Laboratórios de Informática 3

MIEI - 2º ano - 2º semestre
Universidade do Minho

Projeto em C - Grupo 11


Inês Sampaio
A72626


Frederico Pinto
A73639


André Sousa
A74813

5 de Maio de 2018

Conteúdo

1 Introdução

O Stack Overflow é um site de "pergunta e resposta"(Q&A) destinado a profissionais ou entusiastas da programação. Atualmente, é uma das maiores comunidades online de developers a nível mundial, permitindo a qualquer utilizador colocar questões, que serão depois respondidas por outros utilizadores.

Com cerca de 8.4 milhões de utilizadores, 15 milhões de questões já colocadas, às quais se responderam 24 milhões de vezes, o volume de informação gerada na plataforma é enorme, além de muito útil e valiosa. Mas analisar uma quantidade assim de dados pode tornar-se um processo bastante demorado e complexo, dadas as operações necessárias para cruzar os diferentes tipos de informação disponíveis.

No âmbito da unidade curricular de Laboratório de Informática III, foi proposto o desenvolvimento, em C, de um sistema capaz de processar os ficheiros XML (onde se armazenam os dados utilizados pelo Stack Overflow). Uma vez processada essa informação, deve também ser possível executar um conjunto de interrogações específico, de forma eficiente, como por exemplo saber em que perguntas participou determinado utilizador, ou quantos posts foram feitos num determinado intervalo de tempo.

O presente relatório apresenta e justifica todas as decisões tomadas ao longo do projeto bem como as escolhas das estruturas de dados utilizadas.

2 Modularização

É comum, quando se dá os primeiros passos na programação, que o código fonte seja escrito integralmente num único ficheiro, sem grandes preocupações com a sua compilação e execução.

Perante as dimensões e características do nosso projeto, foi fundamental recorrer-se à modularidade para lidar com a quantidade e complexidade do código que este envolve. Todo o trabalho foi desenvolvido em módulos, ficheiros de código pequenos e autónomos, numa perspetiva de *"large-scale programming"*, de modo a que fosse mais robusto, estável e houvesse maior facilidade em isolar erros de compilação ou execução. Outra das vantagens deste método de programação é a facilidade de navegação e leitura e ainda a potencial melhoria no seu desempenho.

Será possível verificar que fazem parte do trabalho inúmeros ficheiros com extensão *.h* e *.c*, sendo que é nos primeiros que são definidas as funções constantes e outro tipo de dados necessários, que depois o ficheiro .c vai exportar. Ou seja, é nos ficheiros .h que se define a API do respetivo ficheiro .c, declarando a sua estrutura sintática. Por sua vez, é no ficheiro .c que está escrito o código fonte do programa, sendo necessário fazer (no .c) o include dos ficheiros .h.

3 Estruturas de Dados

Para guardar toda a informação necessária à resolução das queries propostas, optamos por usar quatro *hashtables* e uma *AVL*. Neste ponto, é de salientar que implementamos todas estas estruturas recorrendo à biblioteca *<glib.h>*.

Todas estas estruturas, que serão apresentadas de seguida, garantem o encapsulamento de dados, ou seja, apenas as respetivas estruturas, utilizando vários métodos, conseguem inserir, alterar ou modificar as suas variáveis.

3.1 Person

Tendo o Stack Overflow uma quantidade muito grande de utilizadores, é necessária a criação de uma estrutura que armazene a sua informação. Desta forma, foi criada a estrutura **person**.

