LI3: Trabalho Prático
Stack Overflow

Mestrado Integrado
em
Engenharia Informática

Grupo 69

Nome	Nº
Pedro Machado	a33524
João Coutinho	a86272
Maurício Salgado	a71407

5 de Maio de 2018

Conteúdo

1 Introdução **3**

2 Módulos e Estrutura de Ficheiros **3**

3 Estruturas de Dados **5**
3.1 Community 5
3.2 Post 6
3.3 User 7

4 Estruturas de armazenamento auxiliares **7**

5 Queries **8**
5.1 Query 1 - *info from post* 8
5.2 Query 2 - *top most active* 8
5.3 Query 3 - *total posts* 8
5.4 Query 4 - *list questions with tag* 8
5.5 Query 5 - *get user info* 9
5.6 Query 6 - *most voted answers* 9
5.7 Query 7 - *most voted answers* 9
5.8 Query 8 - *contains word* 9
5.9 Query 9 - *both participated* 10
5.10 Query 10 - *better answer* 10
5.11 Query 11 - *most used best rep* 10

6 Notas Finais **11**
6.1 Memória 11
6.2 Eficiência 11
6.3 Reutilização de código 11
6.4 Observações 11
6.5 Apreciação Geral 11

1 Introdução

O presente trabalho foi desenvolvido em linguagem C e teve como principal objectivo a extracção e o armazenamento de dados referentes ao site *StackOverflow*, para a execução de várias interrogações e pesquisas. Para a concretização do objectivo, procedeu-se à extracção da informação contida em vários ficheiros XML recorrendo à biblioteca *libxml2*. Posteriormente esta informação foi armazenada na memória.

A dimensão da base de dados exigiu a criação de diversas estruturas de dados, assim como estruturas de armazenamento adequadas, de forma a facilitar o acesso eficaz às mesmas. Para facilitar todo este processo foi usada a biblioteca *glib* que disponibiliza acesso a estruturas de dados já implementadas.

Considerou-se que o âmbito deste projecto não seria a parte low-level, isto é de criação deste tipo de estruturas e listas de dados. Assim tentou tirar-se o máximo proveito das diferentes estruturas que esta biblioteca disponibiliza.

Este relatório procura justificar as escolhas dos diversos tipos de dados e estruturas de armazenamento, assim como as abordagens específicas das várias interrogações propostas.

2 Módulos e Estrutura de Ficheiros

Foram criados diferentes módulos de acordo com a funcionalidade exigida e de modo a garantir o encapsulamento dos tipos de dados criados. Uma vez que se tentou tirar o máximo partido das estruturas existentes da *glib*, evitando código demasiado extenso, optou-se por uma divisão algo simplista dos módulos.

Segue uma breve descrição dos módulos e da estrutura de ficheiros:

Módulo Loader:

Este módulo é responsável unicamente pelo *parse* dos ficheiros XML e carregamento dos dados nas diversas estruturas. Apenas é invocado pelo módulo interface.

- **loader.h -** Assinaturas das funções de *parse* e de *load*.
- **loader.c -** Funções que envolvem o *parsing* dos ficheiros XML e o carregamento para a memória.

Módulo Search:

Este é o principal módulo de auxílio às *queries* presentes no interface. Neste módulo estão presentes todas as funções de procura e recolha de dados e iteração das estruturas.

- **search.h -** Contém as assinaturas apenas das funções que serão invocadas pelo interface.
- **search.c -** Contém as funções de procura de dados e funções de comparação. é usado como auxiliar ao ficheiro interface.

Módulo Post:

Contém as definições do tipo de dados *Post* (devidamente encapsuladas), assim como todas as funções de *set*, *get* e *free* da estrutura.

- **stack_post.h -** Tipo abstracto *Post* e assinaturas das funções deste tipo.
- **stack_post.c -** Define o TCD da estrutura *Post* usada para armazenar os dados relativos a um *post* e as suas funções: *initt*, *set*, *get* e *free*.

Módulo User:

Contém as definições do tipo de dados *User*, e todas as funções de *set*, *get* e *free* da estrutura.

- **stack_user.h -** Tipo abstracto *User* e assinaturas das funções deste tipo.
- **stack_user.c -** Define o TCD da estrutura User usada para armazenar os dados relativos a um *user* e as suas funções de criação, *set*, *get* e *free*.

