
RELATÓRIO DO PROJETO EM JAVA

LABORATÓRIOS DE INFORMÁTICA III

GRUPO 24:

HENRIQUE PEREIRA (A80261)

PEDRO MOREIRA (A82364)

PEDRO FERREIRA (A81135)

Universidade do Minho

Mestrado Integrado em Engenharia Informática



2018

Conteúdo

1	Introdução	2
2	Estrutura	3
3	Interrogações e Abordagem	4
3.1	Info From Post	5
3.2	Top Most Active	5
3.3	Total Posts	5
3.4	Questions with Tag	5
3.5	Get User Info	6
3.6	Most Voted Answers	6
3.7	Contains Word	6
3.8	Both Participated	6
3.9	Better Answer	7
3.10	Most Used Best Rep	7
4	Resultados	7
4.1	Análise dos resultados	7
4.2	Comparação dos Resultados	7
5	Conclusão	9

1 Introdução

No âmbito da unidade curricular Laboratórios de Informática III, do 2º ano do Mestrado Integrado em Engenharia Informática, foi-nos proposta a realização de um projeto que consiste no desenvolvimento de um sistema capaz de processar ficheiros XML, armazenando as várias informações utilizadas pelo Stack Overflow (uma das comunidades de perguntas e respostas mais utilizadas atualmente por programadores em todo o mundo). Além disso, após o tratamento da informação, o sistema teria de ser capaz de responder eficientemente a um conjunto de interrogações (explicitado na secção 3). Esta aplicação teria que ser obrigatoriamente desenvolvida em *Java*.

As estruturas por nós definidas serão apresentadas e justificadas na secção 2.

Um ponto fulcral deste trabalho é conciliar a eficiência com o encapsulamento, sem comprometer nenhum destes.

2 Estrutura

Para a realização da segunda fase do projeto tivemos de definir várias classes de dados em *Java*, de maneira a respeitarmos o enunciado, no que toca aos tipos concretos e abstratos de dados.

```
public class TCD implements li3.TADCommunity {
    private Map<Long, User> users;
    private Map<PostKey, Post> posts;
    private Map<Long, String> postAux;
    private Map<String, Tag> tags;
    ...
}
```

A nossa decisão de utilizar *HashMap* vai de encontro ao propósito de aceder diretamente à informação por nós pretendida, no caso de um utilizador, a partir de um ID, no caso de uma Tag, a partir do seu nome. Porém, para os Posts, decidimos utilizar uma estrutura diferente, neste caso um *TreeMap*, pois necessitávamos de uma ordem (neste caso cronológica) de recolha da informação dos Posts da estrutura que faz uma associação do tipo chave-valor, sendo a ordem dos elementos estabelecida pela comparação das chaves. No entanto, sendo os Posts uma unidade central no enunciado proposto, o grupo deparou-se com a necessidade de obter informação relativa a um certo Post, dado o seu *ID*, da forma mais eficiente possível. Claramente, realizar uma travessia (*in-order*) na árvore até encontrar o desejado não seria uma boa solução. De forma a otimizar o pretendido, foi adicionada à estrutura *TCD* um *HashMap* cuja chave corresponde ao ID de um Post e o valor armazenado ao *timestamp* do mesmo, considerado até ao milissegundo. Deste modo, por comparação de chaves da árvore seria possível efetuar uma procura binária. Contudo verificou-se a existência de diferentes posts cujos *timestamps* eram iguais, e a estrutura usada para os armazenar não contempla chaves repetidas. Assim, foi criada a estrutura *postKey*, também esta tornada opaca e devidamente encapsulada, apenas contendo a data e o ID de um Post, que passou a ser usada como chave de inserção na árvore. Desta forma garantiu-se não só a ordenação pretendida como a unicidade de cada uma das chaves. No caso de dois Posts terem o mesmo *timestamp*, considerou-se como sendo maior o Post com menor ID. Para efeitos de procura de um Post, foi criada a função *getPost*, cuja execução se divide nas duas tarefas descritas anteriormente: dado o ID de um Post, criar uma variável do tipo *postKey*, com o ID e o *timestamp* presente na tabela de Hash *postAux* e, de seguida, proceder a uma procura binária na árvore dos Posts.

```
public class User {
    private long id;
    private String display_name;
    private int n_posts;
    private int reputacao;
```

```

        private String short_bio;
        private Set<Post> user_posts;

        ...
    }

    public class Post {
        private long id;
        private String titulo;
        private long owner_id;
        private int type_id;
        private long parent_id;
        private LocalDateTime data;
        private List<Tag> tags;
        private int score;
        private int n_comments;
        private int n_answers;

        ...
    }

    public class PostKey implements Comparable<PostKey>{
        private LocalDateTime data;
        private long id;

        ...
    }

    public class Tag {
        private String nome;
        private long id;

        ...
    }

```

3 Interrogações e Abordagem

Nesta secção, iremos explicitar a nossa abordagem às interrogações descritas no enunciado, desta vez na linguagem *Java*.

