Universidade do Minho

Laboratórios de Informática III

2017/2018

Relatório do trabalho

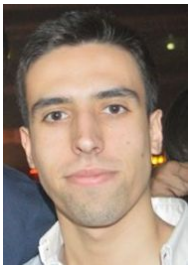
Parte Java

Curso: MIEI

2ºAno

2ºSemestre

Grupo 2



António Lopes

A74357

Bernardo Viseu

A74618

Fernando Pereira

A75496

Índice

1. Introdução

Esta é a segunda parte do trabalho que nos foi proposto pelos docentes da unidade curricular Laboratórios de Informática III e tem como principal objetivo o desenvolvimento de um sistema capaz de processar ficheiros XML que armazenam as várias informações utilizadas pelo Stack Overflow, desta vez usando a linguagem de programação Java. Uma vez que vão passar pelo nosso programa milhões de dados, torna-se um desafio tornar esta programação em larga escala eficiente, respondendo ainda às 11 interrogações propostas no enunciado.

2. Familiarização com o projeto

À semelhança do que fizemos na primeira parte do projeto em C (onde armazenamos todos os Post's num calendário, criando então 2 HashTable's em cada dia para lá ficarem as perguntas e respostas), neste projeto seguimos o conselho da equipa docente aquando da apresentação e optamos por apenas guardar em cada dia um ArrayList com os ID's dos post's criados nesse dia e à parte estão todas as perguntas e respostas em HashMap's separadas.

Deste modo poupamos bastante memória (já não são definidas milhares de HashTables) e continua a ser possível uma rápida procura quando é relevante as datas dos post's.

Os User's e as Tag's foram também guardados em HashMap's.

Dentro do dump que nos é fornecido para testar o programa (divididos os dados nas pastas android e ubuntu) encontram-se os ficheiros Badges.xml, Comments.xml, PostHistory.xml, PostLinks.xml, Posts.xml, Tags.xml, Users.xml, Votes.xml. Para a realização do programa apenas precisamos dos seguintes ficheiros, que contêm a seguinte informação:

- **Posts.xml**

  Ficheiro com os dados de todos os comentários. Apenas é necessário recolher certa informação das perguntas (Id, Score, OwnerUserId, Title, CommentCount, Tags, AnswerCount e CreationDate) e respostas (Id, Score, OwnerUserId, ParentId e CreationDate). Distinguidos através do Id. As perguntas têm PostTypeId=1 e as respostas PostTypeId=2.

- **Users.xml**

  Ficheiro com os dados de todos os Users. Distinguidos através do Id. Destes nós precisamos das informações Id, Reputation, AboutMe e DisplayName.

- **Tags.xml**

  Ficheiro com os dados das tags presentes em cada post, devido à query 11 é necessário armazenar alguma informação (Id e TagName) de cada tag na estrutura.

3. Tipo Concreto de Dados

O nosso **tipo concreto de dados** (TCD) contém as estruturas que foram definidas para armazenar os dados, que são quatro HashMap (Users, Questions, Answers e Tags), e um DataCalendar (Classe criada por nós) que contém uma ArrayList de Years, cada Year contém uma ArrayList com MMonths, e cada MMonth contém uma ArrayList de Days, e cada Day tem uma ArrayList de Longs (Ids de posts deste dia), formando assim um calendário para inserir os posts cronologicamente.

Construtor vazio TCDCom, utilizado para criar um novo objecto comunidade:

```
/**
 * Construtor vazio
 */
public TCDCom() {
    this.hashUsers = new HashMap<>();
    this.hashTags = new HashMap<>();
    this.hashQuestions = new HashMap<>();
    this.hashAnswers = new HashMap<>();
    this.calendar = new DataCalendar();
}
```

