Projeto de Laboratórios de Informática 3
Grupo 34

Alexandre de Freitas Ferreira Pacheco A80760
Diogo José Cruz Sobral A82523
José Pedro Milhazes Carvalho Pinto A80741

5 de Maio de 2018

A80760

A82523

A80741

Resumo

Nesta unidade curricular foi-nos proposta a implementação de um sistema de resposta a *queries* sobre um *dump* da base dados do site *Stack Overflow*.

A primeira fase deste projeto consistiu no desafio de implementar este sistema na linguagem *C*. Era permitido (e recomendado) o recurso a bibliotecas como, por exemplo *libxml2* ou a *glib*. Apesar da intenção de tornar rápida a execução das funcionalidades do programa a ser desenvolvido, alguns dos focos prioritários deste projeto foram a modularidade e o encapsulamento das estruturas de dados por nós utilizadas.

Note-se que estas preocupações vão de encontro à aprendizagem realizada neste semestre no âmbito da programação orientada aos objetos, bem como ao objetivo da segunda fase do projeto, que consiste na implementação do mesmo sistema na linguagem *Java*.

Ao longo do desenvolvimento deste projeto, consideramos que os maiores desafios foram implementar as estruturas onde organizar a informação, e, sobretudo, encontrar o *sweet spot* do compromisso segurança vs. rapidez.

1 Ferramentas Utilizadas

Como referido, este projeto, desenvolvido em C, podia ser auxiliado por diversas ferramentas.

Em relação ao parsing dos ficheiros *.xml* sob a forma dos quais era disponibilizado o *dump*, optámos por utilizar a biblioteca *libxml2*, por uma questão de comodidade.

Para construir as nossas estruturas de dados, decidimos criar os nossos próprios módulos em vez de utilizar totalmente *glib*, a fim de obter ferramentas que, embora não tão robustas quanto as desta biblioteca, apresentam todas as características necessárias neste projeto e uma simplicidade que se traduz numa vantagem no que toca à rapidez em algumas operações.

Ainda assim, recorremos ao *glib* para implementar uma pequena parte da nossa estrutura de dados e para nos ajudar a responder algumas *queries*.

2 Tipos e Estruturas de Dados

De modo a conseguir responder às *queries* propostas de forma eficiente, é natural que o mais importante é a forma como organizamos o grande volume de informação presente nos *dumps* disponibilizados.

Primeiro, analisámos a informação contida em cada um dos ficheiros *xml* disponibilizados por comunidade (e.g. *android*, *ubuntu*...) e o que era pedido nas *queries*, a fim de perceber quais os dados que seriam recorrentemente necessários.

2.1 Tipos de Dados: *Users* e *Posts*

As entidades elementares na forma de descrever cada comunidade, à volta das quais gira a grande maioria da informação, são os *users* e os *posts*. Naturalmente, para cada um destes objetos criámos um tipo de dados próprio, `MYUSER` e `MYPOST`.

```
struct mypost {
    long        id;
    int         typeid;
    long        parent_id;
    MYDATE      cdate;
    int         score;
    long        ownerid;
    char *      ownername;
    char *      title;
    char **     tags;
    int         anscount;
    int         commcount;
    int         favcount;
    STACKPOST   filhos;
    int         type;
};
```

```
typedef struct mypost * MYPOST;
```

```
struct myuser{
    long id;
    int rep;
    char * username;
    char * bio;
    int type;
    STACKPOST posts;
};
```

```
typedef struct myuser * MYUSER;
```

```
struct stackpost {
    long counter1;
    long counter2;
    long n_elem;
    long size;
    MYPOST * array;
};
```

```
typedef struct stackpost * STACKPOST;
```

2.2 Organização das Estruturas Elementares

Criados os tipos de dados dos *users* e *posts*, procedemos à implementação da organização dos mesmos, fator determinante na viabilidade e qualidade deste projeto.

