Laboratórios de Informática III - Projecto em C

Adriana Meireles (a82582), Eduardo Jorge Barbosa (a83344), Filipe Monteiro (a80229)

***Resumo*—Este trabalho prático teve como objectivo a organização de grandes volumes de dados de forma a responder a questões, pré definidas, em tempo útil. Os objectos de estudo foram vários dumps do *Stack Exchange*.**

I. INTRODUÇÃO

O cerne deste trabalho prático consistiu em estruturar a informação relativa a vários posts do *Stack Exchange* de forma a responder em tempo útil às queries fornecidas pelos professores. Para tal, o conhecimento adquirido noutras unidades curriculares, como por exemplo Algoritmos, Programação Imperativa e Calculo de Programas, foi de extrema importância, bem como, um estudo sobre optimização de código C (linguagem exigida por esta unidade curricular). A realização deste projecto foi repartida em várias fases: análise dos dados, escolha das estruturas de dados para os representar, como fazer parse dos ficheiros, resposta as queries e por fim, optimização e limpeza do código. De forma a promover boas práticas de programação e desenvolver um workflow eficiente, todo o código desenvolvido era submetido para um repositório *GIT*.

II. CONCEÇÃO DO PROBLEMA

A. ANÁLISE

Analisando as queries fornecidas pelos professores, restringiu-se o número de ficheiros a serem lidos a 3:

- Users;
- Posts;
- Tags.

Tornou-se também evidente que eram exigidos vários tipos de ordenação: por datas, por score, por número de posts, etc. No entanto, uma grande maioria das queries tinham sempre como base um filtro por datas.

Em relação aos utilizadores tornou-se notório que bastava relacionar o seu id com resto da informação pertinente.

Finalmente,dado que as tags apareciam em várias queries foi bastante discutido como as guardar. Do notar que segundo a documentação no máximo apenas aparecem 5 tags por questão.

III. CONCEÇÃO DA SOLUÇÃO

Decidiu-se usar para a estrutura principal 3 *Hash Tables* da *GLIB* e uma estrutura com dupla ordenação *ad hoc*.

As *tags* ficaram agrupadas numa *Hash Table* onde cada chave é a *tag*, em formato *String*, e o valor, o *id* correspondente. Contemplou-se implementar uma *trie* ou uma *inverted index* de forma a ser mais eficiente procurar as tags, mas dado o aumento do tempo de load, não o fizemos.

Por sua vez, os *Posts* ficaram tanto numa *Hash Table* como numa estrutura por nós criada, separados por *Questions* e *Answers*.

Os *Users* ficaram numa *Hash Table*.

Tanto a procura como inserção numa *Hash Table* é $\mathcal{O}(1)$, e estas são as operações mais utilizadas neste projeto. Para representar os *Posts* por datas foi pensado a utilização de àrvores balanceadas. Consideramos implementar uma *Red and Black Tree* dado a inserção ser menos pesada que numa *AVL* mas não resolvia o problema de várias ordenações. Surgiu daí a nossa estrutura, um array de arrays com *GSequences* o que permite uma ordenação por datas e uma outra ordenação à nossa escolha.

Finalmente, para realizar o parse utilizamos o parser *SAX* da biblioteca *libxml* dado o tamanho dos ficheiros e apenas ser necessário le-los uma única vez.

IV. PARSER

Escolheu-se utilizar o parser do tipo *SAX*. Este parser é orientado a eventos e recomendado pela documentação do *libxml* para dar parse a grandes ficheiros. Ao contrário das duas outras alternativas este parser não constrói a árvore do xml em memória sendo muito mais rápido (cerca de 1.5x mais rápido). Um lado negativo deste parser é não ser possível editar os ficheiros xml nem percorrer várias vezes o ficheiro mas estes factores não têm qualquer relevância para este trabalho.

As outras duas alternativas eram *DOM based* e *DOM based* com *API tipo SAX*. A primeira cria a árvore do xml e carrega toda para a memória, sendo extremamente lenta além de ocupar muita memória (5x mais o tamanho do ficheiro). A sua travessia é bi direcional e não é orientada a eventos. A última alternativa é uma versão mais evoluída do *pure DOM*, sendo que não carrega a árvore toda para a memória mas sim à medida que se faz a travessia, a API é orientada a eventos. No entanto por ir construindo a árvore é mais lenta que a *SAX*.

