Projeto de Laboratórios de Informática III

Grupo 41

Ana Ribeiro (A82474) Jéssica Lemos (A82061) Pedro Pinto (A82535)

abril de 2018

Conteúdo

1 Introdução

Este relatório aborda a implementação de um sistema capaz de processar ficheiros XML que armazenam as várias informações utilizadas pelo Stack Overflow na disciplina de Laboratórios de Informática 3 (LI3), do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Uma vez processada essa informação, pretende-se que seja possivel executar um conjunto de interrogações específicas, apresentadas posteriormente, de forma eficiente.

A substância do relatório centra-se nos seguintes aspectos:

- tipo concreto de dados
- estruturas de dados usada
- modularização funcional
- abstração de dados
- estratégias seguidas em cada uma das interrogações
- estratégias para melhoramento de desempenho

2 Concepção

2.1 Concepção do problema

Tendo em conta o problema apresentado, decidimos organizar o nosso trabalho em dois grupos:

- Posts
- Users

Para realizar as queries relativas aos posts é necessário obter do ficheiro *"Posts.xml"* as seguintes informações: o id e o tipo do post e no caso de este ser uma resposta guardar o id da respetiva pergunta, as tags, o seu título, o id do utilizador que o elaborou, a sua data de criação e o número de comentários. Caso o post seja uma pergunta terá de se guardar também o número de respostas. A fim de tornar a resolução das interrogações deste grupo eficientes, é evidente a necessidade de implementar uma estrutura onde seja rápida a procura dos posts de uma dada data. Acrescido a esta preocupação, é indispensável ter o cuidado de organizar os posts de uma data pelo seu id.

Quanto aos users, é oportuno guardar do ficheiro *"Users.xml"* o id, o nome, a reputação e a informação do seu perfil. Para tornar eficaz a resolução destas queries, é útil estruturar as informações pelo seu id.

Para as tags, é preciso armazenar do ficheiro *"Tags.xml"* o id e o seu nome, ordenadas por este.

2.2 Concepção da solução

Com o objetivo de solucionar todas as interrogações do grupo dos posts de modo eficiente elaboramos uma *Hash Table* organizada por datas, em que cada posição contém uma *Árvore Binária Balanceada* com todos os posts dessa data ordenada pelo seu id. A escolha da *Hash Table* deveu-se ao elevado número de interrogações que envolviam intervalos de tempo, para estas tornou-se imperativo evitar percorrer os posts que não tivessem incluídos nesse intervalo. Uma vez reduzida a procura, foi necessário organizar a *Árvore Binária Balanceada* pelo o id do post, dado que a maioria das interrogações se baseiam neste fator.

Para os users optamos por uma *Hash Table* organizada pelos ids dos utilizadores, dado que todas as queries deste grupo se baseiam no id. Houve também a necessidade de associar a cada user uma *MaxHeap* com todos os posts em que participou cujo fator de comparação é a data de criação, para a resolução das interrogações em que é estabelecida uma relação entre o id dos posts e o utilizador ordenados por cronologia inversa.

Tendo em conta que para o grupo das tags a procura se baseia no seu nome definimos uma *Hash Table* disposta por este. Para tal, somamos todas as letras que compunham o seu nome. Para auxiliar estes dois grupos foi fundamental recorrer a duas *MaxHeaps* para diminuir o tempo de resolução de interregações que envolviam tops. Uma destas incluía os utilizadores estruturados pela sua reputação e a outra engloba os utilizadores ordenados pelo número de posts realizados.

3 Estrutura de dados

O nosso tipo concreto de dados *TCD_Community* é composta por dez items:

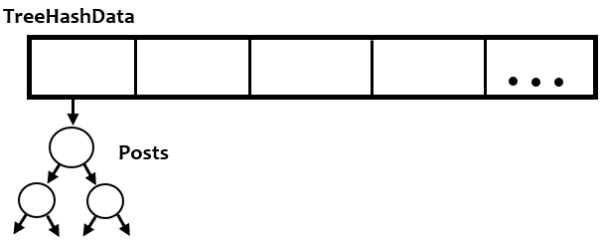
- tamanho de cada tabela de hash
- *Hash Table* que em cada posição contém uma *Árvore Binária Balanceada* dos posts
- *Hash Table* composta pela a informação utilizadores que em cada posição contém uma *MaxHeap* com os ids dos posts em que este participou
- *Hash Table* que possui a informação referente às tags
- *Array* que guarda os ids dos utilizadores com mais posts observados até ao momento
- *Array* que guarda os ids dos utilizadores com maior reputação verificados até ao momento
- *MaxHeap* que inclui os ids dos utilizadores ordenados pelo número de posts
- *MaxHeap* que engloba os ids dos utilizadores cujo fator de comparação é a reputação

```
typedef struct TCD_community{
    int dataSize;
    int usersSize;
    int tagsSize;
    TreeHashData treeHash;
    HashTableUsers hashUser;
    HashTableTags hashTag;
    long *topN;
    long *topNR;
    Heap Top;
    Heap TopR;
} TCD_community;

typedef struct TCD_community* TAD_community;
```

O nosso tipo abstracto de dados, *TAD_Community*, foi declarado no ficheiro .h como sendo um apontador para o nosso tipo concreto de dados. Deste modo, foi possível garantir que a implementação dos dados não se encontra visível para os utilizadores.

3.1 TreeHashData



No nodo de cada *Árvore Binária Balanceada* temos as informações necessárias para responder às interrogações referentes ao grupo dos posts.

```
typedef struct post{
    int postTypeId;
    long parentId;
```

```
    long id;
    char *tag;
    char *title;
    long ownerUserId;
    Date creationDate;
    int answerCount;
    int commentCount;
    int score;
    struct post *esq,*dir;
} Post;
```

Cada local da *Hash Table* é constituída pela árvore dos posts. Acrescentamos ainda três variáveis que contam respetivamente: o número de posts tipo resposta, o número de posts tipo pergunta e o número total de posts destes dois tipos.

```
typedef struct treeHash{
    long contadorR;
    long contadorP;
    int numRespostas;
    Post* tree;
} TreeHash;

typedef struct treeHash ** TreeHashData;
```

3.2 HashTableUsers

Em cada posição da *MaxHeap* existe a data que é o fator de ordenação e o id do utilizador, que é devolvido nas queries em que é utilizada.

```
typedef struct elemPosts{
    long id;
    Date data;
} elemP;

typedef struct heapPosts{
    int size;
    int used;
    elemP *array;
}* HeapPosts;
```

Em cada lugar da *Hash Table* incluimos as informações necessárias para as interrogações deste grupo e a *MaxHeap* referida anteriormente.

```
typedef struct users{
    long ownerUserId;
```

```
    char *displayName;
    int reputation;
    int nPosts;
    char *aboutMe;
    HeapPosts top10;
} Users;

typedef struct users ** HashTableUsers;
```

3.3 HashTableTags

Em cada lugar da *Hash Table* colocamos as informações necessárias para a resolução de uma das queries.

```
typedef struct tags{
    long id;
    char *tagName;
} Tags;

typedef struct tags **HashTableTags;
```

3.4 Heap

Em cada posição da *MaxHeap* encontra-se o uma variável que é o fator de ordenação, que depende do contexto em que é aplicada e o id do utilizador, que é devolvido nas queries em que é usada.

```
typedef struct elemento{
    long id;
    int count;
} elem;

typedef struct heap
    int size;
    int used;
    elem *array;
}* Heap;
```

4 Interrogações

4.1 Init

Na query init simplesmente alocamos espaço para a estrutura *TAD_Community*, inicializamos as *Hash Tables*, as *MaxHeap* e os *Arrays*, dado que é no load que iniciamos as estruturas que constituem a *TAD_Community* pois apenas temos acesso aos tamanhos necessários no *parse*.

4.2 Load

Esta interrogação é a mais demorada visto que é responsável por processar a informação dos ficheiros e armazená-la na nossa estrutura. Tendo em conta a variedade dos requesitos apresentados nas queries, optamos por estruturas mais complexas que se refletiram no tempo de load mas que se irá expressar numa maior efeciência na resolução das interrogações.