Esta entidade possui uma identificação única, o seu **id**, um **nome**, uma breve descrição sobre o utilizador (**AboutMe**), a sua **reputação** e o **número de posts** efetuados. Com estes atributos somos capazes de representar toda a informação acerca dos utilizadores, de maneira a responder a todas as queries que os envolvem.

```
1 struct person {
2     int id;
3     char* nome;
4     char* aboutme;
5     int reputacao;
6     int nposts;
```

```
};
```

3.2 Question

Em relação às perguntas, foi criada uma estrutura **question** que guarda a informação pertinente sobre uma pergunta. Esta estrutura armazena o **id** da pergunta, que é único, o seu **título**, o **número de respostas**, o **identificador do autor** da pergunta, o **identificador da melhor resposta**, a **pontuação da melhor resposta**, as **tags** que caraterizam tal pergunta e o **número de tags** que a pergunta possui. Desta maneira, e sendo esta uma estrutura fundamental, conseguimos obter informações preciosas sobre certos parâmetros que nos permitem responder com mais eficiência a determinadas queries.

```
struct question {
    long id;
    char* titulo;
    int nRespostas;
    long autor;
    long bestAnswer;
    float pontuacaoBestA;
    char** tags;
    int nTags;
};
```

3.3 Answer

Um pouco à semelhança do que acontece para as perguntas, foi criada também uma estrutura **answer** que guarda as informação relativas a uma resposta. Nela é guardado o **id** da resposta, que tal como nas perguntas, é único. Para além disso, é guardado o **identificador da pergunta** à qual a resposta se refere. É guardado também o **identificador do autor** e o **número de votos** que a resposta contém. Com estas informações, tal como as perguntas, conseguimos obter um acesso mais eficiente às informações para responder a certas queries.

```
struct answer {
    long id;
    long idpai;
    long autor;
    int Votes;
};
```

3.4 Tag

A criação da estrutura tag surge para que seja possivel responder à query 11. Nesta estrutura é guardado o **id** da tag, o **nome** e um **contador**. Esse contador é sempre inicializado a 0, visto ser apenas uma variável que nos ajudou a resolver a última query.

```
struct tag {
    long id;
    char* nome;
    int contador;
};
```

3.5 TCD_community

A TCD_community, estrutura concreta de dados que mantém registo dos dados lidos pelo parser, contém cinco parâmetros:

- **AVL**
- **USERS**
- **TAGS**
- **QUESTS**
- **ANSWERS**

Todos estes parâmetros são tipos abstratos de dados que "escondem"as reais implementações das estruturas necessárias para armazenar as informações utilizadas para responder às queries propostas.

```
struct TCD_community {
    AVL datesPosts;
    USERS users;
    QUESTS questions;
    ANSWERS answers;
    TAGS tags;
};
```

Iremos de seguida especificar a implementação dos tipos de dados abstratos que compõem esta estrutura global.

3.6 USERS

A struct users implementa uma *hashtable* criada para guardar os utilizadores da plataforma, de forma simples e clara. Nela, a cada chave - ID do utilizador, é associado uma **struct person**.

```
struct users{
    GHashTable* users;
};
```

3.7 DatesPosts

Esta estrutura representa a informação que estará presente em cada nodo de uma árvore binária de procura balanceada (AVL), que a seguir iremos apresentar. Nesta são guardadas dois apontadores para *GHashTables*, a implementação da *Glib* de uma *hashtable*. A GHashTable **questions** irá armazenar como chave, o id de uma pergunta e como valor o id do seu autor. Por outro lado, a GHashTable **answers** irá armazenar como chave o id da resposta e como valor o id do autor da mesma.

```
struct dateposts{
    GHashTable* questions;
    GHashTable* answers;
};
```

3.8 AVL

Uma AVL é uma árvore binária de procura balanceada, ou seja, uma árvore estruturada de forma a minimizar o número de comparações efetuadas no pior caso (procura de chaves com probabilidade de ocorrência idêntica). Como o trabalho com datas era recorrente, achamos que seria vantajoso criar uma estrutura em que a informação estivesse já ordenada cronologicamente, facilitando assim a sua procura. A nossa struct avl define então uma árvore balanceada (GTree), organizada por datas (a precisão temporal é referente aos dias, não sendo consideradas as horas ou minutos de diferença entre posts). Cada nodo guarda um apontador para uma estrutura DatesPosts, referida anteriormente, ou seja, organizando desta maneira a informação obtida, conseguimos saber quais foram os posts que aconteceram em determinado dia e quem foi o seu autor.

```
struct avl {
    GTree* avl;
};
```