Módulo Interface:

É o *backbone* de todo o trabalho. Contém as funções das *queries*, assim como inicialização das estruturas e *free* das mesmas.

- **interface.h -** Ficheiro previamente disponibilizado.
- **interface.c -** Contém a implementação das funções das *queries*, respeitando as assinaturas do *header* disponibilizado. Importa também os *headers* dos tipos de dados, funções *search* e *load*.

Módulo Teste:

Módulo exclusivamente de testes, para uso pessoal.

- **test.h -** Assinaturas das funções de teste, a serem importadas no ficheiro *main.c*.
- **test.c -** Unicamente funções de teste, de impressão de dados, e teste dos *outputs* das *queries* pedidas.
- **main.c -** Usado maioritariamente para testes e invoca funções contidas no ficheiro. *test*

3 Estruturas de Dados

3.1 Community

```c
#include <interface.h>
#include <glib.h>
#include <loader.h>
#include <search.h>
#include <stack_user.h>
#include <stack_post.h>

struct TCD_community{
  GHashTable *posts;
  GHashTable *users;
  GHashTable *dates;
  GHashTable *tags;
};
```

Figura 1: Estrutura Post definida no ficheiro **interface.c**

O ficheiro interface.c é o ficheiro principal, responsável por inicializar a estrutura e fazer todas as interrogações. Assim, são importados os módulos *loader* (carregamento do XML), *search* (funções de auxílio às interrogações) e os ficheiros dos tipos *User* e *Post*.

GHashTable posts: Contém apontadores para a estrutura *Post* por nós definida, usando como *key* o ID (único) de cada *post*. A escolha desta estrutura possibilita tempos de acesso constantes numa base de dados de grandes dimensões.

GHashTable users: São guardados apontadores para as estruturas *User*, usando o ID como *key*, permitindo assim acesso rápido a um *user* dado o seu ID, tirando partido dos rápidos tempos de acesso da *hashtable*.

GHashTable dates: De modo a facilitar *queries* que nos restrigem a um intervalo de tempo e uma vez que interessa a ordem cronológica dos *posts*, nesta *hashtable* foi usada uma data como *key* com granularidade diária. O conteúdo será uma árvore binária com a data e tempo de cada *post*, que por sua vez contém um apontador para o post em questão.

GHashTable tags: A mais simples das *hashtables*, usa como *key* uma *string* que é a própria *tag* e tem como valor o seu identificador.

A escolha recaíu maioritariamente sobre as *hashtables* devido aos rápidos tempos de acesso e à facilidade de utilização. O facto de a ordem ser irrelevante em grande parte das queries foi outro dos critérios que motivou a escolha.

Quanto às datas, uma vez que a ordem interessa, decidiu-se ter uma hashtable em que cada key corresponde a um dia sob a forma de data, podendo percorrer um intervalo de tempo incrementando a data. O conteúdo são árvores binárias que nos garantem que percorremos os *posts* por ordem cronológica.

3.2 Post

```c
#include <stack_post.h>
#include <common.h>

struct post{
  long  id;
  int  type;
  long  id_parent;
  long  id_user;
  char* title;
  int score;
  int answers;
  GDateTime* date;
  GSList* tags;
};
```

Figura 2: Estrutura Post definida no ficheiro **stack_post.c**

Em relação aos vários ID's guardados na estrutura *post* (parent, user, e post id) foi usado o tipo *long*, já que são números tendencialmente grandes, e asseguramos assim o uso de 4 bytes para os guardar.

Fez-se uso, sempre que possível, das várias estruturas já implementadas da *glib*. Neste caso específico da seguinte forma:

GDateTime: para guardar a data e o tempo da criação do *post*, facilitando comparações e a leitura da string de data presente no XML.

GSList: para guardar os identificadores das *tags* de um *Post pergunta* através de uma lista ligada da *glib*, que nos permite obter o tamanho, fazer reverse da lista, entre outras manipulações úteis.

Apesar de haver outra estrutura organizada por data, decidiu-se que incluir a data em cada *post* facilitaria a organização de uma qualquer lista de *posts* por ordem cronológica. A variável *answers* guarda o número de respostas a um *post* que seja uma pergunta, o que iria ser útil numa das queries. Quanto às *tags*, decidiu-se que seriam guardadas numa lista de ID's pois, seria mais rápido e eficiente comparar *int*'s do que seria *String*'s.