4.3 Info_from_post

Começamos por procurar o post na *TreeHashData* e caso este seja do tipo pergunta retiramos-lhe o título e id do utilizador. Com o id acedemos à estrutura dos users e obtemos o nome do utilizador. Na eventualidade do post ser do tipo resposta retiramos o id da pergunta a que corresponde e efetuamos o proceso anterior para este.

4.4 Top_most_active

Recorrendo à *MaxHeap* que se encontra ordenada pelo número de posts de cada utilizador, extraimos os N ids pedidos, guardamos no *Array* que está na *TAD_Community* e criamos a lista necessária para devolver. No caso do array já ter algum id, o que ocorre quando a query é invocada mais do que uma vez, aproveitamos os ids já existentes no array e caso necessário extraimos o que falta para completar o requerido.

4.5 Total_posts

Para esta query percorremos a *TreeHashData* e no caso de a data pertencer ao intervalo de tempo pertendido, somamos aos contadores do número de respostas e número de perguntas respetivamente, e de seguida criamos o par esperado com essa informação.

4.6 Questions_with_tag

Iniciamos por percorrer a *TreeHashData* e na hipótese de a data estar contida no intervalo de tempo desejado, vamos a todos os nodos da árvore e na eventualidade de o post ser do tipo pergunta, verificamos se a tag dada se encontra nas tags do post. Em caso afirmativo, guardamos o id num *array* ordenado por datas que no final será transferido para a lista solicitada.

4.7 Get_user_info

Nesta interrogação acedemos à posição onde se encontra o utilizador pretendido na *HashTableUsers* e obtemos a informação do seu perfil e extraimos da *MaxHeap* integrante os seus dez últimos posts.

4.8 Most_voted_answers

Começamos por percorrer a *TreeHashData* e na possibilidade de a data se enquadar no intervalo de tempo pedido, acedemos a todos os nodos da árvore e caso seja do tipo resposta inserimos o score e o id em dois *arrays* de tamanho solicidado na query e com fator de comparação o score. Desta forma, apenas são inseridos elementos que potencialmente poderão integrar-se no top N pretendido.

4.9 Most_answered_questions

Para esta query passamos por todas as posições da *TreeHashData* e no caso da data se encontrar no intervalo requerido, percorremos a estrutura dos posts calculando a chave para a *HashTableQuery7* onde inserimos uma flag para o caso do post ser do tipo pergunta e um contador para incrementar no caso do tipo resposta. De seguida, extraimos da *MaxHeap* criada com os elementos da *Hash Table* com flag a 1 (indica que o post é tipo pergunta) e o número de respostas feitas a esse post neste mesmo intervalo. Por fim, devolvemos os ids dos posts ordenados por cronologia inversa.

De modo a tornar mais eficiente a resolução desta query decidimos implementar uma *Hash Table* em que em cada posição guardamos o id do post tipo pergunta, bem como uma flag indicando se este se encontra neste intervalo de tempo e um contador com o número de posts tipo resposta. Desta forma, a inserção e posteriormente a procura na *Hash Table* para a inserção na *MaxHeap*

será muito mais rápida e eficaz do que seria com a utilização, por exemplo, de *arrays*, que para tamanhos consideravelmente grandes torna-se bastante ineficiente.

```
typedef struct query7{
    long id;
    int flag;
    int contador;
} Query7;

typedef struct query7 ** HashTableQuery7;
```

4.10 Contains_word

Passamos por todos os Posts existentes e na eventualidade de ser do tipo pergunta verificamos se a palavra dada se encontra no título. Em caso afirmativo, inserimos em dois *arrays* com o tamanho solicitado, o id e a data, ordenados por esta.

4.11 Both_participated

Começamos por aceder às posições da *HashTableUsers* dos utilizadores solicitados e retiramos as respetivas *MaxHeaps*. Extraimos os ids da maior das *MaxHeaps* para um *array*, mas caso o post a extrair seja do tipo resposta, guardamos o id da pergunta correspondente. De seguida, removemos os elementos da outra das *MaxHeaps* e verificamos se é do tipo pergunta ou tipo resposta. Na eventualide de ser do tipo pergunta, verificamos se o id deste se encontra no *array* criado anteriormente. Caso contrário, retiramos o parentId e utilizamos este como fator de comparação.