3.9 TAGS

Com o objetivo de auxiliar a resposta à query 11, foi criada uma estrutura responsável por guardar toda a informação referente às **tags**. Tal é implementado com recurso a uma *HashTable*, utilizando como chave o nome da tag, e

tendo associado como valor um apontador para uma **struct tag** que guarda todas as informações da tag.

```
1 struct tags {
2     GHashTable* tags;
3 };
```

3.10 QUESTS

A struct quests é uma hashtable criada para armazenar todas as struct question resultantes do parse do input. Guarda-las nesta estrutura, utilizando como chave o identificador da pergunta e como valor a guardar o apontador para a struct question, permite otimizar o tempo de acesso à memória, pois conseguimos obter facilmente o id da pergunta, que é a chave na hashtable, quer como argumento nas queries, quer utilizando a struct AVL ordenada por datas.

```
1 struct quests {
2     GHashTable* questions
3 };
```

3.11 ANSWERS

A struct answers é uma hashtable criada para armazenar todas as struct answer resultantes do parse do input. Para as inserir na estrutura é fornecida como chave o id da resposta e como valor o apontador para a struct answer. Tal como as perguntas, também conseguimos obter o id da resposta, como argumento nas queries ou utilizando a struct AVL.

```
1 struct answers {
2     GHashTable* answers;
3 };
```

4 Parser

Nesta secção iremos falar sobre um dos passos mais importantes na concretização deste projeto, o parser. Para a realização do mesmo, usamos funções já fornecidas pela biblioteca libxml2.

É nesta fase que toda a informação contida nos dumps é lida, analisada, agrupada e posteriormente carregada em diferentes estruturas de dados, consoante o tipo de informação lida. Desta maneira, estamos perante a query que mais tempo demora a ser efetuada, por razões óbvias e totalmente compreensíveis.

Efetuamos três funções que juntas fazem o parse total dos dumps, ou seja, captam todos os dados relevantes para respondermos com sucesso a todas as queries propostas no enunciado. De todo o dump fornecidos concluímos que apenas era necessário parsar os ficheiros de input Users.xml, Posts.xml e Tags.xml.

5 Queries

Após termos efetuado com sucesso o parse e o carregamento dos dados para todas as estruturas que definimos, estamos agora em condições de avançar para a realização das queries propostas.

5.1 Query 1 - info_from_post

Numa primeira fase é necessário verificar se o id fornecido como argumento corresponde a uma pergunta ou resposta. Caso seja um pergunta, acedemos à estrutura quests e com o id fornecido é retirado o apontador para a struct question, como esse apontador podemos agora retirar o titulo e id do autor que a realizou. Com esse id, é possível aceder à estrutura users em que conseguimos o apontador para a estrutura person e assim obter o nome do autor da pergunta. Caso, o id fornecido seja uma resposta, temos que aceder à estrutura answers e com o apontador para a estrutura answer retirar o idpai que diz respeito ao id da pergunta à qual a resposta corresponde. Após isso são efetuados os passos descritos anteriormente para o caso das perguntas.

5.2 Query 2 - top_most_active

Esta query é respondida, em parte, enquanto o parse dos posts é efetuado. Cada vez que é encontrado um post durante o parse, surge também o id do autor do post. Com esse id vamos à estrutura users, que é a primeira é ser criada e após obter o apontador para a estrutura person, incrementamos o valor do número de posts do utilizador. Após o parse estar completo, temos apenas que iterar a hashtable cujo apontador está presente na estrutura users e inserir ordenadamente, pelo número de posts, numa estrutura à parte, o id dos users. É de salientar que iterando uma hashtable, os ids com o mesmo número de posts podem aparecer em posições distintas.

5.3 Query 3 - total_posts

Para responder a esta query, que recebe valores arbitrários de datas, utilizamos a estrutura avl em que, numa primeira fase, verificamos se as datas estão bem, ou seja, a data final (end) ocorre cronologicamente depois da data inicial (begin). Após esta validação vamos iterando, decrementando a data end até chegar à data begin. Em cada iteração são efetuadas duas funções, que verificam o tamanho das hashtables apontadas pelas variáveis presentes em cada estrutura dateposts, isto só acontece se essa estrutura for encontrada na GTree

pela função "get_node()". Recorremos a duas variáveis distintas às quais somamos o seu valor na iteração anterior e o valor obtido pelas funções na iteração atual. Após esse ciclo, estamos perante os valores finais e temos apenas que os transformar no output.

