Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

Laboratórios de informática III

Relatório do Projeto em C

Grupo 59
Armando Santos A77628
Bruno Magalhães A75377
Luís Gomes A78701

April 29, 2017

Contents

1 Introdução

O objetivo deste projeto é criar um sistema que permita analisar artigos presentes em backups da *Wikipedia*, realizados em diversos momentos temporais e extrair informação útil bem como dados estatísticos dos mesmos. Para tal, utilizámos diferentes estruturas de dados que nos permitiram dar resposta em tempo útil às queries propostas.

Com o presente relatório pretende-se explicar de forma sucinta o funcionamento do sistema por nós criado bem como esclarecer os motivos das escolhas tomadas durante a sua realização.

2 Desenho

Para este projeto respondemos a vários tipos de queries. Tendo em conta a natureza das queries foi necessária uma abordagem que visa conjugar da melhor maneira a utilização de memória central e o tempo de resposta do sistema.

2.1 Abordagem

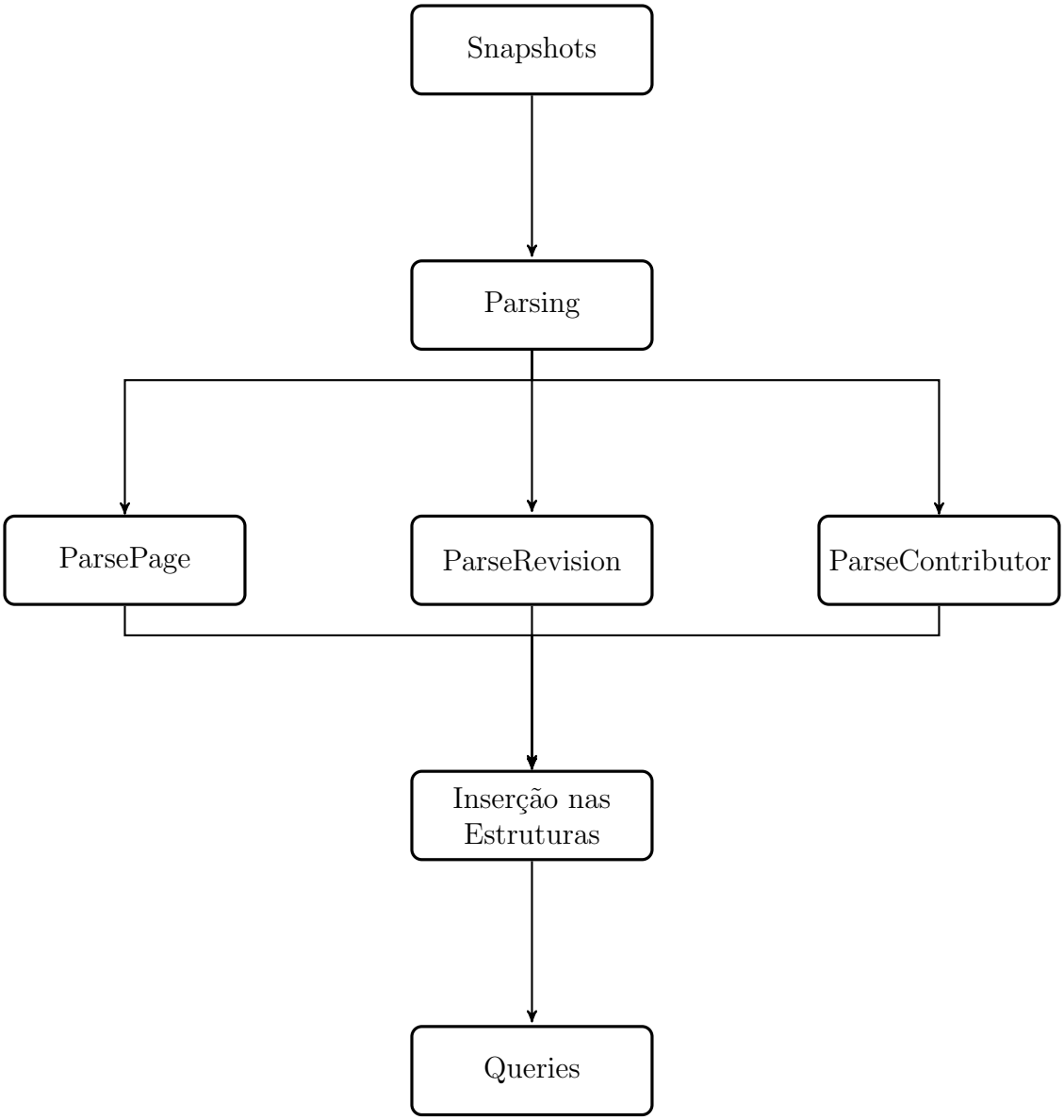
Ao longo deste projeto deparamo-nos com várias possíveis abordagens no que diz respeito à complexidade de tempo e espaço do sistema.

