Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

Laboratórios de informática III

Relatório do Projeto em Java

Grupo 59
Armando Santos A77628
Bruno Magalhães A75377
Luís Gomes A78701

11 de Junho de 2017

Conteúdo

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, pretendemos 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

Neste projeto respondemos a vários tipos de queries. Tendo em conta a natureza das mesmas, 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

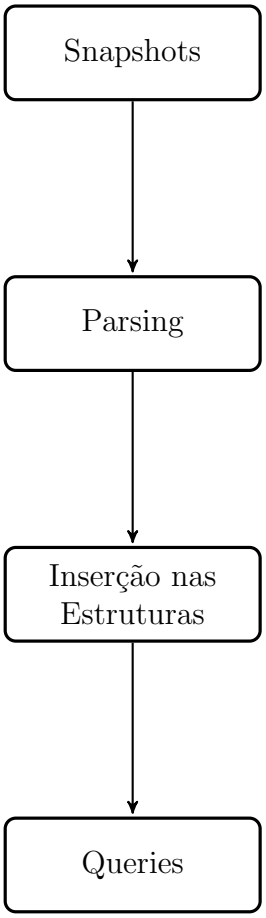
Deparamo-nos com várias abordagens possíveis no que diz respeito à complexidade de tempo e espaço do sistema.

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 disponíveis. Posto isto, decidimos carregar toda a informação necessária de uma só vez.

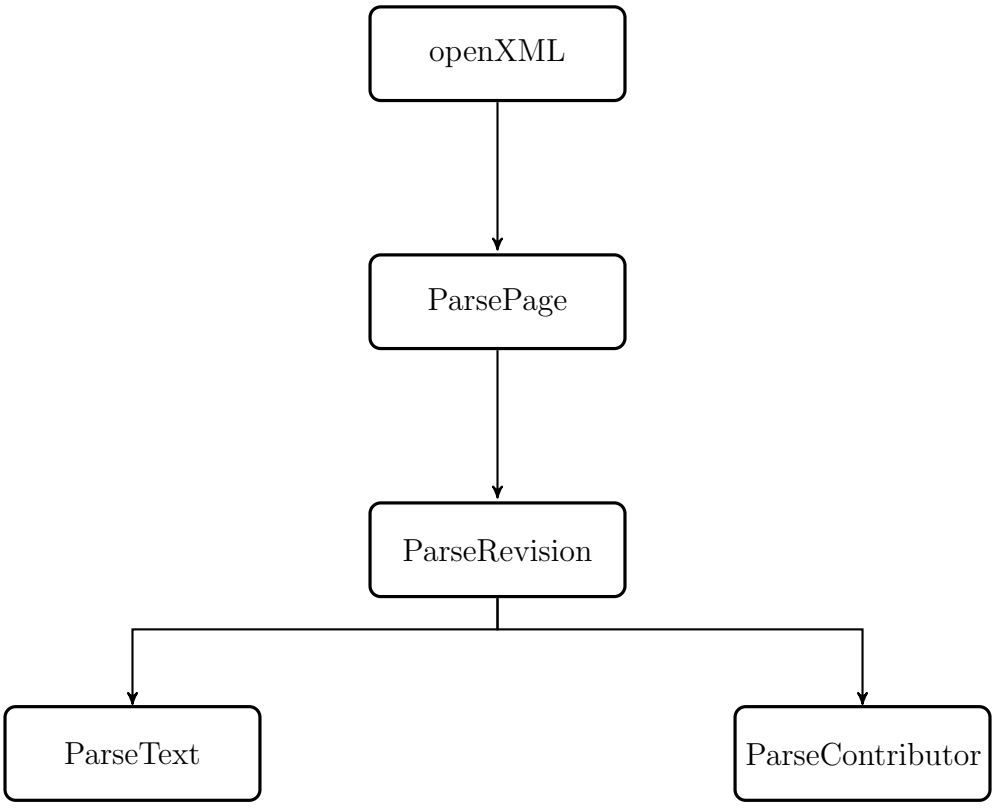
Para cada *backup* da *wikipédia* é feito o *parsing* de cada artigo individualmente, com o auxílio da biblioteca *StAX*. Os vários tipos de informação são organizados e carregados nas respetivas estruturas que, por fim, poderão ser acedidos de forma eficiente aquando da chamada das várias queries.

