégias das Queries

6.1 Init

Função que cria a **TCD_community**. Por consequência, inicializa as estruturas relacionadas a essa mesma struct, alocando memória e usando a função da *glib*, **g_hash_table_new**.

6.2 Load

Função que realiza o parse de todos os ficheiros necessários à realização do trabalho. Após receber um **dump_path** para os ficheiros XML, caso não existam falhas na estrutura XML são realizadas três funções: o **getReferenceUser**, o **getReferencePosts** e o **getReferenceTags**.

A primeira realiza todo o parse relacionado com os **utilizadores**, como por exemplo o **id** ou a **reputação**. De seguida são colocados na estrutura todos estes elementos através da função **set_utilizador**.

A segunda realiza todo o parse relacionado com os **posts**, como por exemplo o **id_post** ou o **post_type_id**. De seguida são colocados na estrutura todos estes elementos através da função **set_posts**.

A terceira realiza todo o parse relacionado com as **tags**, como por exemplo o **id_tag** ou o **tag_name**. De seguida são colocados na estrutura todos estes elementos através da função **set_tag**.

Referir que incluimos um ficheiro *debug.h* em todas as queries de modo a que seja possível, ou não, imprimir as respostas no standard output.

6.3 Query 1

Dado o identificador de um post, a função retorna o título do post e o nome de utilizador do autor. Se o post for uma resposta, a função retorna o título e o id do utilizador da pergunta correspondente.

Nesta questão, sendo o valor do **id** igual ao da **key** da tabela de hash, recorremos à função da *glib*, **g_hash_table_lookup** que dado uma **key**, retorna o **value** associado. Caso nada seja encontrado, é retornado NULL.

Tendo agora todos os valores referentes ao **post**, caso este seja uma pergunta, é atríbuido à primeira coordenada o **title** do post e à segunda o **nome** de quem realizou a questão. Encontramos o **nome** do autor da questão invocando o

parâmetro **owner_user_id** na mesma função **lookup** utilizada anteriormente, passando agora a ser esse o **key/id** associado.

Caso seja uma resposta, o primeiro parâmetro é calculado usando a mesma função **g_hash_table_lookup**, mas agora com o parâmetro **parent_id**, que numa resposta retorna o **id** da pergunta ao qual esta respondeu. O segundo parâmetro é igualmente calculado como se fosse uma pergunta, mudando apenas o novo **value** associado.

6.4 Query 2

Pretendemos obter o top N utilizadores com maior número de posts de sempre. Para isto, são considerados tanto perguntas quanto respostas dadas pelo respectivo utilizador.

Nesta questão, utilizamos uma ***GList**** para armazenar os **values** de cada **utilizador**, sendo isto realizado pela função da *glib*, **g_hash_table_get_values**.

Depois esta lista é ordenada por número de posts (de cada utilizador) em ordem decrescente através da função da *glib*, **g_list_sort**, que usa uma função de comparação **compara_posts_u**, que tem em conta o parâmetro **posts_u** da estrutura.

De seguida, fazemos a filtração para o **set_list** do **id** de cada utilizador.

6.5 Query 3

Dado um intervalo de tempo arbitrário, obtemos o número total de posts (identificando perguntas e respostas separadamente) neste período.

Nesta questão, utilizamos uma ***GList**** para armazenar os **posts** ordenados por data que se encontram na estrutura **date_posts**, para de seguida percorrer estes mesmos posts.

Enquanto isso, verificamos se a **data** do post se encontra entre os limites referenciados pela query, através da função **difDatas**, e caso seja uma pergunta, é incrementada a primeira coordenada do par, caso contrário é uma resposta, e é incrementada a segunda coordenada.

6.6 Query 4

Dado um intervalo de tempo arbitrário, retornamos todas as perguntas contendo uma determinada tag. O retorno da função é uma lista com os IDs das perguntas ordenadas em cronologia inversa.

Nesta questão, utilizamos uma ***GList**** para armazenar os **posts** ordenados por data que se encontram na estrutura **date_posts**.

