



Universidade do Minho

Laboratórios de Informática III

2017/2018

Relatório do trabalho

Curso: MIEI

2ºAno

2ºSemestre

Grupo 2



António Lopes

A74357



Bernardo Viseu

A74618



Fernando Pereira

A75496

Índice

1. Introdução	2
2. Tipo Concreto de Dados	2
3. Estrutura de Dados usada	3
4. Modularização Funcional	4
5. Abstração de Dados	5
6. Parser	6
7. Estratégias	7
7.1 Queries	7
7.2 Melhoramento de Desempenho	9
8. Conclusão	9

1. Introdução

Este trabalho foi nos proposto pelos docentes da unidade curricular Laboratórios de Informática III e tem como principal objetivo o desenvolvimento de um sistema capaz de processar ficheiros XML que armazenam as várias informações utilizadas pelo Stack Overflow. Uma vez que vão passar pelo nosso programa milhões de dados, torna-se um desafio tornar esta programação em larga escala eficiente, respondendo ainda às 11 interrogações propostas no enunciado.

2. Tipo Concreto de Dados

Dentro do dump que nos é fornecido para testar o programa (divididos os dados nas pastas android e ubuntu) encontram-se os ficheiros Badges.xml, Comments.xml, PostHistory.xml, PostLinks.xml, Posts.xml, Tags.xml, Users.xml, Votes.xml. Para a realização do programa apenas precisamos dos seguintes ficheiros, que contêm a seguinte informação:

- **Posts.xml**

Ficheiro com os dados de todos os comentários. Apenas é necessário recolher certa informação das perguntas e respostas. Distinguidos através do Id. As perguntas têm PostTypeId=1 e as respostas PostTypeId=2.

- **Users.xml**

Ficheiro com os dados de todos os Users. Distinguidos através do Id.

- **Tags.xml**

Ficheiro com os dados das tags presentes em cada post, devido à query 11 é necessário armazenar alguma informação de cada tag na estrutura.

O nosso **tipo concreto de dados** (TCD) contém as estruturas que foram definidas para armazenar os dados, que são uma HashTable para os Users e um Array para os anos, que contém um array com meses e ainda outro array com os dias, formando assim um calendário para inserir os posts cronologicamente. Foi usada a biblioteca Glib na implementação das estruturas no programa.

```
struct TCD_community{  
  
    GArray* anos;  
  
    GHashTable* userss;  
  
    GHashTable* tagsht;  
  
};
```

3. Estrutura de dados usada

As estruturas foram implementadas com o auxílio da biblioteca Glib.

- **Users** - Para os users foi usada uma HashTable, onde foi armazenado o Id, Nome, AboutMe e a Reputation de cada user, sendo a key o ID do User.

A função responsável pela criação da estrutura **USER**, contida em **userHashTable.c**, é a seguinte:

```
USERS create_hashtable_users(long id, char* name, char* aboutme, int reputation)
```

- **Posts** -

Array dos Anos

2009	2010	2011	2012	2013	2014	2015	2016	2017
------	------	------	------	------	------	------	------	------

Array dos Meses

Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Array dos Dias

1	2	3	4	5	6	7	8	...	31
---	---	---	---	---	---	---	---	-----	----

HashTable das Perguntas

Id, Data, Score, UserId, Titulo, CommentCount, Tags, NumRespostas

⋮

HashTable das Respostas

Id, Data, Score, UserId, CommentCount, NumRespostas, ParentId
--

⋮

As perguntas e respostas foram organizadas conforme a data do post. Tendo nós o calendário do ano, inserimos os posts nos dias corretos da sua data, por acharmos que isto nos facilitaria mais tarde na resolução das queries. As perguntas e respostas ficaram em HashTables separadas.