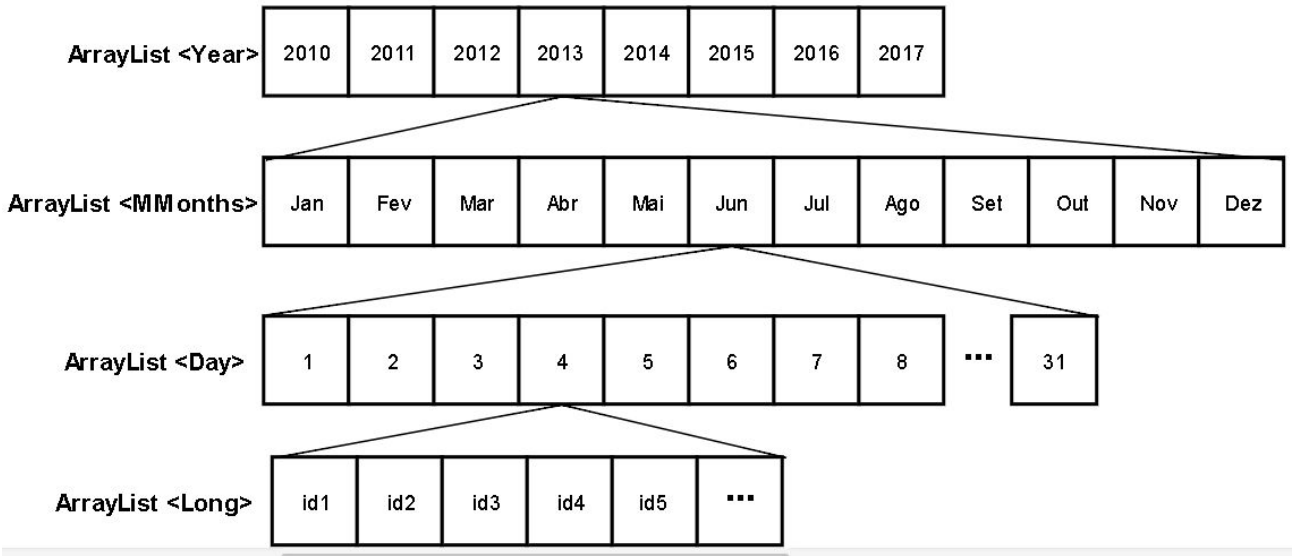
4. Estrutura de dados usada

- **Users:** Para os users foi usada uma HashMap, onde foi armazenado um User como value, sendo a key o Id do User.

O construtor parametrizado responsável pela criação de um objecto da classe **User**, contida em **User.java**, é a seguinte:

```
/**
 * Construtor parameterizado de um User
 * @param id    ID do User
 * @param name  Nome do User
 * @param aboutme   AboutMe (informaçao do User)
 * @param reputation    Reputation do User
 * @param numberOfPosts Numero de posts do User (isto é sempre inicializado a 0)
 */
public User(long id, String name, String aboutme, int reputation, int numberOfPosts){
    this.id = id;
    this.name = name;
    this.aboutme = aboutme;
    this.reputation = reputation;
    this.numberOfPosts = numberOfPosts;
}
```

- **Posts:**



Os ID's dos post's foram inseridos num ArrayList dentro do dia em que foram publicados. post. Tendo nós o calendário do ano (DataCalendar) com um ArrayList de Anos, ArrayList de Meses e ArrayList de dias, colocamos os ID's assim organizados de maneira a mais tarde nós facilitar na resolução das querys.

As perguntas e respostas estão em HashMap's à parte, sendo os ID's as key's.

```java
/**
 * Construtor parameterizado de um Question
 * @param id_q  ID da Question
 * @param score_q  Score da Question
 * @param user_id  ID do User que publicou essa Question
 * @param titulo Titulo da Question
 * @param comment_count Número de comentários dessa Question
 * @param tags  Tags da Question
 * @param numberAnswers Número de Answers da Question
 */
public Question(long id_q, int score_q, long user_id, String titulo, int comment_count, String tags, int numberAnswers) {
    this.id_q = id_q;
    this.score_q = score_q;
    this.user_id = user_id;
    this.titulo = titulo;
    this.comment_count = comment_count;
    this.tags = tags;
    this.numberAnswers = numberAnswers;
}
```

```java
/**
 * Construtor parameterizado de um Answer
 * @param id_a  ID da Answer
 * @param score_a  Score da Answer
 * @param user_id_a  ID do User que publicou essa Answer
 * @param comment_count_a Número de comentários dessa Answer
 * @param parent_id ID da Question a que essa Answer responde
 */
public Answer(long id_a, int score_a, long user_id_a, int comment_count_a, long parent_id) {
    this.id_a = id_a;
    this.score_a = score_a;
    this.user_id_a = user_id_a;
    this.comment_count_a = comment_count_a;
    this.parent_id = parent_id;
}
```