A primeira funcionalidade da aplicação que tivemos que implementar foi o carregamento dos dados a partir dos ficheiros XML. Para tal, assumimos uma abordagem diferente daquela que apresentamos aquando da primeira fase do Projeto em *C*. Isto porque, em vez de carregarmos os documentos todos para a memória, o que seria uma má implementação para ficheiros de grande tamanho, decidimos utilizar a API do Java que permite este carregamento, mas de uma maneira faseada, ou seja, por blocos. A API utilizada foi, portanto, o **StAX** (*Streaming API for XML*), que permite parsing de ficheiros XML através de *streaming*. Preferimos o **StAX** face ao **SAX** pela facilidade de utilização do primeiro.

Assim sendo, utilizando esta API, fizemos o carregamento dos dados presentes no ficheiros XML para as nossas estruturas, de forma a podermos responder às várias *queries* apresentadas.

3.1 Info From Post

A resolução da primeira *query* foi semelhante à estratégia adotada no projeto anterior. Através do método *getPost*, presente na classe *TCD*, tentamos aceder à informação relativa ao *ID* passado como parâmetro. Caso o *post* não exista, é lançada uma exceção que alertará o utilizador para a situação (*PostNotFoundException*). Caso contrário é efetuada uma tentativa de aceder às informações do criador do *post*, através do método *getUser*, ou é repetido o processo caso o *post* seja uma resposta, desta vez, procurando dados sobre a respetiva pergunta.

3.2 Top Most Active

Para obter a resposta desejada utilizamos como estrutura intermédia um *Tre-eSet* criado através do seu construtor que permite que lhe seja passado um critério de ordenação como parâmetro. Para tal efeito, recorremos a uma *lambda expression* de forma a suprimir a necessidade de criar uma classe especificamente para se declarar um novo comparador ou usar uma classe interna anónima, cuja leitura é complicada. De seguida inserimos todos os utilizadores nesta árvore através do método *addAll*, e procedemos ao tratamento dos dados sobre a forma de *stream*. Limitamos os elementos da mesma ao número de utilizadores que pretendemos obter e, de seguida, alteramos o tipo dos dados a circular na *stream* para os seus *ID's*, inserindo-os numa lista de imediato.

3.3 Total Posts

Esta *query* foi resolvida tendo por base o mecanismo funcional que o Java disponibiliza. Fizemos *stream* dos *posts* e filtramos todos aqueles que ocorreram entre as datas passadas como argumento. De seguida, com recurso ao método *groupingBy* serapamo os *posts* filtrados em duas listas diferentes sendo que a resposta é construída com o tamanho de cada uma das listas.

3.4 Questions with Tag

Mais uma vez, a estratégia usada guia-se pelas construções funcionais disponibilizados no *Java 8*. A resolução passa por fazer *stream* dos *post's*, reter apenas aqueles que ocorreram no intervalo de datas passado como parâmetro e que contenham a tag dada. De seguida, finalizamos colecionando os *ID's* dos mesmo para uma lista.

3.5 Get User Info

Esta *query* tem resolução quase imediata. Simplesmente acedemos à informação respetiva ao utilizador cujo *ID* foi passado como argumento, sendo lançada a respetiva exceção caso este não exista. Obtemos os seus últimos dez *post's* recorrendo ao mecanismo funcional e por fim montamos o resultado.

3.6 Most Voted Answers

Para solucionar esta *query* criamos um *TreeSet* cujo critério de ordenação tem por base o *Score* dos *post's* que lá vão ser inseridos. Como critério de desempate, utilizamos a ordem crescente dos *ID's*. Posteriormente, todas as respostas ocorridas entre as datas passadas como argumento são inseridas no mesmo. Para finalizar, retiramos os *N* primeiros elementos do *TreeSet* e construímos uma lista com os seus *ID's*.

A resolução desta *query* é em tudo idêntica à da *query* anterior, à exceção do critério de ordenação usado no *TreeSet*. Neste caso, o *TreeSet* é populado apenas com perguntas, cuja ordem é estabelecida pelo número de respostas que estas obtiveram.

```
public List<Long> mostAnsweredQuestions(int N,
                                         LocalDate begin, LocalDate end);
```

3.7 Contains Word

Através das construções funcionais a resolução desta *query* tornou-se bastante simples, e a sua leitura altamente simplificada em relação à implementação realizada em *C*. Apenas foi preciso fazer *stream* dos *posts* e filtrar as perguntas cujo título continha a palavra passada como argumento. Para tal, usamos o método *contains* presente em *java.lang.String*. Novamente, limitamos o número de elementos na *stream* ao tamanho que foi passado como argumento e colecionamos o resultado.

