Relatório do projeto de LI3

João Pimentel A80874 Jaime Leite A80757
Bruno Veloso A78352

Junho 2018

Universidade do Minho
Departamento de Informática
LI3
Grupo 22

Conteúdo

1 Introdução

Este relatório aborda a resolução do projeto prático de *"Laboratórios de Informática 3"*, na linguagem *Java*. O mesmo consiste em desenvolver um programa para analisar dados presentes em ficheiros XML relativos a conteúdo presente no *StackOverflow*. De modo a realizar esta tarefa foi necessário criar seis classes: quatro sendo mais básicas, que caracterizam os dados presentes nos ficheiros (*answers, questions,tags* e *users*); uma para a realização do *parse/load*; e uma para executar as *queries* propostas ao grupo.

Ao longo do relatório serão abordadas as decisões tomadas na implementação do projeto, nomeadamente o porquê das classes e seus parâmetros, bem como a forma como foram implementados os métodos mais importantes na execução do trabalho.

2 Classes

Nesta secção são apresentados excertos de código comentado, tal como a explicação das soluções utilizadas e as divisões do projeto.

2.0.1 User

É composto pelos seu *ID*, *username*, contagem de *posts* (perguntas e respostas), a sua biografia e reputação.

```
private Integer id;//id do user
private String username; //nome do user
private int post_count; //numero de posts do user
private String short_bio; //biografia do user
private int reputation; //reputacao do user
```

2.0.2 Answer

Possui a data em formato de inteiro (exemplo: 01-12-2018 passa a 20181201), o seu *ID*, o do autor e da respetiva questão, o número de comentários, o *score* e a quantidade de favoritos.

```
private Integer date; //data da resposta
private Integer id_resp; //id da resposta
private Integer id_question; //id da questão
private Integer id_user; //id do user
private int comentarios; //numero de comenantarios
private int score; //score da resposta
private int favoritos; //numero de favoritos da resposta
```

2.0.3 Question

Mais uma vez, possui o seu *ID* e do respetivo autor, a data de criação , título, *tags*, os votos e a contagem de respostas.

```
private Integer id_user;//id do user
private Integer date; //data da questao
private Integer id; //id da questao
private String titulo; //titulo da questao
private String tags; // tags da questao
private int votos; // numero de votos da questao
private int respostas_count; //numero de respostas
```

2.0.4 Tag

Tem apenas três características: uma *String* com a *TAG* em questão (exemplo: “ssh”), o seu *ID* e um contador para usar posteriormente na análise dos dados.

```
public Integer id; //id da tag
public String tag; //nome da tag
public int conta; //numero de vezes que aparece (query 11)
```

2.0.5 TCD community

Possui as quatro estruturas de dados, no formato *Map*, sendo que cada um é composto por elementos do tipo <*Integer,CLASSE*>, sendo que a chave de procura no mapa é um inteiro, representante do *ID*.

```
private Map <Integer,Answer> answers; //guarda as respostas (id da resposta, resposta)
private Map <Integer,Question> questions; //guarda as questoes (id da questao, questao)
private Map <Integer,User> users; //guarda os users (id do user,user)
private Map <Integer,Tag> tags; //guarda as tags (id da tag, tag)
```

2.0.6 Parsing/Load

Classe que representa uma das partes fundamentais do projeto, o *parse* dos ficheiros e *load* do seu conteúdo essencial para a memória.

2.0.6.1 ParsePosts

Esta é o método mais complexo desta classe, já que faz a inserção de tanto questões, como respostas nos respetivos *Maps*.

É utilizada, por diversas ocasiões, o método *"eElement.getAttribute("...")"* para encontrar a característica pretendida, uma de cada vez.

No momento em que todos os parâmetros de uma classe já estão guardados, é criada uma nova instância de *answer* ou de *question,* consoante o caso em questão, para acrescentar ao respetivo *Map*.

É ainda incrementado o número de *posts* de um utilizador.

2.0.6.2 ParseUser

O método funciona da mesma forma que a anterior, só que gera um *user*, partindo do mesmo princípio de procura dos parâmetros e criação de uma instância completa.

2.0.6.4 ParseTags

A função *parseTags* corre o ficheiro *Tags.xml*, criando instâncias da classe. Funciona de igual modo a todos os outros *parses* anteriores.