- **Tags –** A estruturada usada para o armazenamento das Tags foi uma HashMap, onde se armazenou o ID, o nome e count (número de vezes que a tag é usada), em que a key é o ID da Tag.

```java
/**
 * Construtor parametrizado de uma Tag
 * @param tagname Nome da tag
 * @param count Numero de vezes que a tag é usada
 * @param tag_id Id da tag
 */
public Tag(String tagname, int count, long tag_id) {
    this.tagname = tagname;
    this.count = count;
    this.tag_id = tag_id;

}
```

5. Modularização Funcional

O objetivo da modularização passa por tornar o código mais legível e aumentar o desempenho. Na realização do trabalho temos a divisão do trabalho em 3 packages, common, engine e li3.

Na classe common temos as classes utilitárias, ou seja, temos as várias estruturas do calendário (Year, Month, Day), o parser e as classes que definem os Users, Questions, Answers e Tags.

No package engine estão os vários Comparators e Exceptions que são utilizados nas queries, o ficheiro TCDcom que é onde estão definidas as queries e as funções load e clean.

No package li3 é onde se encontra a main, responsável por inicializar o programa, recebendo o path, e a interface TADCommunity que é implementada no TCDcom.

6. Abstração de dados

Os tipos abstratos de dados servem para criar e usar uma classe e definir um conjunto de operações para manipular dados dessa classe. As classes que foram encapsuladas estão na pasta Grupo2/proj-java/src/main/java/common, onde é possível usar as funções definidas sem referência a detalhes da implementação. Os nossos TAD's foram implementados, criando os Getter's e Setter's para todas as variáveis de cada classe, e também toString, clone e equals em cada classe, tornando-se mais tarde muito úteis para a construção do parser e realização das querys, pois permitem trabalhar com uma classe de maneira muito mais simples.

Para além disso, temos as seguintes funções que foram necessárias para a realização das queries:

Na classe Users.java, que incrementa o número de post's de um User:

public void incrementNumberOfPosts();

Na classe Tags.java, que incrementa o número de vezes que a Tag é usada:

public void incrementCount();

Na classe Questions.java, que retorna as tags de uma pergunta, numa ArrayList<String>

public ArrayList<String> getSeparateTags();

7. Load (Parse do Dump)

A estratégia adotada consistiu em iniciar o DataCalendar com 10 anos (utilizando o método **init()**), tendo em cada dia uma ArrayList<Long> dos Ids dos posts.

O nosso parser foi construído com o auxílio das bibliotecas StAX, a partir dos imports:

```
6   import java.io.FileInputStream;
7   import java.io.FileNotFoundException;
8   import javax.xml.namespace.QName;
9   import javax.xml.stream.XMLEventReader;
10  import javax.xml.stream.XMLInputFactory;
11  import javax.xml.stream.XMLStreamException;
12  import javax.xml.stream.events.Attribute;
13  import javax.xml.stream.events.EndElement;
14  import javax.xml.stream.events.StartElement;
15  import javax.xml.stream.events.XMLEvent;
16
```

Depois o método do DataCalendar ***addID()*** trata de inserir cada ID de um post, seja ele resposta ou pergunta, no seu dia (a função ***xmlToDate(String date)*** permitiu-nos transformar uma string numa LocalDate).