Decidimos organizar os *users*, bem como os *posts* em árvores binárias de procura balanceadas. Optámos ainda por utilizar duas árvores para organizar os *posts*, uma em que as chaves de ordenação são o *Id* do *post*, e outra em que as chaves são as datas de criação (e em que em cada nodo da árvore há um *array* de *posts* criados no mesmo dia).

É importante mencionar que os nodos das árvores não contêm a informação que nelas se organiza, propriamente dita, mas sim um apontador para as estruturas elementares que a armazenam, e permitindo assim a partilha dessas "cápsulas de informação"entre diversas estruturas criadas para as organizar e percorrer de forma diferente.

2.3 Outras Estruturas Auxiliares

Para além da organização essencial referida, decicidmos implementar, ainda, estruturas auxiliares, que são úteis sobretudo quando a resposta a determinadas *queries* envolve calcular um conjunto de perguntas mais respondidas ou um conjunto de utilizadores com maior reputação.

Duas destas estruturas são *heaps*. Uma armazena as reputações de todos os users, outra armazena o número de posts de cada user. Estas *heaps* são criadas e preenchidas logo após a construção das árvores referidas anteriormente.

Outras duas estruturas são *stacks* que estão de certa forma relacionadas com as *heaps*. Estas são apenas construídas quando é necessário calcular um conjunto de N utilizadores com maior reputação, ou com maior número de posts. Após serem construídas, ficam guardadas e podem crescer se for necessário obter novamente um conjunto semelhante mas com um número maior que N. Assim, há cálculos que após serem efetuados uma vez, deixam de ter de ser efetuados, na mesma *query* e mesmo entre *queries* diferentes.

Utilizámos ainda uma tabela de *hash* da biblioteca *glib*, que armazena as *tags* como chaves e os `Ids` das `tags` como valor correspondente.

```
struct TCD_community {
    TREE users;
    TREE posts_Date;
    TREE posts_Id;
    HEAP num_posts;
    STACK pre_posts;
    HEAP rep_users;
    STACK pre_rep;
    GHashTable * tags;
};
```

`typedef struct TCD_community * TAD_community;`

```
struct generico {
    long key;
    long data;
};

struct heap{
    long size;
    long n_elem;
    GEN * array;
};
```

`typedef struct heap * HEAP;`

A nossa `HEAP` é uma *max-heap*, no entanto, construímo-la de modo a que, nos casos em que dois elementos da *heap* têm chaves iguais, estes estão ordenados pelo seu valor `data` de forma crescente. Por exemplo, se estamos a organizar `Ids` de *tags* pelo número de ocorrências (como na *query* 11), e as *tags* 10 e 20 têm as mesmas ocorrências, então a 10 fica acima da 20.

3 Modularização e Abstração de Dados

Como temos aprendido ao longo dos últimos tempos, são boas práticas as de manter um código modular e abstraído do tipo de dados com que trabalhamos.

Encapsulamento

Seguindo este pensamento, separámos as partes funcionais do nosso programa em diferentes módulos e deixámos o mínimo de funções visíveis para o exterior. As funções que podem ser chamadas de fora do módulo (por exemplo os *"gets"*) devolvem sempre uma cópia da informação contida na nossa estrutura, de modo a não comprometer o encapsulamento dos nossos tipos de dados e, por conseguinte, a integridade da estrutura da montada.

```
char * getOwnerNameP(MYPOST post){
        return post ? mystrdup(post->ownername) : NULL;
}
```

Exemplo de método *get* a clonar informação.

Um sistema de proteção que implementámos consistiu em ter uma variável dentro de cada tipo (por exemplo um `MYPOST` ou `MYUSER`) que especifica se a *struct* em questão é ou não uma cópia. Se for uma cópia pode ser libertada normalmente, mas se for a instância "original"(a construída no `load`), apenas pode ser libertada por funções que não são visíveis para fora (e que apenas são chamadas na função `clean`.