5.4 Query 4 - questions_with_tag

Tal como a query anterior, também esta recebe valores arbitrários de datas, logo o principio é exatamente o mesmo, iterando e decrementando a data no final de cada iteração de modo a percorrer todos os nodos da GTree presente na estrutura avl, sendo estes referentes a datas entre as datas fornecidas como argumento. Contudo, nesta query, em cada iteração é efetuada uma função que itera toda a GHashTable apontada pela variável questions presente na estrutura datesposts, e verifica, acedendo à estrutura quests com o id da pergunta encontrado, se essa pergunta contém a tag que é fornecida como argumento da query. Caso contenha a tag indicada, o id da pergunta é guardado num array dinâmico por nós definido, pois não sabemos qual o tamanho do output que nos espera.

Por fim, convertemos esse array dinâmico para o tipo de output esperado.

5.5 Query 5 - get_user_info

Para resolver esta query, dividimos o problema em duas partes. Na primeira parte temos que obter o aboutme do utilizador, logo acedemos à estrutura users e com o id retiramos o apontador PERSON. Através desse, obtemos o respetivo aboutme. Após este processo inicial, temos que descobrir quais os 10 últimos posts que o utilizador efetuou. Para tal, efetuamos um *foreach*, função presente na glib que itera toda a GTree apontada na estrutura avl, por ordem, neste caso por data. Iterando a árvore aplicando uma função em cada nodo, que é uma estrutura datesposts, conseguimos saber os posts em que este utilizador participou iterando as hashtables dentro de cada estrutura datesposts, comparando o valor lá presente com o id do utilizador. Caso isso seja verdade, o id do post é guardado na primeira posição de um array e todos os restantes são movidos uma posição para a direita. Como o foreach vai da data mais antiga para a mais recente conseguimos obter os posts por ordem cronológica inversa. Depois é só transformar os resultados obtidos nos tipos de output esperado.

5.6 Query 6 - most_voted_answers

Esta query segue o mesmo principio que a query 3 e 4, que recebem datas arbitrárias e iteram a estrutura avl decrementando a data no final de cada iteração até chegar à data de inicio. De salientar que em cada iteração, tal como na query 3 e 4, é feito um "get_node()", para ir buscar o valor presente na GTree, que tem como chave a data atual na iteração. Nesta query em especifico é feita um função que, em cada iteração do ciclo principal itera a GHashTable apontada por answers presente na estrutura datesposts e vai verificar o número

de votos de cada resposta à estrutura answers obtendo primeiro o apontador para a estrutura answer. A função insere ordenadamente num array os ids das respostas com mais votos. Após isso é apenas necessário transformar o array no tipo de output pretendido.

5.7 Query 7 - most_answered_questions

Esta query é praticamente igual à query 6. A única diferença é que na estrutura datesposts é iterada a GHashtable apontada pela variável questions e com o id da pergunta vai à estrutura quests retirar o apontador para a estrutura question. Com esse apontador é verificar o número de respostas que a question tem e posteriormente inserir ordenamente os ids das perguntas com mais respostas num array. Tal como a query 6, também é necessário no final transformar o array no tipo de output pretendido.

5.8 Query 8 - contains_word

Para responder a esta query temos de utilizar o mesmo principio utilizado na query 5 para descobrir os últimos posts. Aplicamos um foreach na GTree apontada na estrutura avl, que a cada nodo itera a GHashTable apontada na estrutura datesposts pela variável questions e com o id da pergunta obtido através da iteração da mesma, acede à estrutura quests. Com o resultado obtido, conseguimos extrair o titulo e aí sim, comparar com a palavra fornecida como argumento da query. Caso essa palavra exista no titulo, o id da pergunta é guardado num array em que é inserido na primeira posição e os restantes movidos uma posição para a direita. Como estamos a utilizar o foreach, obtemos o resultado cronologicamente inverso.