Esta função é chamada no método da classe Parser **parserQuestionsAnswers**, que é responsável por correr o ficheiro Posts.xml e inserir na estrutura os dados que pretendemos guardar, ou seja, os ID's no respectivo dia, e uma Answer ou Question na respectiva HashMap (dependendo do PostTypeId).

O mesmo mecanismo ocorreu para os Users e as Tags. O método ***parserUser*** é onde é fornecido o path para o ficheiro onde está o dump, e a HashMap dos Users, corre o ficheiro Users.xml e insere na estrutura os dados que pretendemos guardar para os Users. Nas Tags é na função ***parserTags*** que damos o caminho para o dump e a HashMap de Tags, e onde se corre o ficheiro Tags.xml e insere na estrutura as tags com os dados pretendidos.

Como apenas é dado o path para o dump, utiliza se concat para adicionar o ficheiro que se quer percorrer no fim da String do path, por exemplo ***path.concat("Posts.xml")*** no método parserQuestionsAnswers.

8. Estratégias

8.1 Queries

1. Info from Post

Nesta query utilizamos dois if's, para verificar se o Id dado é uma pergunta ou resposta, utilizando o método containsKey da HashMap. Caso o Id seja um key na HashMap das perguntas, vamos buscar a pergunta correspondente, e o User que publicou a pergunta (a partir do getUser_id()) e retornamos um Pair com o título da Question (usando getTitulo()) e o nome do User (usando getName()). Caso o Id seja um key na HashMap das respostas, vamos buscar a pergunta a qual o post está a responder (a partir de getParent_id()), e o User que publicou a pergunta (a partir do getUser_id()) e retornamos um Pair com o título da Question (usando getTitulo()) e o nome do User (usando getName()).

2. Top Most Active

Na realização desta query começamos por percorrer as duas HashMaps de Posts (Questions e Answers), e por cada post, vamos buscar o User que o publicou, e incrementar o número de posts (incrementNumberOfPosts()) nos dados do User, após isto, utilizamos a classe ComparatorNumberPosts (classe implementa Comparator<User>), e usamos uma stream dos values da HashMap de Users, para ordenar os Users pelo número de posts, retiramos os primeiros N Users, e retornamos os ids de esses Users como List<Long>.

3. Total Posts

Nesta query voltamos a percorrer o calendário mas apenas entre as datas fornecidas. Em cada dia percorremos a ArrayList<Long> de Ids e caso um Id fosse de um Question (verificamos isto a partir de containsKey(id)), incrementamos uma variável local questions (long inicializado a 0), caso seja Answer incrementamos a outra variável local (answers). No fim retornamos um Pair<Long, Long> com as variáveis questions e answers.

4. Questions with Tag

Novamente percorre-se o nosso calendário entre as datas dadas e verificamos se o Id é de uma Question. Retiramos a String das tags da Question, e utilizamos um contains() para verificar se contém a String tag dada, em caso afirmativo recolhemos o Id da Question e adicionamos numa ArrayList<Long>. Por fim utilizamos Collections.reverse() da ArrayList dos Ids, para termos o resultado em cronologia inversa, e retornamos essa lista.

5. Get User Info

Nesta query a estratégia adotada passou por correr o calendário a partir do fim (data do último post), acedendo assim aos últimos post's efetuados, e verificar se o Id do post foi publicado pelo User fornecido, caso se confirme isso, guardamos o Id do post numa ArrayList<Long>. Por fim retornamos os 10 primeiros Id's dessa lista.

6. Most Voted Answers

Para a contagem do número de votos utilizamos o atributo Score dos Post's.

Nesta query percorremos o calendário entre as datas dadas e recolhemos para uma ArrayList<Answer> os dados de cada resposta. Depois disso utilizamos uma stream aplicada a essa ArrayList, que ordena as Answers (a partir do comparator ComparatorVotosAnswer) por ordem decrescente do Score de cada post, retira os primeiros N resultados, e coloca os Ids dessas Answers numa List<Long>, que é retornada.