Para além disso, tivemos também o cuidado de escrever código "neutro"em relação a tipo de dados. A nossa implementação de árvores binárias de procura balanceadas, por exemplo, trabalha com chaves e valores do tipo `void *` de modo a podermos a qualquer altura mudar a organização da árvore, ou criar uma nova, apenas escrevendo uma função de comparação (para as *keys*) e funções para libertar os dados usados.

```
struct AVBin {
    int altura;
    void * key;
    void * data;
    struct AVBin * esq, * dir;
};

typedef struct AVBin * AVL;

struct tree{
    AVL arv;
    long nnodes;
    void * (*replace_fun)(void *,void *);
    int (*f_compare)(void *,void *);
    void (*destroy_key)(void *);
    void (*destroy_data)(void *);
    int heigth;
};
```

```
typedef struct tree * TREE;
```

4 *Queries*

Nesta secção passamos a explicar a abordagem que tomámos em relação a cada uma das *queries*.

Query 1 - Título e autor da pergunta

O procedimento que tomamos é bastante simples.

1. Procuramos o post com o `Id` fornecido e verificamos se se trata de uma pergunta.
   (a) Se for uma pergunta, devolvemos o seu título e o `DisplayName` do autor (procurado o `OwnerUserId` na árvore de users).
   (b) Se for uma resposta, repetimos o ponto **1** com o `ParentId` do post, que será o `Id` da pergunta correspondente.

Query 2 - Top N *users* com mais posts

Esta query toma partido da heap de utilizadores ordenados pelo seu número de posts que construímos na `load` e da *stack* de `Ids` de users organizados pelo número de posts que preenchemos à medida que as *queries* chamadas necessitam.

1. Inicializa-se a `LONG_list` com N elementos e verifica-se se a *stack* de `Ids` de utilizadores organizada por número de posts tem N ou mais elemntos.
   (a) Se tiver N ou mais elementos, então, a essa *stack* é percorrida e preenche-se a lista resultado.
   (b) Se tiver menos de N elementos, então percorre-se os elementos que esta já tem preenchendo-se a lista resultado, e depois, até haver N elementos na *stack*, faz-se *pop* à *heap* e inserem-se os elementos que vão sendo colocados na *stack* também na lista resultado.

Query 3 - Número de perguntas e respostas ao longo de um período

Como já referido, a árvore que organiza os *posts* por data, em cada nodo, tem um array de posts. Para além do array propriamente dito, com apontadores do tipo `MYPOST`, há dois contadores: um para o número de respostas, e outro para o número de perguntas nesse array (calculados no `load`). Ora, é desta organização que esta *query* toma partido.

1. Percorre-se os nodos da árvore de posts dentro do período especificado.

   Em cada nodo da árvore, soma-se os respetivos contadores à variável cujo endereço é passado como argumento pela função de travessia.

Query 4 - Perguntas com determinada *tag* feitas num período

1. Percorre-se todos os nodos da árvore de posts dentro do período especificado.

   Percorre-se o *array* de posts em cada nodo, e em cada post verifica-se se este é uma pergunta e contém a tag especificada. Se for esse o caso, adiciona-se o `Id` a um *array* resultado (de onde posterioremente se passarão os valores para `LONG_list`)

Query 5 - Informação de um *user* e os seus últimos *posts*

Esta *query* consiste em simplesmente procurar um utilizador na nossa estrutura e devolver a sua informação. O tipo de dados `MYUSER` contém, como já foi visto, um *array* com os `Ids` dos seus posts,o que facilita imenso esta resposta.

Query 6 - N respostas mais votadas ao longo de um período

A diferença entre os *upvotes* e *downvotes* de uma resposta equivale ao seu parâmetro `Score`, pelo que este já se encontra calculado a partir do momento em que o carregámos do ficheiro *xml*.