Enquanto isso, verificamos se a **data** do post se encontra entre os limites referenciados pela query, através da função **difDatas**, e se é uma pergunta. Nos posts em que os dois parâmetros são cumpridos, colocamos as tags desses mesmos posts uma nova *GList** usando a função **get_tags**. Verificamos se a **tag** passada como parâmetro pertence à lista anterior usando a função **strcmp**, e caso pertença é inserida no início duma *GList** **res** que vai conter os resultados finais, através da função **g_list_insert**.

De seguida, ordenamos a lista **res** por data através da função **g_list_sort**, de modo a depois poder retornar os **id's** em questão.

6.7 Query 5

Dado um ID de utilizador, devolvemos a informação do seu perfil (short bio) e os IDs dos seus 10 últimos posts (perguntas ou respostas), ordenados por cronologia inversa.

Nesta questão, recorremos à função da *glib*, **g_hash_table_lookup**, para termos o **value** associado ao **id** do utilizador. Desta forma, retornamos a **bio** do utilizador.

Retornamos os últimos 10 posts acedendo à estrutura que tem os posts ordenados por data e, começando no último elemento da ***GList****, vamos percorrendo a lista para trás. Caso o **owner_user_id** seja o requerido adicionamos o **id** do post à lista a retornar.

6.8 Query 6

Dado um intervalo de tempo arbitrário, devolver os IDs das N respostas com mais votos, em ordem decrescente do número de votos.

De referir que tivemos em conta o score das perguntas, ao invês da diferença de votos.

Nesta questão, recorremos à estrutura **date_posts** que possuí as datas ordenadas. Caso o **post** seja uma pergunta e se encontre entre as datas requeridas na query, é inserida numa nova ***GList**** **glvotes** criada para armazenar os valores necessários para a resposta final.

Depois esta lista é ordenada por votos em ordem decrescente através da função da *glib*, **g_list_sort**, que usa uma função de comparação **compara_score**, que tem em conta o **score** da estrutura. Finalmente acedemos ao **id_post** relacionado, através dos dados respetivos da lista **glvotes**.

6.9 Query 7

Dado um intervalo de tempo arbitrário, devolver os IDs das N perguntas com mais respostas, em ordem decrescente do número de votos.

De referir que as respostas tidas em conta abordam o tempo total, e não o intervalo de tempo passado como parâmetro.

Nesta questão, recorremos à estrutura **date_posts** que possuí as datas ordenadas. Caso o **post** seja uma resposta e se encontre entre as datas requeridas na query, é inserida numa nova ***GList**** **glanswers** criada para armazenar os valores necessários para a resposta final.

Depois esta lista é ordenada por número de respostas em ordem decrescente através da função da *glib*, **g_list_sort**, que usa uma função de comparação **compara_answer**, que tem em conta o **answer_count** da estrutura. Finalmente acedemos ao **id_post** relacionado, através dos dados respetivos da lista **glanswers**.

6.10 Query 8

Dado uma palavra, devolver uma lista com os IDs de N perguntas cujos títulos a contenham, ordenados por cronologia inversa.

Nesta questão, recorremos à estrutura **date_posts** que possuí as datas ordenadas. A lista dos posts é iterada do fim para o início de modo a ir em conta da cronologia inversa. Caso o **post** seja uma pergunta, e, usando a função **strstr**, caso a **word** passada como parâmetro esteja presente no **title**, acrescentamos o **id_post** à lista a retornar no final.

6.11 Query 9

Dados os IDs de dois utilizadores, devolver as últimas N perguntas, em cronologia inversa, em que participaram dois utilizadores específicos, via pergunta ou respostas.

Nesta questão recorremos à estrutura **utilizador** que contém a informação sobre todos os utilizadores, para retirar a informação relativa aos **posts_ frequentados**, de cada utilizador que nos são passados como argumentos. Resta apenas comparar estas duas ***GList**** para retirar os **id_post** que coincidem.

6.12 Query 10

Dado o ID de uma pergunta, obter a melhor resposta, tendo em conta uma média ponderada.