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


Snapshots
Parsing
Inserção nas
Estruturas
Queries

2.2 Parser

Tal como mencionado em cima, utilizamos a biblioteca *StAX* para realizar o *parsing* do XML. Esta usa o modelo de *pull Streaming* invés do modelo DOM. Este modelo permite um menor uso da memória, menos requisitos de processamento e, por vezes, maior performance. No entanto, apenas temos acesso ao elemento onde nos encontramos que, após iterarmos para o próximo, é descartado e recolhido pelo *garbage collector*.
O processo de *parsing* está ordenado da forma apresentada no seguinte diagrama.



2.3 Estruturas

Para guardar toda a informação relativa aos artigos e aos contribuidores utilizamos a *Collection HashMap* por ser possível fazer uma pesquisa em tempo constante. Para queries que impliquem ordenação dos elementos utilizamos

a *Collection TreeSet*, que proporciona acesso a dados ordenados em tempo logarítmico. Usamos também um *ArrayList* para guardar em lista todas as revisões relativas a cada artigo. Separamos a informação relativa aos artigos e aos contribuidores em classes diferentes. Criamos também uma classe que guarda a informação relativa a cada revisão.

```
public class Estruturas {

  private int unique_articles;
  private int all_articles;
  private int all_revisions;

  private HashMap<Long, Artigo> mapArtigos;
  private HashMap<Long, Contribuidor> mapContribuidores;
  private TreeSet<Contribuidor> topContribuidores;
  private TreeSet<Artigo> topArtBytes;
  private TreeSet<Artigo> topArtWords;
    ...
```

Listing 1: Classe Estruturas

A começar pelas variáveis *int* 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* sempre que o método que trata da inserção de informação nas estruturas é invocado.

```
  public void addInfo(Artigo article, Contribuidor contributor){
    long article_id = article.getTitle_ID();
    boolean sameRevision = false;

    Artigo old_article = mapArtigos.get(article_id);
    if(old_article == null){
      mapArtigos.put(article_id, article);
      this.unique_articles++;
      this.all_revisions++;
    } else{
      // se a revisao mais recente ja guardada for diferente da
   que queremos inserir)
            ...
       mapArtigos.replace(article_id, article);
       this.all_revisions++;
```

```
      } else
16      sameRevision = true;
    }
18  this.all_articles++;
      ...
```

Listing 2: Método addInfo()

De uma forma simples, sempre que invocamos o método incrementamos o *all_articles*. Caso a revisão a inserir seja mais recente que a atual o *all_revisions* é também incrementado. Quando ainda não existe um *mapping* para o *title_id* do artigo a inserir incrementamos o *unique_articles*.

Nos *HashMap's*, guardamos apenas as informações que nos serão úteis para as pesquisas que serão feitas posteriormente. Os elementos são mapeados de acordo com o *id* de artigo ou de contribuidor.

Ao inserir no *mapArtigos*, verificamos primeiro se já existe um *mapping* para o *id* do artigo. Se não existir, adicionamos o artigo. Se existir e se as revisões mais recentes não coincidirem, atualizamos o número de bytes que ocupa o artigo para o maior valor já registado, o número de palavras que o artigo contém e adicionamos, ao artigo já existente, a revisão mais recente.