A função responsável pela criação da estrutura **ANOS**, contida em **anosArray.c**, é a seguinte:

```
ANOS create_array_anos (GArray* meses_a)
```

A função responsável pela criação da estrutura **MESES**, contida em **mesesArray.c**, é a seguinte:

```
MESES create_array_meses (GArray* dias_a)
```

A função responsável pela criação da estrutura **DIA**, contida em **diasNodo.c**, é a seguinte:

DIA create_nodo_dia (GHashTable* questions_a, GHashTable* answers_a)

A função responsável pela criação da estrutura **POSTS**, contida em **postsHashTable.c**, é a seguinte:

POSTS create_hashtable_posts(long id_p, Date data_p, int score_p, long user_id, char* titulo, int comment_count, char* tags, int numeroRespostas)

A key da HashTable é o ID do Post.

A função responsável pela criação da estrutura **ANSWERS**, contida em **answersHashTable.c**, é a seguinte:

ANSWERS create_hashtable_answers (long id_a, Date data_a, int score_a, long user_id_a, int comment_count_a, long parent_id)

A key da HashTable é o ID do Post.

- **Tags** – A estruturada usada para o armazenamento das Tags foi uma HashTable, onde se armazenou o ID, o nome e tagcount (número de vezes que a tag é usada), em que a key é o ID da Tag.

TAG create_hashtable_tag(long id, GString* name, int tagcount)

4. Modularização Funcional

O objetivo da modularização passa por tornar o código mais legível e aumentar o desempenho. Na realização do trabalho dividimos o programa em 2 partes distintas, colocando os ficheiros **.h** na diretoria **include** e os ficheiros **.c** na diretoria **src**. As headers files é onde estão as opções de pré-processamento, são declaradas as funções, declarados os tipos através dos typedef e estão os #includes, mantendo tudo bem mais organizado. Nos **.c** é onde se implementam as funções, já declaradas nos respetivos **.h** de cada ficheiro.

5. Abstração de dados

Os tipos abstratos de dados servem para criar e usar uma classe e definir um conjunto de operações para manipular dados dessa classe. Através da diretiva `#include` dos `.h` é possível usar as funções definidas sem referência a detalhes da implementação. Os nossos TAD's foram implementados e seguem em baixo algumas das funções com que mais jeito deram:

- **Users:**

long get_id(USERS u)

GString get_name(USERS u)*

GString get_aboutme(USERS u)*

int get_reputation(USERS u)

int get_numberOfPosts(USERS u)

void increment_numberOfPosts(USERS u)

- **Posts**

long get_id_p(POSTS p)

int get_score_p(POSTS p)

long get_user_id(POSTS p)

GString get_titulo(POSTS p)*

int get_comment_count(POSTS p)

GString get_tags(POSTS p)*

int get_numeroRespostas(POSTS p)

*GList *get_listaTags(POSTS p)*

- **Answers**

long get_id_a(ANSWERS a)

int get_score_a(ANSWERS a)

long get_user_id_a(ANSWERS a)

int get_comment_count_a(ANSWERS a)

long get_parent_id(ANSWERS a)

- **Tags**

long get_id_tag(TAG t)

GString get_tagName(TAG t)*

int get_tagCount(TAG t)

void increment_tagCount(TAG t)

6. Parser

O nosso parser foi construído com o auxílio das bibliotecas libxml2, mais concretamente usando os #includes `<libxml/parser.h>` e `<libxml/tree.h>`.

A estratégia adotada consistiu em iniciar um calendário com 10 anos (`init_calendario()`), tendo em cada dia 2 HashTables para as Perguntas e Respostas. Depois a função `insert_hashtable_answers_calendario` trata de inserir cada resposta na HashTable das respostas do seu dia (a função `xmlToDate` permitiu-nos transformar uma string numa data), o mesmo faz a `insert_hashtable_questions_calendario` para as perguntas. Estas funções seriam chamadas nas `print_element_namesa` e `print_element_namesq`, respetivamente, que são as funções responsáveis por correr o ficheiro xml e inserir na estrutura os dados que pretendemos guardar. Estas duas últimas funções são chamadas na `parse_answers`, na qual é passado os argumentos path para os ficheiros e o calendário.