Além da definição concreta do tipo de dados presente no ficheiro, temos também as várias funções para obter e atribuir diferentes valores aos campos, assegurando assim o encapsulamento dos dados. Foi criada também uma função *free* para a estrutura *Post*, que é utilizada na criação das *hashtables* para que se possa libertar a memória aquando da destruição da estrutura.

3.3 User

```c
#include <stack_user.h>
#include <common.h>

struct user{
  char* name;
  char* bio;
  long id;
  int reputation;
  int postcount;
  GSList* posts;
```

Figura 3: Estrutura Post definida no ficheiro **stack_user.c**

À estrutura *User* por nós definida, para além dos campos expectáveis, foi também acrescentada uma variável *postcount* a ser incrementada aquando do carregamento dos *posts*. Decidiu-se usar novamente uma *GSList* para guardar apontadores para os *posts*, que facilita o acesso imediato a todos os *posts* desse *user*, que não aconteceria ao guardar ID's.

À semelhança da estrutura *Post*, de modo a assegurar o encapsulamento dos dados foram criadas as funções: get, set e free.

4 Estruturas de armazenamento auxiliares

Como referido anteriormente, sempre que possível foram usadas as diversas estruturas da *glib*, destacando-se duas principais:

GSList: Esta implementação de lista ligada foi usada sempre que foi necessário criar listas de *posts* ou de *users*, pois permite inverter, ordenar (sempre que se providenciasse uma função de comparação), e guardar qualquer tipo de apontador de estrutura. Esta versatilidade fez com que fosse escolhida na maior parte das situações.

GPtrArray: Esta estrutura foi a segunda escolha de armazenamento, pois na iteração de árvores não era possível retornar qualquer tipo de dados (a GSList precisava de actualizar o apontador para o inicio da lista). O GPtrArray permitia guardar também apontadores de qualquer estrutura, mas não necessitava de retornar nenhum valor, mudando simplesmente o espaço alocado em memória. Tudo isto fez com que a escolha recaísse para esta estrutura sempre que eram usadas árvores de procura.

GDateTime: Estrutura usada para armazenar a data e o tempo de um *post*; permitia acrescentar dias (útil para iterar a *hashtable* com granularidade diária) e tinha já definidas funções de comparação. Usada também durante o *parse* do ficheiro, pois reconhecia o formato da string usada para o campo Data, permitindo a criação de uma instância de data-tempo rapidamente usando esta *string*.

5 Queries

5.1 Query 1 - *info from post*

Provavelmente a mais simples das queries, onde é simplesmente consultada a *hashtable* **posts** usando o ID do *post*, e, se este existir, procura na hashtable **users** pelo ID do autor, retornando assim o par com a informação.

Estratégia: Sendo uma query simples, a estratégia usada foi também a mais simples, tendo sido implementada a função **get_post_pid** no módulo *stack_post* que, consoante o tipo do *post*, devolve o seu ID ou o ID da pergunta à qual é resposta.

5.2 Query 2 - *top most active*

Nesta query foi usada uma *G*SList e percorre-se a tabela de **users** verificando o número de *posts* de cada user.

Estratégia:

- Previamente, na fase de *l*oad, usou-se a variável de *post count* em cada *user* para facilitar as comparações.
- Para tornar mais eficiente a criação da lista, manteve-se a lista ordenada por número de *posts* e comparou-se simplesmente com a cabeça da lista.
- Criação da primeira função de *compare* para fazer *sort* da lista: **compare_count**

5.3 Query 3 - *total posts*

Sendo este um *post* com intervalo de tempo, foi criada uma função de iteração de árvores, que por sua vez foi passada à função que itera a *hashtable dates f_between_dates*, cujos elementos são árvores de *posts*.

Estratégia:

- Criação de uma função que recebe uma função *boolean* de iteração, eliminando assim a redundância de ter código repetido.
- *Hashtable* de datas com granularidade diária, permitiu percorrer apenas o intervalo pedido em oposição a percorrer todos os elementos.

5.4 Query 4 - *list questions with tag*

A abordagem foi similar à *Query* 3, uma vez que também era dado um intervalo de tempo. Assim, foi também criada uma função de iteração que recebe o endereço de uma lista e vai acrescentando os *posts* que contenham a *tag*.

