Projeto de LI-3

Martins, José(a78821) Costa, Mariana
Quaresma, Miguel(a77049)

April 26, 2017

Contents

Part I

Introdução

Projeto realizado no âmbito da cadeira de LI3 que consiste no desenvolvimento, em C, de um programa que responda a um conjunto de *queries* relativas a *snapshots* do site Wikipédia. O produto final deverá ter em conta não só a correção das suas respostas às queries mas também o tempo que demora a obter as mesmas. As queries às quais era necessário dar resposta são as seguintes:

1. long all_articles(TAD_istruct qs);
2. long unique_articles(TAD_istruct qs);
3. long all_revisions(TAD_istruct qs);
4. long* top_10_contributors(TAD_istruct qs);
5. char* contributor_name(long contributor_id, TAD_istruct qs);
6. long* top_20_largest_articles(TAD_istruct qs);
7. char* article_title(long article_id, TAD_istruct qs);
8. long* top_N_articles_with_more_words(int n, TAD_istruct qs);
9. char** titles_with_prefix(char* prefix, TAD_istruct qs);
10. char* article_timestamp(long article_id, long revision_id, TAD_istruct qs);

Part II

Desenvolvimento

0.1 Bibliotecas

0.1.1 libxml2

Na realização do trabalho foi utilizada a biblioteca *libxml2*, por forma a automatizar o parsing de ficheiros xml.

0.2 Filosofia de desenvolvimento

O código foi desenvolvido tendo em mente que a maior parte do trabalho podia ser feito na função load permitindo assim a minimização do tempo de resposta às queries mas mantendo, ainda assim, o seu tempo de execução baixo. Para isto foi tido em conta a possibilidade de que nem todas as queries seriam efetuadas e, por isso, apenas foram calculados os valores que tivessem o mínimo de impacto adicional na performance , isto é, valores que podiam ser calculados pela simples travessia da árvore gerada pela biblioteca libmlx2. Sendo assim, queries de ordenação com base em valores que não os usados para esse efeito (id de contribuidor vs nro de contribuidor do mesmo) são calculadas quando chamadas enquanto que queries em que a resposta são valores que podem ser calculados pelo simples processamento dos ficheiros xml possuem os valores já calculados (nro de artigos totais/unicos) ou as estruturas optimizadas para a execução das mesmas (consulta de nome de contribuidor).

0.2.1 Estruturas de Dados

Na escolha das estruturas de dados a utilizar optámos por estruturas que nos oferecessem um tempo de consulta/inserção/remoção mais detrimento da quantidade de memória utilizada. Exemplo disto, e como iremos referir, é a estrutura utilizada para armazenar informação relativa aos contribuidores que, a troco da quantidade de memória ocupada, permite, na maioria dos casos, manter os tempos de consulta/inserção/remoção na $O(\log n)$. Esta opção foi tomada na assunção de que, ainda que o hiato processador-memória seja um dos grandes *bootlenecks* atuais, a quantidade de cache existente na maioria dos processadores atuais já permite o uso de estruturas como estas sem que haja uma perda de performance em relação a estruturas mais simples.

Estrutura Principal(struct TCDistruct)

```
struct TCD_istruct {
    long artUn, artTot;
    articTableP articCollect;
    contribTreeP contribuitors;
};
```

A estrutura principal(struct TCDistruct) utilizada consiste em duas subestruturas, uma tabela de hash que guarda informação sobre os artigos(struct

articTable) e uma árvore binária (de procura)nea) que quarda a informação relativa aos contribuidores (struct contribTree). A escolha desta estrutura baseia-se no facto de que, face às queries apresentadas, a relação entre artigos e Esta divisão deve-se a termos entendido que podiamos responder às interrogações 4 e 5 independentemente das restantes interrogações, bem como responder às interrogações 3,6,7,8,9,10 com uma só estrutura. A estrutura é ainda constituída por dois long's que guardam o número de artigos únicos(artUn) e o número de artigos totais(artTot) que são calculados à medida que os snapshots vão sendo processados, permitindo assim que as respotas às duas primeiras queires sejam efetuadas em tempo (quase instantaneo) constante e tornando assim negligenciavel o tempo de cpu adicional requerido para o seu cálculo.