O mesmo mecanismo ocorreu para os Users e as Tags. A função `parse_users` é onde é fornecido o path para o ficheiro onde está o dump, correndo depois a `print_element_namesu`, que insere na estrutura os dados que pretendemos guardar para os Users. Nas Tags é na função `parse_tags` que damos o caminho para o dump e na `print_element_tags` onde se corre esse ficheiro e insere na estrutura as tags com os dados pretendidos.

7. Estratégias

7.1 Queries

1. Info from Post

Começamos por percorrer todos os dias do calendário à procura do post com o ID fornecido, quando é encontrado verificamos se trata-se de uma resposta ou pergunta. Caso seja uma pergunta, recolhe-se diretamente o seu título e, com o User ID, ir à HashTable dos Users sacar o nome do autor. No caso de ser uma resposta, vamos à Hash das respostas para tirar o Parent ID (id da pergunta a que se está a responder) e adotar a mesma estratégia que no caso das perguntas acima explicado.

2. Top Most Active

Na realização desta query começamos por percorrer todo o calendário e por cada post incrementar o número de post's nos dados dos User's, após termos o número de post's de cada User, com o auxílio de uma função que compara o número de post's entre dois User's, criamos a lista com o top N de utilizadores com mais post's.

3. Total Posts

Nesta aqui voltamos a percorrer o calendário mas apenas entre as datas fornecidas. Como em cada dia está uma Hash das perguntas e uma Hash das respostas, obtemos essas HashTables e calculámos o seu tamanho (g_hash_table_size) separadamente para as perguntas e respostas (por termos de retornar um par). Somando todos esses tamanhos calculados para cada dia ficamos com o número total de respostas e perguntas entre as duas datas.

4. Questions with Tag

Novamente percorre-se o nosso calendário entre as datas dadas e acedemos às HashTable's das Perguntas. Verificamos se contêm Tag's, em caso afirmativo recolhemos o ID da Question e adicionamos numa lista. Depois inserimos a informação dessa lista na LONG_list do output e ordenamo-la.

5. Get User Info

Nesta query a estratégia adotada passou por correr o calendário a partir do fim (data do último post), acedendo assim aos últimos post's efetuados, e procurar (tanto na Hash das perguntas como respostas) post's com o User ID igual ao fornecido, guardando assim os primeiros os ID's dos primeiros 10 post's que pertencessem ao User.

6. Most Voted Answers

Para a contagem do número de votos utilizamos o atributo Score dos Post's.

Nesta query percorremos o calendário entre as datas dadas e recolhemos para uma g_list os dados de cada resposta. Depois ordenamos essa lista por ordem decrescente do Score de cada post (com a função g_list_sort e uma auxiliar) e tirámos os primeiros N ID's, que são os ID's das respostas com o melhor Score.

7. Most Answered Questions

Nesta query percorremos o calendário entre as datas dadas e recolhemos para uma g_list os dados de cada pergunta. Depois ordenamos a lista por ordem decrescente do número de respostas (atributo presente em todos os Post's) e tirámos os primeiros N ID's, que são os ID's das perguntas com mais respostas.

8. Contains Word

Nesta query, como pedem para devolver os ID's ordenados por cronologia inversa, percorremos o calendário a partir do final. Acedemos à Hash das Perguntas e guardamos numa lista os ID's cujo título continham a palavra dada. Para o caso de não existir menos que N perguntas que contenham a palavra dada, criamos uma lista mais pequena, com apenas os post's cujo título contém a palavra do input.