Estratégia:

- Usou-se novamente a função que percorre a *hashtable* de datas e as árvores de *posts*, enviando-lhe apenas uma função de iteração, eliminando assim redundância de código.
- Tirou-se partido das *tags* serem *ints* e serem mais rápidos de comparar do que *strings*.
- Uma vez que a função de iteração recebe apenas um parâmetro de *User Data*, usou-se a posição 0 da *GSList* para guardar a *tag* dada.

5.5 Query 5 - *get user info*

Uma *query* simples, cuja parte mais desafiante tenha sido talvez retornar a lista dos últimos 10 posts.

Estratégia: Foi tirado partido da escolha prévia de uma *GSList* para guardar os *posts* de cada *User*, o que facilitou retirar os últimos *posts* e inverter a lista.

5.6 Query 6 - *most voted answers*

Uma vez que esta *query* implica uma travessia dos *posts* num dado intervalo, a estratégia usada foi similar à da *Query* 3. Usou-se a função *f_between_dates* à qual passamos uma função de iteração. Foi usada a estrutura GPtrArray para guardar os *scores*, pois como na iteração das árvores não podemos retornar qualquer tipo de valor, não seria possível usar uma lista ligada.

Estratégia:

- Foi criada uma função de comparação, para comparar o *score* de cada *post* com a cabeça de uma lista já ordenada de modo a maximizar a rapidez.
- A função principal desta *query* é novamente uma função *boolean* a ser passada à nossa função que percorre as datas sequencialmente.

5.7 Query 7 - *most voted answers*

A abordagem foi quase idêntica à Query 6, uma travessia entre duas datas na qual eram filtrados os *posts* tipo 1 e verificado o seu respectivo *score*.

Estratégia: Foi decidido que na fase de *load* do XML, cada *post* resposta incrementava a *answer-count* de cada *post*. Logo foi criado um GPtrArray de tamanho N, ao qual se iria comparando com a cabeça e adicionando se fosse maior.

5.8 Query 8 - *contains word*

Esta *query* exigia que fossem percorridos todos os *posts*, a ordenação por data seria apenas exigida na lista a retornar, daí que tenha sido usada a *hashtable Posts*, e posteriormente a ordenação por data. Para saber se a *string* existia no título foi usada a função *strstr* da biblioteca *string.h*.

Estratégia:

- Criação de GSList de tamanho N, e novamente a estratégia de comparar com a cabeça da lista.
- Foi tirado partido de armazenar o campo da data em cada *post* (e não apenas como chave da árvore), assim no final tendo uma lista já com os N posts que continham a palavra seria mais fácil ordenar por data, ao invés de percorrer ordenadamente os *posts*.
- Nova função de comparação: *compare_date* para ordenação do *array* final por ordem decrescente de data.

5.9 Query 9 - *both participated*

Nesta *query* o esforço computacional exigido foi passado para o *load* dos ficheiros XML, onde foi criada uma GSList em cada *User*, com apontadores para os *posts* feitos por cada *User*. Assim a abordagem foi verificar o número de *posts* de cada *User*, bastando um deles não ter *posts* para retornar a lista vazia. Caso ambos tivessem *posts*, cada elemento da lista seria comparado com a lista de *posts* do outro utilizador.

Estratégia: A única estratégia usada foi mesmo carregar a lista de *posts* de cada *User* durante o *parse* (algo que viria a ser útil também para a Query 11). Compararam-se de seguida os *posts* de ambos os *users* recorrendo a dois ciclos *for*, e recorreu-se a uma função auxiliar *add_common_post* que apenas carrega os *posts* em comum para uma lista.

5.10 Query 10 - *better answer*

Apesar de nesta *query* ajudar ter uma lista de respostas em cada *Post* pergunta, decidimos que por uma *query* não valia a pena sobrecarregar mais a fase de *load* dos ficheiros XML. Dado que não é uma *query* muito intensiva a nível computacional, seria um compromisso desvantajoso perder tempo no *load* da estrutura para o ganhar numa *query* deste tipo. Assim sendo, optou-se por iterar toda a *hashtable* de *posts*, e procurar as respostas cujo *parent id* correspondesse ao ID fornecido.