7. Most Answered Questions

Nesta query utilizamos uma abordagem semelhante a query anterior, em que usamos uma ArrayList<Question> auxiliar que no fim é aplicada uma stream para ordenar a lista (desta vez com um ComparatorNumberAnswers) pelo número de respostas, e dos primeiros N resultados retira se os Ids para uma List<Long> que é retornada.

8. Contains Word

Nesta query, como pedem para devolver os ID's ordenados por cronologia inversa, percorremos o calendário a partir do final. Em cada dia verificamos se um Id é de um post Question, e caso a String título da Question contenha a String dada, guardamos esse Id numa ArrayList<Long> auxiliar, depois disto retornamos os primeiros N elementos dessa lista (apenas no caso de a lista ter mais que N elementos, caso contrário retornamos a lista toda).

9. Both Participated

Nesta interrogação, como pedem para devolver as últimas N perguntas em que participaram dois utilizadores, começamos por percorrer o calendário a partir do final e percorrer os Ids dos posts (tanto das Questions como das Answers) à procura dos User's ID que coincidam com os do input. Guardamos então os Posts dos Users do input entre as datas em 4 listas: User1 -> questionsUser1 (ArrayList<Long> de Ids das Questions) e answersUser1 (ArrayList<Long> de Ids do ParentId da Answer); User2 -> questionsUser2 e answersUser2. A seguir percorremos a questionsUser1 e verificamos se existe na answersUser2 respostas às perguntas feitas pelo User1, em caso afirmativo guardamos o ID da pergunta numa lista result. Voltamos ao mesmo raciocínio mas neste caso percorrendo a questionsUser2 e verificar se existe alguma resposta às perguntas da answersUser1, caso exista, adiciona-se o ID da pergunta à mesma lista result. Finalmente, só falta verificar se existe nas answersUser1 e answersUser2 respostas à mesma pergunta e adicionar na lista result caso isso se verifique. Depois disto retornamos os primeiros N elementos dessa lista (apenas no caso de a lista ter mais que N elementos, caso contrário retornamos a lista toda).

10. Better Answer

Nesta query começamos por percorrer a HashMap de Answers à procura das respostas com o Parent ID (ID do post a que estão a responder) igual ao fornecido, guardando então numa lista todas as respostas em que essa condição se verifique. Depois utilizamos um ComparatorBestAnswer (Comparator<Answer> em que damos a HashMap de Users) e ordenamos essa lista por ordem decrescente das melhores respostas após serem feitos os cálculos indicados ((Score × 0.65) + (RepUser × 0.25) + (CommentCount × 0.1)) e retornarmos o ID do primeiro elemento da lista que é a melhor resposta à pergunta dada.

11. Most Used Best Rep

Nesta query foi necessário utilizar a HashMap das Tags pela primeira vez. Percorremos o calendário entre as datas dadas, e o que se fez foi guardar as Questions desse período de tempo numa ArrayList<Question> e também guardamos todos os users que postaram questões neste período de tempo, numa ArrayList<User>, depois disto ordenamos essa lista pela reputação dos Users (usando um ComparatorRepUsers), e guardamos apenas os primeiros N Users dessa lista. De seguida percorremos a ArrayList<Question> referida anteriormente, e caso uma Question tenha sido publicada por um dos N melhores Users, percorre se cada Tag desse post (utilizando o auxílio do método getSeparateTags(), que retorna uma ArrayList<String> com cada tag) e usa se o método (incrementCount()) para incrementar a count das tags usadas nessas questões. Por fim aplica se uma stream aos values da HashMap de Tags, que as ordena pela count (usando um ComparatorTagCount), e nos dá os primeiros N Ids em forma de List<Long>, que é retornada.