Na inserção dos contribuidores no *mapContribuidores*, caso já exista um *mapping* para o *id* do contribuidor, a sua variável *contributions_number* é incrementada, caso contrário é adicionada uma nova entrada na tabela.

```
1 ...

3 if(!sameRevision)
      if(!contributor.isIP()){
5       long contributor_id = contributor.getContributor_id();
        Contribuidor old_contributor = mapContribuidores.get(
    contributor_id);
7       if(old_contributor == null)
          mapContribuidores.put(contributor_id, contributor);
9       else{
            old_contributor.addContributions_number();
11      }
      }
```

Listing 3: Método addInfo()

Após serem preenchidos os *HashMap's* com toda a informação de cada *backup*, são preenchidas também as *Collections* que guardam a informação dos artigos mas de forma ordenada.

Consideramos mais eficiente preencher estes *tops* no fim do *parse* a partir dos *HashMap's*, visto que mais nada é adicionado às tabelas nesse momento.

Esta implementação garante o encapsulamento dos dados.

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.

3.1 Tempo

O tempo de resposta foi um dos aspetos 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 -> 2 ms
LOAD -> 15916 ms
ALL_ARTICLES -> 0 ms
UNIQUE_ARTICLES -> 0 ms
ALL_REVISIONS -> 0 ms
TOP_10_CONTRIBUTORS -> 7 ms
CONTRIBUTOR_NAME -> 0 ms
TOP_20_LARGEST_ARTICLES -> 2 ms
ARTICLE_TITLE -> 0 ms
TOP_N_ARTICLES_WITH_MORE_WORDS -> 0 ms
TITLES_WITH_PREFIX -> 9 ms
ARTICLE_TIMESTAMP -> 0 ms
CLEAN -> 13 ms
```

[1]Resultados de 09/06/2017, 12:00h. Fonte : `http://li3.lsd.di.uminho.pt/results/grupo59/results_2017-06-09-12-00/times.txt/`

Como podemos ver, o facto de utilizarmos várias estruturas, para além de uma maior complexidade de espaço, existe também um maior custo temporal no método *load* uma vez que é dedicado mais tempo à organização e tratamento dos dados. Por outro lado, caminhando ao encontro do nosso objetivo, verificamos que todas as queries apresentam um tempo de resposta na ordem 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 privilegiasse estas funções.

3.2 Streams

Com a nova versão do Java foram introduzidas as *streams* que exploram o lado mais funcional da linguagem e permitem processar dados declarativamente. Deste modo, conseguimos de forma fácil e intuitiva percorrer diversas estruturas e aplicar um conjunto de métodos, por composição, a cada objeto individualmente de forma a obter o resultado pretendido.

Neste projeto é feito o uso desta API em dois locais distintos. Na inserção dos dados nas estruturas auxiliares e na realização das queries respetivas aos *tops*.

```
public ArrayList<Long> top_20_largest_articles() {
        return this.dataBase.getTopArtBytes()
        .stream()
        .map(Artigo::getTitle_ID)
        .limit(20)
        .collect(Collectors.toCollection(ArrayList::new));
    }

public void addToTopContribs(){
    this.topContribuidores = mapContribuidores.values()
        .parallelStream()
        .collect(Collectors.toCollection(TreeSet::new));
  }
```

Listing 4: Top 20 Largest Articles e Inserção no TreeSet de Contribuidores

4 Conclusão

A implementação do sistema é capaz de responder a todas as questões de forma correta e com um tempo de resposta baixo. Porem, sentimos que o *load* das estruturas poderia ser mais rápido.

Encontramos também uma certa dificuldade em compreender como utilizar o diferente modelo de *parsing*, onde são necessárias mais iterações do que o normal e o conteúdo referente a uma *tag* encontra-se num elemento diferente do da própria *tag*. Contudo, após alguma pesquisa sobre o assunto, chegamos à conclusão que a sua implementação não foi de tanta dificuldade como aparentava. Quanto ao restante, criar, preencher e aceder às estruturas, foi tarefa fácil, graças à vasta biblioteca da linguagem Java.

O código respeita o encapsulamento de dados e encontra-se, na sua generalidade, bem dividido e estruturado.