1. São visitados todos os nodos da árvore de *posts* dentro do período especificado, percorrendo o *array* de *posts* em cada nodo e inserindo numa `HEAP` (*max-heap*) o par de informação {`Id`, `Score`}.

2. Dá-se *pop* a N elementos da heap e preenche-se a lista resultado com os `Ids` das respostas a ser retiradas da *heap*.

Query 7 - N perguntas com mais respostas ao longo de um período

Dado que as respostas às perguntas a devolver têm de ter sido feitas ao longo do período especificado, esta *query* não se resume a consultar o número de respostas em cada pergunta no período de tempo, mas a calcular o número de respostas feitas ao longo desse período para cada pergunta.

1. Percorre-se todos os *posts* criados no período de tempo dado.

   (a) Percorre-se a lista de filhos desse *post* (que só existe se este for uma pergunta) e conta-se quantos foram criados no período de tempo especificado.

   (b) Insere-se numa `HEAP` (*max-heap*) o par de informação {`Id`, número de respostas}.

2. Faz-se *pop* a N elementos da *heap*, guardando na lista resultado os `Ids` das perguntas nas condições especificadas.

Query 8 - N perguntas mais recentes com determinada *tag*

A query 8 não exige que todos os *posts* sejam consultados, uma vez que estes se encontram organizados por tempo, e assim, podemos obter tudo o que precisamos visitando o número mínimo de posts.

O critério que utilizámos para determinar se a palavra ocorre ou não em cada título (para além de obviamente essa sequencia de caracteres aparecer na *string* do título) foi a existência de espaços, pontuação, *newline* ou `EOF` nas extremidades da *substring* com a palavra. Para nos facilitar este trabalho utilizámos as funções de biblioteca `ispunct()` e `isspace()`.

1. Efetua-se uma travessia semelhante à *inorder* na ordem "posts recentes - posts antigos". Que termina quando todos os nodos da árvore tiverem sido percorridos ou quando obtivermos N *posts* nas condições pretendidas.

   (a) Em cada nodo, percorre-se o array de `MYPOST`, e em cada *post* verifica-se se este contém a *tag* especificada. Sendo esse o caso, adiciona-se o `Id` do *post* a um *array* (que posteriormente é passado para a `LONG_list` resultado).

Query 9 - N perguntas mais recentes em que dois *users* participaram

Esta é outra query em que se tira bastante partido do facto de termos, no `load`, carregado para o tipo `MYUSER` um array `STACKPOST` com os posts de cada *user*.

1. São procurados os dois *users* a partir dos `Ids` especificados e verifica-se qual dos dois tem menos posts.

2. São percorridos os *posts* desse user e são inseridos numa tabela de *hash* os `Ids` das perguntas relativas a cada post (ou seja, o próprio `Id` se o *post* for uma pergunta, ou o `ParentId` se o mesmo for uma resposta).

3. É repetido o processo para o outro utilizador, mas em vez de se inserir na tabela *hash* os `Ids`, verifica-se se estes estão já inseridos na tabela. Aqueles que já estiverem são constituintes do reultado, sendo armazenados num *array* `STACKPOST`.

4. Os elementos deste *array* são ordenados segundo um algoritmo *quicksort* (segunda a sua `CreationDate`, e os N mais recentes são passados para a `LONG_list` resultado.

Query 10 - Melhor resposta

Esta *query* é relativamente simples, devido ao facto de no `load` armazenarmos um *array* `STACKPOST` que guarda as respostas a cada pergunta.

1. É procurada a pergunta na árvore ordenada por `Ids`.

2. É percorrido o `STACKPOST` das respostas à pergunta e para cada um deles é calculada a pontuação (segundo a fórmula especificada no enunciado) e registado a melhor pontuação bem como a respetiva resposta, cujo `Id` é depois retornado.

Query 11 - N *tags* mais usadas ao longo de um período pelos N *users* com maior reputação

Esta *query* faz uso de grande parte das estruturas que temos montadas, bem como de estruturas criadas ao longo da sua execução.