3 Queries

Nesta secção serão mencionados os modos como foram executadas as *queries* fornecidas e qual o algoritmo desenvolvido para atingir ao resultado esperado.

3.0.1 Init

A *query init* apenas inicia as estruturas das classes com um *Map* vazio. Serão, depois, alteradas para o seu valor final depois de aplicadas as funções de *load*.

3.0.2 Load

Esta *query* possui a carga de trabalho mais elevada do programa, já que carrega todas os dados necessários à realização das outras *queries*, correndo os ficheiros de *Users*, *Posts*, e *Tags*. De modo a amortizar o tempo de *loading*, foi decidido que certos parâmetros que podiam ter sido acrescentados às estruturas, como por exemplo, os *ID's* de todos os *posts* de um *user*, implicandi um aumento no período de execução e este aspeto nem sempre é necessário à realização das interrogações.

3.0.3 Info From Post

Dado o identificador de um *post*, a função retorna o título do *post* e o nome de utilizador do autor. Se o *post* for uma resposta, a função deverá retornar informações da pergunta correspondente. Para isto foi necessário encontrar as informações em questão nos *Maps*, existentes na *TAD community*, usando funções de procura.

3.0.4 Top Most Active

Tendo em conta que o *output* da *query* em questão deveria ser uma *List<Long>* com *N ID's* dos *users* mais ativos, ou seja, com mais *posts* (perguntas e respostas), bastou "clonar" os *users* para uma lista com os *values* do *Map* e ordená-la pelo parâmetro *post_count*. Posteriormente, são retirados os *N* elementos com mais *posts* da referida lista para a estrutura final.

3.0.5 Total Posts

Como é pedido que seja devolvido *Pair<Long,Long>*, que representem o número de perguntas e respostas num dado intervalo de tempo, a resolução começou por converter as datas dadas para inteiros, de forma a serem feitas comparações mais eficientes. Definiram-se duas listas com os *values* dos *Maps* das respostas e das perguntas e, com uma função que percorre todos os elementos das mesmas, em que se verifica se a data do *post* está contida entre dois valores acima mencionados, adiciona-se uma unidade ao respetivo contador.

3.0.6 Questions with tag

Uma vez que as *tags* de uma determinada *QUESTION* estão guardadas na forma de *String*, sendo divididas entre si por *"<>"*, a solução evidente passou por usar o método *contains* já definido em *Java* em que se verifica se o argumento *"<tag>"* pertence a um conjunto de *tags* de uma determinada questão.

Usando o mesmo método na *querie* anterior para validar se uma questão pertence ao intervalo dado, só foi necessário criar uma *List<Question>* e ir adicionando as questões (ordenadas por data) que contêm a *tag* e que pertencem ao intervalo de tempo.

3.0.7 Get User Info

Para a *query* em questão foi adotado um método bastante simples, em que, primeiramente, foi encontrado o utilizador do input, usando um método de procura sobre o *Map* dos *users*, consoante o *ID* do mesmo. Em seguida, agruparam-se perguntas e respostas, do utilizador em questão, em duas listas, ordenadas por data. Finalmente juntam-se as duas coleções numa única, em que se vão colocando todos os *posts* por ordem crescente de data de publicação.

3.0.8 Most Voted Answers

De modo a obter as *N* respostas com mais votos, num certo intervalo de tempo, foi criada uma lista com os *values* representativos de cada resposta, presentes no *Map* das mesmas.

Usando um *foreach* sobre esta lista, é testado se uma *ANSWER* pertence ao intervalo e, caso aconteça, é adicionada à mesma lista, ordenadamente, consoante o seu *score*. No final, basta retirar os *N ID's* de respostas com melhor *score* (equivalente às mais votadas).

3.0.9 Most Answered Questions

Posteriormente à filtragem das respostas pertencentes ao intervalo de tempo, percorrendo o *Map* das questões, seriam adicionadas, a um novo *Map*, as questões cuja data de publicação se enquadrasse no intervalo temporal.

Em seguida, era percorrida a lista que contém as respostas, em que se analisa se a pergunta a que responde existia no *Map* acima obtido. Caso isso acontecesse, seria incrementada uma unidade no contador.