V. ESTRUTURAS DE DADOS

Utilizou-se a biblioteca *GLib* para a implementação de maior parte das nossas estruturas. A estrutura principal é a seguinte:

```
struct TCD_community{
    GHashTable* profiles;
    GHashTable* posts;
    GHashTable* tags;
    TARDIS type40;
    long n_tags;
};
```

A. POSTS, QUESTIONS, ANSWERS

A representação de um *Post* foi um ponto interessante deste trabalho, visto dividirem-se em *Questions* e *Answers*. Dado partilharem atributos, decidiu-se implementar uma *union*, de forma a conseguirmos juntar os *Posts* ou separá-los como nos fosse mais conveniente.

```
struct post{
    long type; // 1 Question 2 Answer
    union content{
        QUESTION q;
        ANSWER a;
    }content;
};
```

Na representação de uma *Question* além da informação básica guardamos os *Ids* das respostas correspondentes. Os *Ids* são inseridos via *insertion sort* comparando a data de criação de cada resposta. Desta forma é possível obter todas as respostas de uma dada pergunta. Escolheu-se utilizar um *GArray* visto que o número de inserções é pequeno mas desconhecido, à partida.

```
struct question{
    MyDate creation_date;
    GArray* id_answers;
    char* title_question;
    char* tags;
    long id_question;
    long owner_id_question;
    long n_answers;
    long comments;
    int score;
};
```

B. USERS

Escolheu-se utilizar uma *Hash Table* para agrupar os *Users*. Cada chave é o *Id* do *User* e o valor é a instância que representa aquele *User*. Juntamente com os atributos que representam um utilizador guardamos numa *GSequence* as questões e respostas criadas pelo dito utilizador. Escolhemos guardar desta forma em vez de um array fazendo *insertion sort* pois desconhecemos o número de inserções. Além disso, é possível ordernar por vários critérios (um de cada vez).

```
struct profile{
    GSequence* posts;
    char* about_me;
    char* name;
    long n_posts;
    long id;
    int reputation;
};
```

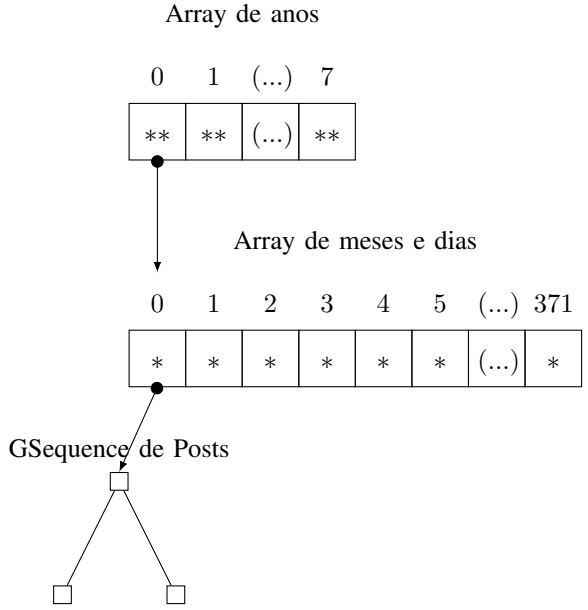
C. TARDIS

Com vista a resolver o problema de vários tipo de ordenação, maioritariamente em relação aos *Posts*, decidiu-se criar uma estrutura que consiste num array de arrays com *GSequences*.

```
struct tardis {
    GSequence*** year_questions;
    GSequence*** year_answers;
    int years;
};
```

O primeiro array corresponde aos anos, o segundo array corresponde aos meses e aos dias desse ano. O tamanho de cada array de meses e dias é sempre 31*12. Desta forma torna-se mais fácil iterar sobre a estrutura. Apesar desta forma existirem *"buracos"* não são um número que justifique abandonar esta estratégia. Cada array de meses e dias só é criado quando o primeiro *Post* com uma data nessa alcance é processado, desta maneira foi possível optimizar a inserção nesta estrutura e gastar menos memória. Com este array de arrays, temos uma granularidade de dias, meses e anos que é o pedido pelas *queries* e dado os *Posts* estarem numa *GSequence* é possível ordenar pelo critério que nós quisermos. Num caso extremo em que temos que ordenar um número enorme de *GSequences* demora $\mathcal{O}(\log N)$ por *GSequence*, onde o N é o número de *Questions* ou *Answers* (pequeno). De facto, é possível obter uma complexidade amortizada de $\mathcal{O}(1)$ nas queries que peçam Top N entre data X e data Y. Para iterar pela TARDIS, o índice do array de anos é calculado subtraindo 2008 ao ano passado como argumento, dado que segundo a documentação do *Stack Exchange* o ano mais antigo é esse. Por sua vez, os meses e dias seguem a seguinte fórmula: $dia - 1 + (mes - 1) * 31$

Segue-se uma representação da TARDIS.



D. Datas

Visto que a granularidade pedida era apenas até aos dias, escolhemos criar a nossa estrutura de datas que abrange até ao milissegundo. Desta forma a ordenação por datas fica mais estável.

