Universidade do Minho

MIEI - Grupo 49

Laboratórios de Informática 3

Projeto Java LI3

(a) Sérgio Alves A78296

(b) Hugo Brandão A78582

(c) Tiago Alves A78218

Conteúdo

1 Introdução

Este relatório explica como foi feita a implementação do mesmo projeto realizado na primeira parte desta disciplina, desta feita na linguagem de programação Java.

O projeto consistia numa implementação que permita analisar os artigos presentes em backups da Wikipedia, e retirar desses artigos todos os dados necessários para responder a uma lista de queries dada pelos professores. Contudo, tornou-se mais simples de realizar já com a parte C elaborada.

Para isto, tínhamos de criar estruturas de dados para armazenar toda a informação, e depois de carregada, estaríamos então em condições para executar as queries.

Dito isto, neste relatório são explicadas as nossas decisões tomadas para a elaboração deste projeto de Laboratórios de Informática,

2 Desenho da Solução

Para esta segunda parte do trabalho, e para responder corretamente às queries pretendidas, optámos por uma implementação simples mas eficiente. Dividimos o trabalho por módulos, ou seja, separamos o código por pequenos pedaços reutilizáveis de tal forma que seja fácil tanto reaproveitar estes pedaços de código, como fazer a manutenção sem que tenha impacto direto nos outros. Deste modo, a programação torna-se mais inteligente e menos suscetível a erros.

Na nossa implementação temos 5 classes fundamentais, sendo elas a *Parser*, que realiza o parse de todos os ficheiros, fornecidos pela equipa docente, para a nossa estrutura de dados, a *Article*, que guarda toda a informação necessária de um artigo, a *Revision*, com os dados de cada revisão, a *Contributor*, relativamente a cada contribuidor, e a *Structure*. A classe *Structure* define as estruturas onde vão ser guardadas todas as informações que queremos sobre os artigos. Nesta, utilizámos dois *TreeMap*, sendo um para os artigos e outro para os contribuidores. Optámos por esta decisão, devido às condições que um *TreeMap* nos oferece. Temos uma "chave", que é o id de cada artigo, associada ao respetivo artigo. Quando for necessário procurar por um deles, por exemplo, na query *contributor_name*, basta aceder à estrutura e fazer o "get(contributor_id)", em que contributor_id corresponde ao id do contribuidor pretendido, e temos imediatamente a resposta desejada, com o retornar do nome desse contribuidor. O mesmo se aplica à escolha da estrututra para os artigos. Além disto, utilizámos um inteiro que nos dá o total de artigos nos *snapshots*.

Em relação ao *parser* escolhemos *stax*, pois é de simples implementaão, com código legível, e com alguma rapidez na sua execução. Quanto à resolução das queries, foram feitas praticamente todas elas com recurso a streams, o que torna a solução rápida e eficaz.

3 Classes

Na seguinte secção vamos explicar as classes que decidimos criar para a resolução do projeto, incluindo diagramas e explicação das estruturas escolhidas para cada classe.

3.1 Article

Para cada artigo é necessário guardar toda a sua informação, mais concretamente, o seu id, o número de revisões que cada artigo tem, e uma estrutura para guardar todas as revisões. Para esta última, utilizámos um *TreeSet* proporcionando assim uma pesquisa rápida.

```
1 public class Article{
2     private int nrevisions;             /* Numero de revisoes do
          artigo */
3     private long id;                    /* O id do artigo */
4     private TreeSet<Revision> revList;  /* Lista de revisoes */
5     ...
6 }
```

3.2 Revision

Esta classe guarda tudo o que é necessário sabermos sobre uma revisão de um dado artigo. Aqui guardamos o seu id, o id do contribuidor dessa revisão, o título, data, número de palavras e número de carateres.

É uma classe simples, legível mas de grande importância para s execução correta deste projeto.

```
1 public class Revision{
2     private long id;              /* id of the revision */
3     private long idcontributor;   /* id of the contributor that
          wrote the */
4     private String title;         /* title of the revision */
5     private String date;          /* date of the creation of the
          revisions */
6     private int nwords;       /* number of words of the revision's
          article */
7     private int nchars;       /* number of characters of the revison
          's article */
8 }
```

3.3 Contributor

Aqui, estão todos os dados de um contribuidor, sendo eles o seu id, o número de contribuições que realizou em todos os artigos que nos foram fornecidos, e o seu nome.

Implementámos um método *compareNContributions* que é útil para o comparador de contribuidores, chamado na query *top_10_contributors*.

```
public class Contributor{
  private long id;                  /* id do contribuidor */
  private int ncontributions;    /* Numero de contribuicoes */
  private String name;              /* Nome do contribuidor */
}
```

3.4 Structure

Esta é a classe principal do projeto. Aqui é guardada a informação de todos os artigos, revisões e contribuidores. Tal como foi dito acima, na secção *Desenho da solução*, usámos um *TreeMap* para os artigos, e um outro para os contribuidores.