9. Both Participated

Nesta interrogação, como pedem para devolver as últimas N perguntas em que participaram dois utilizadores, começamos por percorrer o calendário a partir do final e percorrer as HashTable's dos post's (tanto das Perguntas como das Respostas) à procura dos User's ID que coincidam com os do input. Guardamos então os Post's dos User's do input entre as datas em 4 listas: User1 -> listaQuestions1 e listaAnswers1; User2 -> listaQuestions2 e listaAnswers2. A seguir percorremos a listaQuestions1 e verificamos se existe na listaAnswers2 respostas às perguntas feitas pelo User1, em caso afirmativo guardámos o ID da pergunta numa lista result. Voltámos ao mesmo raciocínio mas neste caso percorrendo a listaQuestions2 e verificar se existe alguma resposta às perguntas da listaAnswers1, caso exista, adiciona-se o ID da pergunta à mesma lista result. Finalmente, só falta verificar se existe nas listaAnswers1 e listaAnswers2 respostas à mesma pergunta e adicionar na lista result caso isso se verifique.

10. Better Answer

Nesta query começamos por percorrer o calendário à procura das Respostas com o Parent ID (ID do post a que estão a responder) igual ao fornecido, guardando então numa lista todas as respostas em que essa condição se verifique. Depois com o auxílio da função g_list_sort_with_data (o data seria a HashTable dos User's para recolhermos a reputação dos autores das respostas) ordenamos essa lista por ordem decrescente das melhores respostas após serem feitos os cálculos indicados $((Scr \times 0.65) + (Rep \times 0.25) + (Comt \times 0.1))$ e retornarmos o ID do primeiro elemento da lista que seria a melhor resposta à pergunta do input.

11. Most Used Best Rep

Nesta query foi necessário obter a hashtable das tags pela primeira vez. Percorremos o calendário usando um for com pequenas alterações para delimitar as datas (begin e end). O que se fez foi guardar todas as questões numa GList, depois disto guardamos todos os users que postaram questões nesse período de tempo, também numa GList, e ordenamos essa GList pela reputação dos Users, e guardamos apenas os primeiros N Users dessa GList. Depois disto guardam se todas as questões postadas por esses N Users numa GList. De seguida usa se uma função auxiliar (incrementaTags) para incrementar a count das tags usadas nessas questões, e guarda se todas as tags numa GList, de maneira a usar uma g_list_sort para ordenar as tags pela sua count. Por último enche se a LONG_list que se vai retornar com os primeiros N ID's das Tags da lista ordenada.

7.2 Melhoramento de desempenho

Toda a estrutura foi montada tendo já em mente ter o melhor desempenho possível. Como existem várias queries em que é preciso aceder aos post's entre duas datas, consideramos que organizar os post's num calendário seria a melhor maneira do programa responder às interrogações o mais rápido possível.

Em relação ao parser, houve também uma mudança que alterou significativamente o tempo de execução do programa. Inicialmente tínhamos o parser correndo o ficheiro **Posts.xml** duas vezes, corria e inseria na estrutura as perguntas e corria novamente para inserir as respostas. Modificamos o código e conseguimos que corresse somente o ficheiro dos post's uma vez, inserindo as perguntas e respostas nos seus respectivos locais, usando então a função xmlReadFile apenas uma vez para os post's, o que nos permitiu diminuir o tempo de execução em cerca de 12 segundos.

8. Conclusão

Ao longo da realização deste trabalho adquirimos muitos conhecimentos na linguagem de programação c. Aprendemos muito sobre o funcionamento e utilização de estruturas em c, ganhando outra perspetiva de como a estrutura escolhida para o armazenamento dos dados se torna um fator muito importante na performance do programa, principalmente tendo em conta que estamos a trabalhar com quantidades enormes de informação. Aprendemos também a trabalhar com o parser libxml2. Este trabalho tornou-se ainda mais interessante devido ao facto de ser possível executá-lo de várias maneiras, utilizando diferentes tipos de estruturas. Uma das maiores dificuldades foi decidir qual a melhor estrutura adequada para os dados que tínhamos, mas consideramos que a decisão final de armazenar tudo no calendário foi a adequada tendo em vista as interrogações propostas.

Para concluir estamos ansiosos por rever este trabalho em java para ver as diferenças, dificuldades e de que maneira se deve concluir numa linguagem de programação diferente.