O objetivo do sistema é efetuar o *load* de uma lista de *backups* e o utilizador poder retirar toda a informação que desejar dessa lista, chamando as queries quantas vezes pretender tendo como garantia uma rápida resposta às mesmas.

Optamos por priveligiar o tempo de resposta das queries sacrificando, assim, o tempo de *load* tentando gerir da melhor maneira os recursos disponiveis. Posto isto, decidimos carregar toda a informação necessária de uma só vez.

Para cada *backup* da *wikipédia* é feito o *parsing* de cada artigo individualmente com o auxílio da biblioteca *libxml2*. Os vários tipos de informação são organizados e carregados nas respetivas estruturas que por fim poderão ser acedidos de forma eficiente aquando da chamada das várias queries.

O nosso programa segue, em suma, o diagrama apresentado.


Snapshots
Parsing
ParsePage
ParseRevision
ParseContributor
Inserção nas
Estruturas
Queries

2.2 Estruturas

Armazenamos os dados de forma a conseguir aceder à informação pretendida de modo eficiente. Com isto para responder a queries de procura não ordenada utlizamos HashTables por nos permitirem pesquisas em tempo constante.Para queries que impliquem ordenação utilizamos AVL's, que proporcionam acesso a dados ordenados em tempo logaritmico. Utilizamos ainda como estrutura auxiliar uma Lista Ligada que permite manter um pequeno registo de dados ordenados. Separamos a informação relativa aos artigos e aos contribuidores ficando com a seguinte estrutura:

```
struct TCD_istruct{
        long all_articles, unique_articles, all_revisions;
        hashTArt ht_art;
        hashTContrib ht_contrib;
        LinkedList top10Contribs;
        LinkedList top20LongestArticles;
        avlArtWords avlAW;
};
```

Listing 1: Estrutura Principal

A começar pelas variáveis *long* instanciadas no inicio da estrutura, o seu propósito é armazenar o contador do número total de artigos, de artigos únicos e o número total de revisões, de forma a responder às 3 primeiras queries. Estas variáveis são atualizadas ao longo do processo de parsing dos snapshots e o fluxo condicionado é:

```
  add_code = hashTArt_Add (...);
  qs->all_articles++;
  if (add_code) qs-> all_revisions++;
  if (add_code == 2) qs-> unique_articles++;
```

Listing 2: Atualização dos contadores

A variável *add_code* visível no código acima é o valor de retorno da função de inserção na HashTable de artigos que será discutida mais à frente e que poderá ter os seguintes valores:

- 0 - Nada de novo foi adicionado, ou seja, já existia aquele artigo e revisão;
- 1 - Foi adicionada uma revisão a um artigo existente
- 2 - Foi adicionado um novo artigo e consequentemente uma nova revisão

De uma forma simples, sempre que adicionamos um elemento à HashTable incrementamos o *all_articles*. Caso a revisão não exista, o valor devolvido é não nulo (1 ou 2) e é incrementado também o *all_revisions*. Sempre que o artigo não existe na HashTable será devolvido o valor 2 e incrementado ainda o *unique_articles*.