O método fundamental aqui é o *insert*, que faz exatamente o que o nome diz, isto é, insere toda a informação nas estruturas, artigos para o *articles* e contribuidores para o *contributors*. Desta forma, bastava apenas, nas queries, aceder a esta classe e implementar algoritmos para solucioná-las.

```
public class Structure{
    private int allArticles;                /* Numero total de
        artigos */
    private Map<Long,Article> articles;         /* Lista de artigos
         */
    private Map<Long,Contributor> contributors; /* Lista de
        contribuidores */
}
```

3.5 QueryEngineImpl

Nesta classe está toda a resolução das queries pretendidas. Optámos por, na maior parte, usar streams. Esta escolha deve-se ao facto de, desta forma, a execução tornar-se mais rápida e eficiente. *All_Revisions*, *Top_10_Contributors*, *Top_20_Largest_Articles*, *Top_N_Articles_With_More_Words*, e *Titles_With_Prefix* foram implementadas com streams. As restantes, não sendo feitas desta forma, foram igualmente elaboradas de modo que a sua execução fosse também eficiente.

```
public long all_revisions() {
  try{
    return this.struct.getArticles().values().stream()
                      .mapToLong(Article :: getNRevisions)
                      .sum();
  }catch(NoArticlesException e){System.out.println("Sem artigos");}
  return 0;
}
```

3.6 ArticleCharComparator

Esta class serve principalmente para ordenar um artigo segundo o número de carateres da revisão mais recente. É necessário para a query *top_twenty_Largest_Articles*, como se pode ver no excerto de código abaixo. Ordenamos a stream de artigos através deste comparador, obtendo uma stream ordenada como pretendíamos.

```
public ArrayList<Long> top_20_largest_articles() {
  ArrayList<Long> top_twenty_ids = new ArrayList<Long>(20);
  try{
   this.struct.getArticles().values()
                .stream()
                             .sorted(new ArticleCharComparator())
                             .limit(20)
                             .forEach(a -> top_twenty_ids.add((a.
                                 getId())));
  }catch(NoArticlesException e){System.out.println("Sem artigos");}
  return top_twenty_ids;
}
```

3.7 ArticleWordComparator

Do modo que funciona a classe anterior, o mesmo se aplica a esta, mas em vez de ordenar por número de carateres, ordena por número de palavras. Esta classe é aplicável na query *top_N_Articles_With_More_Words*.

```
public ArrayList<Long> top_N_articles_with_more_words(int n) {
  ArrayList<Long> top_n_ids = new ArrayList<Long>(n);
  try{
   this.struct.getArticles().values().stream()
                                     .sorted(new
                                         ArticleWordComparator())
                                     .limit(n)
                                     .forEach(a -> top_n_ids.add((a
                                         .getId())));

  }catch(NoArticlesException e){System.out.println("Sem artigos");}
  return top_n_ids;   }
```

```
public class ArticleWordComparator implements Comparator<Article> {
  public int compare(Article a1, Article a2){
    return  Integer.valueOf(a2.getBiggestRevision()
                  .getNWords()).compareTo(Integer.valueOf(a1
                  .getBiggestRevision().getNWords()));
    }
}
```

3.8 ContributorComparator

Para resolver a query dos 10 contribuidores com mais contribuições, criámos um comparador de contribuidores, sendo este comparador invocado para ordenar uma stream de estes mesmos. A ordenação é feita segundo o número de contribuições de cada contribuidor. Uma vez com a stream ordenada, bastou-nos fazer "limit(10)"e imprimir os resultados.

```
public class ContributorComparator implements Comparator<
    Contributor>{
  public int compare(Contributor c1, Contributor c2){
    return c1.compareNContributions(c2);
  }
}
```

3.9 RevisionComparator

Mais uma vez, esta classe é usada para ajudar na realização de uma das queries, desta feita a *Titles_With_Prefix*. O objetivo é ordenar a stream de revisões por título(ordem alfabética). Obviamente, são filtradas as revisões cujo título tem como prefixo a string passada como argumento.

```
public class RevisionComparator implements Comparator<Revision> {
    public int compare(Revision r1, Revision r2){
        return r1.getDate().compareTo(r2.getDate());
    }
}
```

```
public ArrayList<String> titles_with_prefix(String prefix) {
  ArrayList<String> res = new ArrayList<String>();
  try{
    this.struct.getArticles().values()
                .stream()
                            .map(Article::getRecentRevision)
                            .filter(r -> r.getTitle().startsWith(
                                prefix))
                            .map(Revision::getTitle)
                            .sorted(new RevisionTitleComparator())
                            .forEach(title -> res.add(title));
  }catch(NoArticlesException e){System.out.println("Sem␣artigos");}
  return res;
}
```

3.10 Exceptions

Como muitas das queries necessitam de ter acesso às estruturas, e visto que essas estruturas podem estar vazias (null), temos que prevenir erros futuros. Para isto, e como se mostra a seguir, criámos exceptions para os artigos, contribuidores e revisões. Quando uma query invocar um método que lançe um destas exceptions, é imprimida uma mensagem de erro.

3.10.1 NoArticlesException

```
1 public class NoArticlesException extends Exception implements
      Serializable{
2   public NoArticlesException(){ super(); }
3
4   public NoArticlesException(String message){super(message);}
5 }
```

3.10.2 NoContributorsException

```
1 public class NoContributorsException extends Exception implements
      Serializable
2 {
3   public NoContributorsException(){super();}
4
5   public NoContributorsException(String message){super(message);}
6 }
```

3.10.3 NoRevisionsException

```
1 public class NoRevisionsException extends Exception implements
      Serializable
2 {
3   public NoRevisionsException(){super();}
4
5   public NoRevisionsException(String message){super(message);}
6 }
```

4 Conclusão

Em suma, o trabalho foi realizado com sucesso apesar de alguns percalces. Tivemos de nos adaptar a um novo estilo de paradigma, ser eficazes e implementar as soluções mais ótimas para conseguir uma performance que satisfizesse os objetivos do grupo.

Achamos, no entanto, que este trabalho nos ajudou bastante a entender como a otimização e o processamento de grandes volumes de dados não é igual em todos os paradigmas da programação, tal como os seus cuidados.

Após a conclusão deste trabalho, pensamos que todos os elementos deste grupo saem com uma percepção mais clara do que é ser eficiente na hora de aceder a estruturas de dados e de processar grande quantidade de informação.

Ficamos, então, contentes com o nosso desempenho e com o resultado obtido, acreditando que todos os objetivos foram cumpridos.