4.12 Better_answer

Nesta query, percorremos a *TreeHashData* e os nodos da *Árvore Binária Balanceada* até encontrar o post requerido e retiramos o número de respostas existente. Depois voltamos a percorrer a *TreeHashData* a partir dessa posição e sempre que encontramos um resposta ao post vamos buscar as informações necessárias para obter a média ponderada e decrementamos ao número de respostas existentes, pois deste modo evitamos continuar a percorrer a estrutura quando já não houverem mais respostas à pergunta. Sempre que encontramos uma média maior do que a guardada até ao momento, substituimos esse valor e atualizamos o id. Na possibilidade de chegarmos ao fim da hash e ainda existirem respostas retomamos a procura desde o início. No caso de o id dado corresponder a um post resposta o número de respostas será zero e não efetuará qualquer cálculo.

4.13 Most_used_rep

Iniciamos por preencher o *array* com os ids dos utilizadores com melhor reputação que faltam para obter o top N pretendido e transferimo-los para a *HashTableTopN*, pois a procura de um utilizador num *array* consideravelmente grande seria muito lenta. De seguida percorremos a *TreeHashData* e se a data da posição pertencer ao intervalo de tempo, acedemos aos nodos da árvore e para cada um verificamos se o utilizador está incluido no top N estabelecido anteriormente. Caso tal se suceda, para todas as tags desse post vamos verificar se o seu id já se encontra na *HashTableQuery11* e caso esteja incrementamos o número de vezes que esta ocorre. Caso contrário, vamos procurá-la à estrutura das tags e inserimos o seu id na nossa *Hash Table*. Posteriormente os elementos da *Hash Table* são inseridos numa *MaxHeap* de modo a obtermos as N tags mais usadas pelos N utilizadores com melhor reputação.

Na resolução desta query seguimos o raciocínio da interrogação 7, sendo que nesta *Hash Table* decimos em cada posição guardar o id da tag de modo a não procurar a mesma na *HashTableTags* repetidamente, a própria e um contador com o número de vezes que esta é utilizada. Como já

referido, a inserção e posteriormente a procura será muito mais rápida e eficaz.

```
typedef struct query11{
    long id;
    char* tag;
    long contador;
} Query11;

typedef struct query11 ** HashTableQuery11;
```

4.14 Clean

Percorremos todas as estruturas criadas libertando as informações para o qual foi alocada memória. Tendo ainda o cuidado de libertar os arrays.

5 Modularidade

Dada a dimensão do projeto, foi imperativo organizá-lo de modo a torná-lo monitorizável e legível. Para tal, optamos por dividi-lo em módulos, sendo os fundamentais o *parse* e a *struct*. No *parse* é realizado o *parsing* dos ficheiros XML utilizando a biblioteca *libxml*. Na *struct* inicializamos, carregamos e libertamos as estruturas com os dados provenientes do *parse*.

Quanto às interrogações estruturamo-las em dois grupos: users e posts. Sendo que as estruturas auxiliares encontram-se num novo módulo, a *auxiliary*.

6 Melhoramento de desempenho

As *flags* do tipo *-O* definem o nível de otimização do compilador. Neste caso, optamos por utilizar a *flag -O3* de modo a melhorar o desempenho do código resultando num melhor tempo de execução. Testamos também a *flag -O2* para comparar o tempo de execução com as diferentes *flags* e verificamos que com a *flag -O3* o desempenho era consideravelmente melhor.

7 Conclusão

Neste trabalho foi notória a importância da escolha das estruturas a utilizar, uma vez que estas têm influência direta no desempenho das interrogações requeridas. Para desenvolver este projeto optamos por estruturas mais complexas que demoram mais tempo a serem carregadas, mas que permitem um melhor tempo de execução das queries. O facto de cada interrogação poder ser invocada mais do que uma vez contribuiu para esta decisão.

Com o objetivo de melhorar o tempo de load, ao longo do trabalho pensamos em alterar as estruturas. No entanto, na resolução das queries verificamos que uma estrutura menos bem conseguida tinha um impacto negativo no seu desempanho.

Em última instância, consideramos que os nossos objetivos neste projeto foram alcançados, contudo ainda podiam ser melhorados alguns aspetos.