5.9 Query 9 - both_participated

A resolução desta query apresentou um maior nível de complexidade. Para realizar a query, efetuamos um foreach na GTree presente na estrutura avl e, a cada nodo que é uma struct datesposts, iteramos primeiro a GHashTable que a variável questions aponta. Quando encontramos uma pergunta que foi efetuada por algum dos dois utilizadores, inserimos esse id da pergunta num array dinâmico que está associado a esse utilizador. Posteriormente, iteramos a GHashTable que a variável answers aponta e se encontrarmos alguma resposta que foi efetuada por um dos dois utilizadores, acedemos à estrutura answers e com o id dessa resposta obtemos o apontador para a estrutura answer, podendo assim obter o id da pergunta à qual ela responde. Depois verifica-se se esse id da pergunta está presente no array dinâmico do outro utilizador, se estiver é porque ambos participaram nessa pergunta; logo o id da pergunta é guardado num array na primeira posição e os restantes elementos movem-se um indice para a direita.

Caso o id da pergunta não exista no array dinâmico do utilizador oposto, esse id será inserido no array dinâmico do utilizador visto ele ter participado nessa

pergunta. Utilizando o foreach voltamos a garantir que os resultados aparecem por ordem cronológica inversa. Após tudo isso, apenas é necessário transformar o array no tipo de output necessário.

5.10 Query 10 - better_answer

Esta query é praticamente respondida através de processamento no parser. Durante o parse de uma resposta, obtemos todas as variáveis necessárias para o cálculo da fórmula. Caso esse valor seja maior que o valor da pontuação da melhor resposta que a pergunta à qual a resposta a ser parsada pertence, é feita a mudança. O id da melhor resposta presente na struct question passa a ser o da resposta que está a ser parsada e a pontuação também. Depois na query em si, é só aceder à estrutura quests e com o id fornecido retirar o apontador para a estrutura question e posteriormente obter a melhor resposta.

5.11 Query 11 - most_used_best_rep

Há três fases para responder a esta query. Numa primeira fase, temos que iterar a estrutura users e criar um array ordenado de tamanho N com os N utilizadores com melhor reputação. Após essa fase, efetuamos um passo semelhante ao da query 6, em que iteramos a GTree presente na estrutura avl desde end até begin, e em que no final de cada iteração a data decrementa. Durante essa iteração, é feita uma função que itera a GHashTable apontada pela variável questions presente na struct datesposts, e se encontrar alguma pergunta efetuada por um dos utilizadores presentes no array acima descoberto, com o id da pergunta, acede-se à estrutura quests e retira-se o apontador para a estrutura question. Descobrindo o número de tags da pergunta e o array onde elas estão guardadas, iteramos esse array e a cada tag descoberta, acedemos à estrutura tags com a tag anteriormente descoberta como chave e obtemos o apontador para a estrutura tag. Com esse apontador, fazemos set do contador que incrementa o contador presente na estrutura tag em um.

A partir deste momento podemos efetuar a terceira fase, em que iteramos a GHashTable apontada na estrutura tags e ordenamos os ids das tags num array consoante o seu contador. Após essa inserção, o valor do contador é de novo colocado a 0.

Por fim, temos apenas que transformar este ultimo array no tipo de output pretendido.

6 Conclusão

Ao longo deste trabalho acabamos por perceber as dificuldades associadas à gestão e análise grandes quantidades de dados, e o esforço necessário para que essa gestão seja o mais eficiente possível. Desde arranjar estruturas não muito complexas para cada tipo de informação, formular um *parse* que permitisse mostrar e gerir os dados de forma aceitável, até ao criar duma interface que pudesse resolver as interrogações pretendidas.

Por fim, assumimos que o desenvolvimento deste projeto nos ajudou a evoluir bastante as nossas capacidades enquanto programadores, nomeadamente no que respeita à linguagem C, e a perceber a importância da modularidade em projetos de larga escala.