Nesta query, é criada uma tabela de *hash* com as ocorrências de cada *tag*. É ainda criada uma `HEAP` (*max-heap*) onde depois são inseridos os `Ids` das *tags* e as suas ocorrências como chave, de modo a ficarem ordenados. É ainda importante mencionar que, na nossa interpretação do que foi pedido, utilizámos os N utilizadores com maior reputação de sempre (mesmo que não tenham postado nesse período de tempo). No resultado da query, `Ids` de tags com o mesmo número de ocorrências estão em ordem crescente.

1. É obtido um array com os `Ids` dos N *users* com maior reputação (procedendo da mesma forma que na *query* 2, mas em relação à reputação e não ao número de posts).

2. O array obtido é percorrido, procurando cada `Id` na árvore dos *users* e obtendo o `STACKPOST` com os *posts* criados por cada um desses utilizadores.

   (a) Se o *post* for uma pergunta e tiver sido criado dentro do intervalo de tempo especificado;

   (b) Percorre-se cada *tag* desse *post* e, consultando a tabela de *hash* pré-calculada, obtém-se o `Id` da mesma.

   (c) Se este `Id` já constar na outra tabela de *hash* que faz corresponder cada `Id` ao seu número de ocorrências, é incrementado esse número. Caso contrário, esse `Id` é inserido.

3. Preenchida a tabela *hash* que a cada `Id` faz corresponder o seu número de ocorrrências, percorre-se todas as entradas da mesma e insere-se o par {número de ocorrências, `Id`} numa *max-heap*, ficando assim os `Ids` ordenados.

4. É feito *pop* de N elementos da `HEAP`, e preenche-se a `LONG_list` resultado com os `Ids` desses elementos.

5 Estratégias para melhorar a Eficiência

Tendo em conta o grande volume de dados que nos propusemoss a processar neste projeto, surge a necessidade de adotarmos estratégias que melhorem a eficiência das operações que levamos a cabo.

Uma prática que procurámos ter foi a utilização de arrays sempre que possível. Um bom exemplo da nossa "aproximação"aos *arrays* é a nossa implementação de *max-heaps* que utilizámos frequentemente para armazenar e retirar ordenadamente elementos para respostas a *queries*.

Outra forma de tornar o nosso programa mais eficiente foi, durante o `load`, construir estruturas auxiliares que, embora armazenassem informação que pudesse ser obtida a partir das árvores de *posts* e *users*, e levassem a um ligeiramente maior gasto de memória, eram úteis a muitas *queries* e poupavam imensos cálculos e travessias.

Aproveitar a organização das estruturas de dados para os percorrer de diferentes maneiras, consoante a necessidade, foi também uma opção que tornou a nossa resposta a certas *queries* mais eficiente, como por exemplo, a existência de várias travessias nas árvores binárias balanceadas de procura, ou, como já referido, a utilização da *max-heap* para criar mapas ordenados.

6 Conclusão

Atingidos os objetivos deste projeto, retivemos alguns prontos principais.

Primeiro, a necessidade de abdicar de performance para ter um código seguro, protegido não só do utilizador comum mas principalmente de terceiros que pretendam aceder e possivelmente modificar uma estrutura de dados.

Outra observação foi a importância da organização das estruturas de dados, a influência destas nos algoritmos a aplicar, e como diferentes combinações destes dois conceitos tiveram um impacto palpável nos tempos de execução das funcionalidades do programa.

Por fim, não pudemos deixar de notar no facto de que C, por ser uma linguagem de baixo nível de abstração, carece de funcionalidades que outras linguagens (*e.g. Java*, *Python*...) apresentam e facilitam o trabalho do programador (não tendo este de "reinventar a roda"). Assim, esperamos que a segunda fase deste trabalho (implementação em *Java*) seja mais simples de realizar, ainda que com uma penalização na performance.