VI. QUERIES

A. QUERY 1

Dado o *Id* de um *Post* vai-se buscá-lo consultando a *Hash Table* e verifica-se se é uma *Question* ou uma *Answer*. Caso seja uma *Answer* obtém-se o *Id* da *Question* correspondente. No fim, é chamada uma função auxiliar que recebe uma *Question* e devolve o título e o nome do criador. O nome é obtido consultando a *Hash Table* dos *Users*.

B. QUERY 2

Todos os *Users* são percorridos e é feito um *insertion sort* num array com N+1 elementos tendo por base o número de *Posts*. No fim removemos o elemento N+1. Poderia ser otimizado.

C. QUERY 3

A TARDIS é percorrida duas vezes dentro desses intervalos de tempo, uma para cada tipo de *Post*. No fim é visto o tamanho das *GSequence* e é feito o *return*.

D. QUERY 4

A TARDIS é percorrida entre as datas, percorre-se as tags de cada *Question* à procura da tag e se existir é posta num array. Podia ser otimizada.

E. QUERY 5

Dado o *ID* de um *User* obtém-se o seu perfil. Do perfil conseguimos a *short bio* e ordenamos a *GSequence* inversamente e tiramos os 10 primeiros.

F. QUERY 6

A TARDIS é percorrida entre as datas dadas e devolve uma *GSequence* ordenada por score, que é convertida para um array com N *IDs*.

G. QUERY 7

A TARDIS é percorrida entre as datas dadas e devolve uma *GSequence* ordenada por número de respostas. De seguida filtramosas as respostas pelas que possuem perguntas dentro do tempo pedido finalmente organizamos as questões com mais respostas dentro desse tempo.

H. QUERY 8

A TARDIS é percorrida entre as datas dadas e devolve uma *GSequence* ordenada inversamente por datas. É feito um *for each* na *GSquence* que verifica se contém a palavra e em caso afirmativo é guardado o *Id*.

I. QUERY 9

Dados os *Ids* dos dois utilizadores, vê-se qual dos dois têm menos *Posts* consultando a *GSequence*. No *User* com menos *Posts* percorre-se a *GSquence* por ordem cronológica inversa e em cada *Post* verifica-se se o outro *User* participa na thread.

J. QUERY 10

Dado o *Id* consulta-se o *GArray* com as respostas e para cada aplica-se a fórmula dada.

K. QUERY 11

Obtemos os Top N *Users*. Percorremos a TARDIS entre das datas dadas e filtramos as *Questions* que não fora feitas por nenhum dos top *Users*. Finalmente as *Questions* são percorridas, as tags removidas e processadas. É criado um array de correspondência para a *Tags* que sempre é encontrada dada tag é incrementado numa unidade. No fim verificamos nos N indices com maiores elementos e fazemos a correspondência inversa.

VII. OTIMIZAÇÕES

O principal foco a nível de otimizações foi estruturar a estutura principal segundo as queries. Fora isso, várias otimizações a nível de código foram feitas, *exempli gratia* $++counter$ vs $counter++$, switch vs vários ifs, etc. Foram usadas algumas flags de otimização do *GCC*, a mais notória é *-march=native* que gera código assembly específico para o CPU do computador. De forma a ocupar menos memórias, os elementos das estruturas estão ordeanados por tamanho dos tipos.

VIII. MODULARIDADE

Dado o tamanho do projecto foi preciso dividi-lo em várias fases.

- Init;
- Pase/Load;
- Estrutura;
- Queries.

Na fase do Init a estrutura principal é inicializada, imediatamente a seguir começa o Parse/Load. Os ficheiros xml são carregados para a memória (um de cada vez e aos poucos) e processados para a estrutura.

O código responsável pelas queries foi deixado para o fim, sendo que o código das estruturas e do parser foi desenvolvido em paralelo após as estruturas estarem desenhadas.

IX. ENCAPSULAMENTO

Visto que usamos o parser *SAX* não é possivel escrever nos ficheiros xml. Toda a informação vem de fora e é apenas organizada de outra forma.. Dado que o ficheiro *interface.h* corresponde a *API* do programa e sendo que nenhuma *query* devolve uma instância do estado interno o programa é fechado para o exterior. Dado a quantidade enorme de informação não é viavel estar a fazer "cópias"das estruturas sempre que se acede à estrutura interna. Tendo em conta este custo tempora, decidiu-se não o fazer.

X. DIFICULDADES

O facto de ser utilizar *GLIB* levou a algumas dificuldades. A *TARDIS* foi inicialmente implementada usando *GPtrArrays*, e o encadeamente destas estruturas levantou vários problemas dado não terem funções genéricas. A utlização de *function pointers* também foi confuso no início mas rapidamente se tornou normal. Tentou-se também implementar as funções mais básicas e mais usadas em *inline assembly* mas dado a falta de tempo teve que se abandonar a ideia.

Dados o número de estruturas instanciadas, limpar os *memory leaks* tornou-se numa tarefa complexa.