Retirando os elementos deste *Map* de questões para uma lista, ordenando pelo contador, é possível obter as questões mais respondidas naquele intervalo temporal.

3.0.10 Contains Word

Para resolver a *query* seguinte, foi pensado numa forma semelhante à *query 4*, mas agora foram acrescentados espaços ao título de cada questão, convertendo, também, os caracteres que não estivessem entre 'a' e 'z' em espaços. Foi, então, usando o método *split("\\s")* de modo a dividir a *String* do título em *sub-strings*, a cada ocorrência de um espaço. Finalmente, bastou validar se a palavra ocorria no título de uma questão percorrendo o resultado da aplicação do *split*, onde, caso acontecesse, seria guardada numa lista ordenada pelas datas, à qual, no final, são copiados para uma *List<Long>* os *ID's* das *N* questões mais recentes.

3.0.11 Both Participated

Para conseguir obter uma lista com os *posts* em que ambos os utilizadores do input participaram, foi decidido que as questões dos mesmos seriam guardadas em dois *Maps* e que as respostas seriam guardadas na forma da pergunta correspondente. Assim, utilizando uma função que confirma se os elementos do *Map* de menor dimensão ocorrem no maior, é possível gerar uma lista com as questões em que ambos os *users* participam, ordenada em termos temporais, de forma rápida e eficiente.

3.0.12 Better Answer

De modo a ser obtida a melhor resposta a uma questão, ou seja, segundo o enunciado, a resposta que tenha o valor de ***(score * 0.65) + (reputação * 0.25) + (comentários * 0.1)*** mais elevado, o método adotado foi percorrer as respostas da questão, calculando os valores necessários e inserindo-as numa lista auxiliar, ordenada por este.

Posteriormente, bastou devolver a cabeça desta lista, já que representa a questão com o valor mais elevado.

3.0.13 Most Used Best Rep

Sabendo que o output pedido era uma *List<Long>* com os *ID's* das *N tags* mais usadas pelos *N users* com melhor reputação, o algoritmo de resolução passou por criar listas auxiliares que estivessem ordenadas pelos parâmetros mais convenientes: uma com os utilizadores ordenados pela reputação; outra com as questões dos *N* utilizadores com melhor reputação, no determinado intervalo de tempo; e outra com as *TAGS* e respetivos contadores, inicializados a zero. De notar que foram acrescentados < > à *String* da *tag*, de modo a facilitar a procura (método utilizado na *query 4*). Percorrendo a lista das questões para cada *TAG*, é possível incrementar ao valor do contador quando encontra uma correspondência. Finalmente, os *ID's* das *tags* mais utilizadas são colocados numa *List<Long>*, sendo que, as que tiverem zero ocorrências, não aparecerão.

3.0.14 Clean

Os quatro *Maps* com o conteúdo lido dos ficheiros *XML* foram libertados de memória usando o método "clear()", pois o seu conteúdo já não será necessário após a execução das *queries* propostas, não deixando rasto de terem ocupado a mesma.

4 Conclusão

Em suma, foi proveitosa a realização do trabalho proposto, na medida em que possibilitou o desenvolvimento de aptidões relativas à utilização de uma linguagem imperativa, no caso da fase inicial em C, e de uma orientada aos objetos, neste caso, Java. Além disso, permitiu uma maior compreensão das estruturas de dados a utilizar para solucionar certos problemas que podem vir a ocorrer novamente em projetos futuros, de forma fácil e eficaz.

Apesar disso, é notório que havia aspetos que podiam ser melhorados, dentro dos quais o uso de estruturas de dados mais eficientes para resolver os desafios, tais como possíveis inclusões de árvores binárias balanceadas, ou mesmo o desenvolvimento de funções que tirassem ainda mais proveito das bibliotecas existentes em Java.

Para finalizar, já dizia Fernando Pessoa *"Adoramos a perfeição, porque não a podemos ter; repugná-la-íamos, se a tivéssemos. O perfeito é desumano, porque o humano é imperfeito.",* deste modo, apesar do projeto não estar perfeito, é de nossa opinião que este cumpriu todos os requisitos apresentados, inclusive, com tempos de execução bastante reduzidos.

5 Bibliografia

- http://www.tutorialspoint.com/java_xml/java_dom_parse_document.htm
- https://docs.oracle.com/javase/8/docs/api/