Nesta questão, utilizamos uma ***GList**** para armazenar os **values** de cada **utilizador**, sendo isto realizado pela função da *glib*, **g_hash_table_get_values**.

Depois, identificamos a pergunta ao qual o **post** está a responder, através do **parent_id**. Caso essa pergunta seja o **id** passado como parâmetro, calculamos a média ponderada tendo em conta o **score**, o **comment_count** e a **reputação**. Este último parâmetro é acedido na estrutura do utilizador através da função **g_hash_table_lookup**. Quando encontramos uma média melhor é alterado o **id_post** a retornar no final.

6.13 Query 11

Dado um intervalo arbitrário de tempo, devolver os identificadores das N tags mais usadas pelos N utilizadores com melhor reputação. Em ordem decrescente do número de vezes em que a tag foi usada.

Nesta questão, utilizamos uma ***GList**** denominada **glista** para armazenar os **values** de cada **utilizador**, sendo isto realizado pela função da *glib*, **g_hash_table_get_values**. Esta lista é depois ordenada tendo em conta a reputação através da função **g_list_sort**, para depois manter só os **N utilizadores** necessários, algo que fica guardado numa ***GList**** de nome **aux**.

Criamos uma ***GHashTable**** que vai conter todos as tags. De seguida, percorremos a ***GList**** **posts_perguntas** que só contêm as **N perguntas** da lista **aux**.

Caso o **post_pergunta** esteja entre as datas passadas como argumentos, colocamos numa ***GList**** denominada **tags** as tags desse mesmo **post_pergunta**. Vamos então percorrer esta lista e procurar as tags desta lista na ***GHashTable**** **todas_tags**.

Nesse instante, caso a tag não exista na lista é criada uma nova instância da estrutura **Tag_Unique** e são passados o **tag_name** e a **ocorrência**, que será igual a 1. Caso já exista, as **ocorrências** são incrementas em 1 valor.

De seguida, é ordenada a lista com as **N tags** de acordo com o número de ocorrências.

Para finalizar usamos a função **g_hash_table_lookup** para procurar o **id_tag** relacionado com o **tag_name** no ficheiro Tags.xml.

```
struct tag_unique{
        gchar* key_tag_unique_name;
        gint ocorrencias;
};
```

Figura 7: Estrutura *Tag_Unique*

6.14 Clean

Função que liberta a memória da **TCD_community**. Por consequência, realiza o **free** da estrutura **utilizador**, utilizando a função da *glib*, **g_hash_table_foreach**. De seguida, libertamos a própria ***GHashTable**** através da função **g_hash_table_destroy**. O mesmo sucede-se com a estrutura **posts** e **tag**. A estrutura **date_posts**, sendo uma ***GList**** tem uma forma diferente de libertar a memória. Para tal ser efetuado vamos a todos os parâmetros da mesma e usamos a função **g_list_remove** para libertar a memória de todos os elementos constituintes da lista em questão. De seguida, realizamos o **g_list_free** de forma a libertar a própria estrutura.

7 Conclusão

Tendo em conta os objetivos definidos, o grupo acha que os resultados principais foram obtidos. Além de aprender a trabalhar com ficheiros do tipo **XML**, aprendemos a trabalhar com a biblioteca **GLib**. Aliamos também uma evolução da nossa aprendizagem no tema do **encapsulamento**, da **modularidade** e da **abstração de dados**.

Outros objetivos alcançados são a resolução total das queries e a eficiência que se faz notar nestas mesmas, assim como a libertação total de memória alocada pelo programa.

De uma forma natural, o grupo sentiu dificuldades durante o projeto, sendo uma, por exemplo, a necessidade de alterar a base total do trabalho devido a termos inicialmente considerado a ordenação dos ficheiros, quando só nos faltava resolver a query 11.

8 Bibliografia

1. Prof. F. Mário Martins, *Programação Modular em C*;
2. Prof. F. Mário Martins, *Tipos Incompletos em C*.