Vistas as varíaveis, passamos agora para as subestruturas que desempenham o papel principal do sistema.
Começando pelas HashTables, em particular a dos artigos, guardamos apenas a informação que nos será útil para as pesquisas que serão feitas posteriormente.

Os elementos são mapeados de acordo com o *id* do artigo e a função de hash apenas devolve o resto da divisão do *id* pelo tamanho da HashTable.

```
typedef struct hashtable{
  char * title;
  long title_ID;
  int n_bytes;
  int n_words;
  LinkedList revisions;
  struct hashtable * next;
}*hashTArt[SIZE], *artNodo;
```

Listing 3: HashTable de Artigos

O tipo *LinkedList* é uma Lista Ligada com a finalidade de guardar as revisões feitas em cada artigo, sendo a revisão mais recente adicionada à cabeça.

```
typedef struct llig{
  void *node;
  struct llig *next;
} *LinkedList;
```

Listing 4: Lista Ligada de Revisões

O tipo *Revision* serve como nodo para a LinkedList.

```
typedef struct revision{
  long revision_id;
  char* revision_timestamp;
} *Revision;
```

Listing 5: Nodo de Revisões

A HashTable respetiva aos contribuidores permite-nos mapea-los segundo o seu *id* e assim guardar informações úteis.

```
typedef struct hashtablecontrib{
  char* contributor_name;
  long contributor_id;
  int contributions_number;
  struct hashtablecontrib *next;
} *Contrib, *hashTContrib[SIZE];
```

Listing 6: HashTable de Contribuidores

Ao longo do *parsing*, sempre que é encontrada uma revisão nova de um certo contribuidor este é mapeado na HashTable e caso seja encontrada uma correspondência a sua variável *contributions_number* é incrementada, caso contrário é adicionada uma nova entrada na tabela.

Após as duas HashTables preenchidas com toda a informação de cada *backup* são criadas as duas Listas Ligadas auxiliares assim como a AVL de artigos que os organiza por ordem de número de palavras.

```

#pragma omp parallel sections
  {
  #pragma omp section
      getTop10NodesC(qs->ht_contrib, &(qs->top10Contribs));
            /* Top 10 Contribuidores */
  #pragma omp section
      getTop20NodesA(qs->ht_art, &(qs->top20LongestArticles));
            /* Top 20 Maiores Artigos */
  #pragma omp section
      qs->avlAW = avlArtWords_InsertALL(qs->ht_art, qs->avlAW);
            /* AVL de Artigos */
  }
```

Listing 7: Carregamento paralelo da informação para as estruturas auxiliares

Dada a natureza das queries *top_10_contributors* e *top_20_largest_articles* consideramos mais eficiente computar estes dois *tops* no fim do parsing visto que são fixos ao contrário da query *top_N_articles_with_more_words* cujo input é variável e, por isso, recorremos a uma AVL para sermos capazes de responder a esta interrogação.

3 Desempenho

Para conseguirmos apresentar um sistema não só funcional mas também eficaz na forma como apresenta os resultados tivemos em grande conta o desempenho por ele apresentado.
Como dito anteriormente priorizámos o tempo em função da memória utilizando assim estruturas que nos permitiram obter uma rápida resposta às queries.

3.1 Tempo

O tempo de resposta foi um dos aspectos que mais pesou na conceção deste projeto e na forma como foi conduzido. Para melhor percebermos as razões da nossa decisão são apresentados os tempos de execução de cada uma das queries solicitadas.[1]

```
init() -> 0.071000 ms
load() -> 6563.910000 ms
all_articles() -> 0.001000 ms
unique_articles() -> 0.000000 ms
all_revisions() -> 0.000000 ms
top_10_contributors() -> 0.003000 ms
contributor_name(28903366) -> 0.000000 ms
contributor_name(194203) -> 0.001000 ms
contributor_name(1000) -> 0.000000 ms
top_20_largest_articles() -> 0.003000 ms
article_title(15910) -> 0.001000 ms
top_N_articles_with_more_words(30) -> 0.002000 ms
article_title(25507) -> 0.001000 ms
article_title(1111) -> 0.001000 ms
titles_with_prefix(Anax) -> 0.696000 ms
article_timestamp(12,763082287) -> 0.000000 ms
article_timestamp(12,755779730) -> 0.000000 ms
article_timestamp(12,4479730) -> 0.000000 ms
clean() -> 6.051000 ms
```