8.2 Melhoramento de desempenho

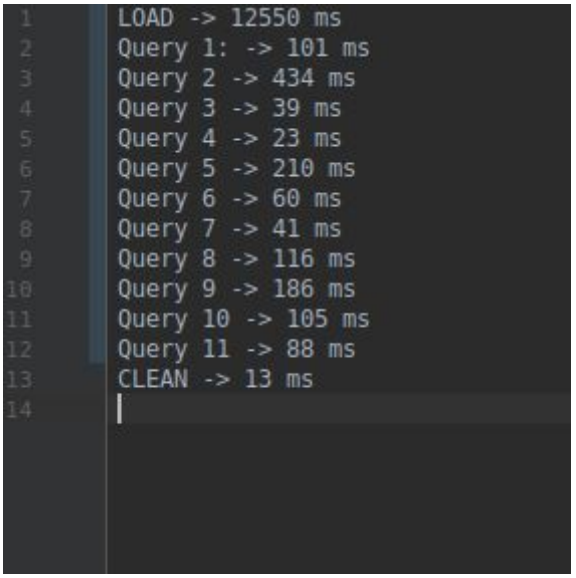
Este projecto foi abordado de maneira diferente do que a parte em C, o facto de já estarmos familiarizados com as estruturas deu nos logo um ponto de partida, tivemos em conta o que nos foi dito na apresentação da parte em C, e mudamos a nossa estrutura em conta de isso, em vez de termos duas HashTables em cada dia do calendário, fizemos com que cada dia tivesse apenas uma ArrayList de Ids de todos os posts feitos nesse dia, e guardamos separadamente, em duas HashMaps, as Questions e as Answers, isto provou ser muito útil no realizamento de queries, já que não foi necessário percorrer o calendário todo para percorrer as perguntas/respostas todas, o que fez o nosso programa muito mais optimizado.

A utilização de streams mostrou se muito útil neste projecto, para ordenar, delimitar e transformar o conteúdo de listas/coleções e foi uma maneira de diminuir o número de linhas de código do trabalho (em vez de utilizar demasiados for loops).

No parsing dos ficheiros utilizamos uma abordagem semelhante a do projecto em C, em que percorremos apenas uma vez o ficheiro Posts.xml, separando os posts a partir do PostTypeId, de maneira a não ser preciso abrir o ficheiro mais do que uma vez, que seria uma grande perda de tempo.

Começamos o projecto a utilizar DOM (Document Object Model) como método de parse dos ficheiros dump, mas depois de pesquisa achamos melhor utilizar StAX, porque foi nos dito que era um método mais rápido e que utiliza menos memória, por isso refizemos o parser todo para utilizar StAX, o que se mostrou ser um grande improvement, diminuindo o tempo de load por cerca de 5 segundos (de ~18s a ~12s).

Os nossos tempos finais foram os seguintes:



9. Conclusão

Ao longo da realização deste trabalho adquirimos muitos conhecimentos na linguagem de programação Java. Aprendemos muito sobre o funcionamento e utilização de classes como Maps, Lists, e os vários métodos delas em Java, ganhando outra perspetiva de como a estrutura escolhida para o armazenamento dos dados se torna um fator muito importante na performance do programa, principalmente tendo em conta que estamos a trabalhar com quantidades enormes de informação.

Aprendemos também a utilizar e aplicar parsing em Java. Este trabalho tornou-se ainda mais interessante devido ao facto de ser possível executá-lo de várias maneiras, utilizando diferentes tipos de estruturas. Decidimos utilizar uma estrutura parecida com a que utilizamos em C, tendo em conta algumas optimizações já referidas anteriormente neste relatório, cujas foram aconselhadas pelos professores durante a apresentação do trabalho em C.

Java mostrou se ser uma linguagem de programação mais “amigável” que C, em que a existência de muitas classes e API’s facilitaram o desenvolvimento deste projecto. O facto de existirem métodos mais convenientes de percorrer listas (como streams) também foi uma grande diferença do trabalho em C.

Para concluir ficamos satisfeitos com o nosso projecto, porque, apesar de sabermos que é possível abordar este trabalho de várias maneiras e que a nossa não será a melhor de todas, de qualquer modo aprendemos muito sobre programação em C e Java, e como se deve fazer um trabalho de grupo, com divisão de tarefas entre nós, e este tipo de projectos da nos experiencia para conseguirmos trabalhar bem em projectos de maior escala, num ambiente de trabalho profissional.