Estrutura dos artigos(struct articTable)

```
typedef struct articTable{
    long nArt;
    long size;
    articleInfoP *table;
}*articTableP;
```

Como já foi referido a estrutura utilizada para a informação relativa aos artigos é uma tabela de hash. Como método de tratamento de colisões escolhemos usar closed addressing visto que, apesar da estrutura adicional que requere (lista ligada), trata-se de uma forma mais simples de tratamento de colisões do que open addressing que introduziria complexidade desnecessária. Nesta estrutura recorremos a dois long's para armazenar o número de artigos(entradas) da hash table (nArt) bem como o tamanho da mesma(size). Estes dois valores sao usados no calculo do factor de carga da tabela de modo a sabermos quando necessita de ser redimensionada, permitindo assim a manutenção de um tempo amortizado (constante) nas operações de consulta, inserçao e remoçao.

Artigo(struct articleInfo)

```
typedef struct articleInfo {
    long id;
    xmlChar *title;
    struct revDict *revs;
    long nRev;
    long len;
    long words;
    struct articleInfo *next;
}*articleInfoP;
```

Cada estrutura artigo quarda o id do artigo (long id) de modo a ser possivel responder às (interrogações 6,8 e 10), bem como o seu título mais recente(xmlChar *title) (interrogações 7 e 8). Tem também um dicionário com as revisões do artigo(struct revDict *), a chave corresponde ao id da revisão

e o valor corresponde à altura em que esta foi efetuada(timestamp). Recorremos ainda a 3 long's (nRev,len,word) para armazenar o número de revisões no dicionário, o numero de caracteres da maior revisão efetuada e o número de palavras também da mais recente revisão.

Estrutura dos contribuidores(struct contribTree)

```
typedef struct contribTree{
    long id;
    xmlChar *nome;
    int nRev;
    struct contribTree *left;
    struct contribTree *right;
}*contribTreeP;
```

Para os contribuidores recorremos a uma árvore binária de procura ordenadaordenada pelo id dos mesmos. Esta escolhe foi feita tendo em mente um tempo de resposta reduzido às operações de consulta de contribuidores(interrogação 5) em detrimento da consulta dos 10 contribuidores mais ativos (interrogaçao 10) que, num cenário real, seria executada menos frequentemente a primera já que, num cenário real, a resposta não varia para o mesmo grupo de snapshots. Cada nó da árvore armazena o id do contribuidor, o nome do contribuidor (xmlChar *nome) e o número de revisões realizadas pelo mesmo(int nRev).

0.2.2 Pontos a destacar

De seguida destacamos algumas das decisões tomadas que consideramos terem em grande parte determinado o desempenho do trabalho. Uma destas opções foi a contagem de palavras e do tamanho do texto em simultaneo percorrer o texto apenas uma vez. Outra das opções a destacar foi o uso de funções não recursivas para estruturas recursivas, sendo exemplo disto a função

```
long* runTree(contribTreeP tree)
```

que calcula os 10 contribuidores mais ativos. O facto desta função não serrecursiva apresenta várias vantagens das quais destacamos a ausência do *overhead* envolvido na chamada de uma função bem como a diminuição da "*memory footprint*" do programa visto que não é preciso armazenar endereços de retorno nem de controlo de stack para as funções chamadas recursivamente, em contrapartida recorremos a uma stack sob a forma de um array dinamico. Por fim, na função

```
char** titles_with_prefix(char* prefix, TAD_istruct qs )
```

o uso do algoritmo *quickSort* para o ordenação por ordem alfabética do resultdo por oposição ao *mergeSort* deveu-se ao facto de a última ocupar mais memória ao recorrer a arrays auxiliares.

Part III

Conclusão

Apesar dos resultados obtidos terem sido satisfatórios, uma medida que poderia ser tomada para reduzir a *memory footprint* do programa seria o uso de um array com pesquisa binária ao invés de uma árvore binária, necessitando no entanto de realizar uma ordenação após a inserção dos valores. No entanto concluimos que o trabalho consegue, com sucesso e em tempo útil, responder às *queries* dadas, cumprindo assim o objetivo.