[1]Resultados de 29/04/2017, 21h. Fonte : `http://li3.lsd.di.uminho.pt/results/grupo59/results_2017-04-29-21-00/`

Como podemos ver, o facto de utilizarmos várias estruturas, para além de uma maior complexidade de espaço, existe também um maior custo temporal nas queries *load* e *clean* uma vez que é dedicado mais tempo à organização e tratamento dos dados. Por outro lado e indo ao encontro do nosso objetivo, verificamos que todas as queries apresentam um tempo de resposta na ordem dos nanossegundos ou inferior, excetuando a função *titles_with_prefix* que se aproxima dos milissegundos. Tendo isto em consideração, podemos aferir que o desempenho do nosso programa proporciona ao utilizador uma melhor experiência do que se preferíssemos optar por uma abordagem que priveligiasse estas funções.

3.1.1 Paralelização

Para conseguir minimizar os tempos, principalmente no *parsing* que eram os mais custosos, e obter assim os resultados apresentados utilizámos paralelização tendo em conta as arquiteturas multicore dos processadores atuais. Para proceder à paralelização do código utilizámos o OpenMP.
Nos próximos excertos de código veremos a maneira como foi posta em prática a paralelização.

```
1  TAD_istruct load(TAD_istruct qs, int nsnaps, char * snaps_paths
       []){

3    xmlDocPtr doc;
     xmlNodePtr cur;
5    int i;
     #pragma omp parallel for ordered private(doc, cur)
7    for(i=0; i<nsnaps; i++){
       /* ....... */
9      /* Parsing */
           /* ....... */
11   }
```

A primeira paralelização efetuada foi ao nível dos snapshots em que pretendemos que cada um deles corra em simultâneo utilizando assim a paralelização do ciclo for do OpenMP, no entanto garantindo que correm por ordem (*ordered*). Temos que fazer privadas a cada thread de execução as variáveis doc e cur para que não haja conflitos no acesso às mesmas.

```
#pragma omp ordered
    {
      while(cur){
        if((!xmlStrcmp(cur->name, BAD_CAST "page"))){
          parsePage(qs,cur);
        }
        cur=cur->next;
      }
      xmlFreeDoc(doc);

  }
```

Pela mesma razão ao invocar as sucessivas funções *parsePage* utilizamos a mesma diretiva para que a informação de cada página seja processada simultaneamente mas mantendo a ordem de entrada.

```
  #pragma omp parallel sections
  {
  #pragma omp section
      getTop10NodesC(qs->ht_contrib, &(qs->top10Contribs));
  #pragma omp section
      getTop20NodesA(qs->ht_art, &(qs->top20LongestArticles));
  #pragma omp section
      qs->avlAW = avlArtWords_InsertALL(qs->ht_art, qs->avlAW);
  }
```

Neste caso foi utilizada a diretiva *sections* que nos permite criar secções em que cada uma será executada em paralelo.

4 Conclusão

A implementação deste sistema cumpre com os objetivos pretendidos com um desempenho dentro do previsto, no entanto talvez seja possível fazer uma melhor gestão dos recursos.

No que toca ao tempo de resposta das queries, à exceção da query número 9 (*titles_with_prefix*) que se desviou dos restantes resultados. Uma possível melhoria consiste na implementação de uma *Radix Tree* ou *Prefix Tree* que, como o nome indica, permite a ordenação dos artigos por prefixo que não foi efetuada devido ao peso que esta nova estrutura iria ter em termos de espaço utilizado e à escassez de tempo.

O código é modular, encontra-se bem dividido e estruturado. Respeita a abstração de dados e a *API* pretendida.