Estratégia: Filtraram-se os *posts* de resposta ao ID, e uma vez que não era pedida uma lista, mudámos a abordagem da lista ordenada, e usou-se um *LONG_pair* com a finalidade de guardar o valor máximo encontrado, e substitui-lo caso fosse encontrado um *post* com maior *score*. Para calcular o *score* foi usada uma função auxiliar que o calculava segundo a fórmula dada no enunciado.

5.11 Query 11 - *most used best rep*

Não tendo ficado claro aquilo que era pretendido no enunciado, optou-se por encontrar o top global de *users*, e de seguida filtrar os seus *posts* que pertenciam ao intervalo de tempo dado. Já com a lista final de *posts* construiu-se a lista de *tags* e contagem.

Estratégia:

- Foi percorrida a *hashtable* de *Users* e retirado o top N utilizadores, recorrendo novamente à estratégia de comparar com a cabeça da lista de N elementos.
- Uma vez obtidos os N utilizadores, os respectivos *posts* foram concatenados numa única lista de modo a serem filtrados por perguntas e intervalo de tempo.
- A lista de *post* deu origem a uma lista de *tags*, mais concretamente ID's, pois seria mais eficiente comparar números em oposição a *strings*.
- Foram criados pares (usando a estrutura *LONG_pair* fornecida) de *tags* com a respectiva contagem.

Observação:

O pretendido nesta *query* seria restringir o top de utilizadores ao intervalo dado, o que sugeria uma alteração profunda da estrutura de dados usada, nomeadamente ao iterar unicamente aquele intervalo de tempo usando as árvores de *posts*, onde seria útil em vez de guardar o ID do *user* guardar um apontador para a estrutura *User* dentro do *Post*. Isto permitira fácil acesso ao mesmo, uma vez que durante a iteração de uma árvore há uma restrição nos parâmetros que poderão ser enviados.

6 Notas Finais

6.1 Memória

Houve o cuidado, mesmo antes de serem criadas as estruturas de armazenamento, de evitar redundâncias e alocações de memória desnecessárias. Assim sendo, libertar a memória usada pelas mesmas foi relativamente simples, já que durante a criação (hashtables, árvores, e listas) eram logo passadas as funções de *free* das estruturas. Foram efectuados vários testes usando o *Valgrind* permitindo descobrir as funções que causavam leaks de memória e garantir também que a memória era correctamente libertada.

6.2 Eficiência

Mais do que a obtenção dos resultados correctos, priorizou-se a elegância e eficácia da solução, embora os resultados enquanto objectivo não tenham sido negligenciados. Ainda que esta filosofia tenha causado diversas reestruturações profundas ao longo da elaboração do trabalho e a sua evolução ter implicado vários recomeços.

6.3 Reutilização de código

Houve certamente uma curva de aprendizagem em relação às estruturas já implementadas da glib, o que nos levou a ter preferências e muitas vezes usar as estruturas consideradas mais eficientes ou que serviam melhor o propósito. Tal levou a que muitas funções fossem re-escritas de modo a aperfeiçoar a solução. Foi criado unicamente o essencial e tentou fazer-se uso do máximo de implementações possíveis, tirando assim partido da estabilidade e fiabilidade das mesmas.

6.4 Observações

Mesmo tendo havido alturas que a estrutura foi alterada e muito do código re-escrito e melhorado, é difícil não achar que haveria espaço para melhorar. Um exemplo concreto é o caso das funções de iteração, que, por serem enviadas a uma única função que percorre uma hashtable e posteriormente árvores binárias, permitiu eliminar muita redundância de código ao evitar criar uma função para percorrer cada estrutura. Implementar algo semelhante em todas as queries seria difícil, dada a limitação de parâmetros nas funções de iteração e, por isso ter que recorrer a truques. No entanto, seria talvez a solução mais elegante e eficaz.

6.5 Apreciação Geral

Concluído o trabalho, entendemos que a solução apresentada é eficiente e bem estruturada, seguindo as normas de modularidade e encapsulamento dos dados. Foram efectuadas várias medições de tempo que comprovam a boa performance das abordagens seguidas nas várias queries. No geral faz-se uma avaliação muito positiva das diversas métricas consideradas e entende-se que a abordagem ao problema foi concisa e directa, não havendo assim espaço para muito mais simplificação e melhorias de desempenho.