3.8 Both Participated

De forma a chegar ao resultado correto, procedemos, tal como no projeto anterior, à evocação de um método auxiliar sobre a coleção de *posts* de ambos os utilizadores passados como argumento. Esse método troca todas as ocorrências de respostas pelas respetivas perguntas. Deste modo, é nos possível intercetar os *posts* em que ambos participaram recorrendo simplesmente ao método *retainAll*. Para finalizar, colecionamos, no máximo, os *N ID's* de *posts* que estes tenham em comum.

3.9 Better Answer

Nesta *query* recorreremos novamente à utilização de um *TreeSet* como estrutura de ordenação, inserido os candidatos a resposta aquando do fecho de uma *stream* de todos os *posts*, filtrada para conter apenas as respostas à pergunta cujo *ID* foi passado como argumento. No caso de essa pergunta não existir ou não ter resposta são lançadas as devidas exceções. Caso contrário é retornado o *ID* do primeiro elemento da árvore.

3.10 Most Used Best Rep

A abordagem à décima primeira (e última) *query* pode-se dividir em três fases. Numa primeira fase, é montada uma lista com todas as *tags* usadas no intervalo de tempo em causa (incluindo repetições, dado que é fundamental para o próximo passo). De seguida é criado um *Map* cujas chaves correspondem ao *ID* de uma *tag* e o valor ao número de ocorrências da mesma. Esta tarefa foi bastante simplificada fazendo uso dos métodos *groupingBy* e *counting* presentes em *java.util.stream.Collectors*.

4 Resultados

4.1 Análise dos resultados

Tabela 1: Tempo (ms) - Ubuntu		
Interrogação:	Input nº1:	Input nº2:
1	0.01	0.01
2	198.0	197.0
3	50.0	54.0
4	47.0	47.0
5	1.0	1.0
6	39.0	42.0
7	43.0	48.0
8	7.0	11.0
9	2.0	0.0
10	28.0	28.0
11	182.0	198.0

4.2 Comparação dos Resultados

De forma a ser mais fácil comparar os resultados obtidos com as nossas abordagens em *C* e *Java*, desenvolvemos os gráficos 1, 2 e 3, que representam os tempos de execução do carregamento dos dados (*load*) e das *queries* (um gráfico para cada *input*, respetivamente).

Como podemos ver pelo gráfico 1, os tempos de execução do `load` do *Java* são inferiores aos do *C*, sendo que esta diferença é mais dispar quando executamos o carregamento do *dump* do Ubuntu. Podemos observar que os tempos do Java para o Android e Ubuntu são de 2 segundos e 12 segundos, respetivamente, enquanto que os do C são, para os mesmos *dumps*, de 3 e 19 segundos.

Os gráficos 2 e 3 mostram-nos o seguinte:

- Na *Query* 1, na *Query* 5 e na *Query* 9, ambos os tempos de execução da aplicação em C e Java são muito reduzidos, quando comparados com os resultados das outras interrogações.
- As *Queries* 2 e 11, são as únicas interrogações onde o tempo de execução em Java é superior ao tempo de execução em C em cerca de 50 milissegundos. No input 1, podemos verificar também um ligeiro aumento do tempo da aplicação em Java perante o tempo daquela em C.
- Os tempos de execução das restantes *Queries* mostram que o desempenho da aplicação em Java é bastante superior à em C, diferença esta mais denotada nas interrogações 4, 8 e 10. Os resultados dos dois inputs divergem, sendo que os tempos em C são imensamente maiores no segundo input, enquanto que os tempos em Java se mantêm.

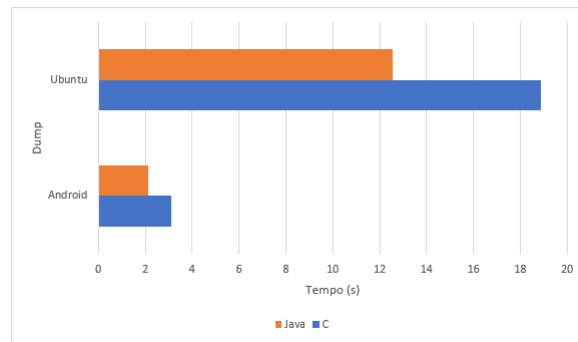


Figura 1: Tempos de Execução - Load

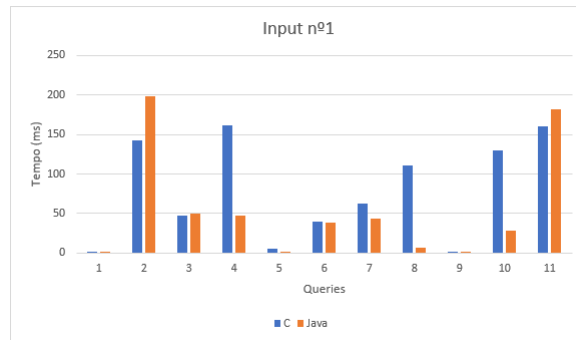


Figura 2: Tempos de Execução - Queries (Input 1)

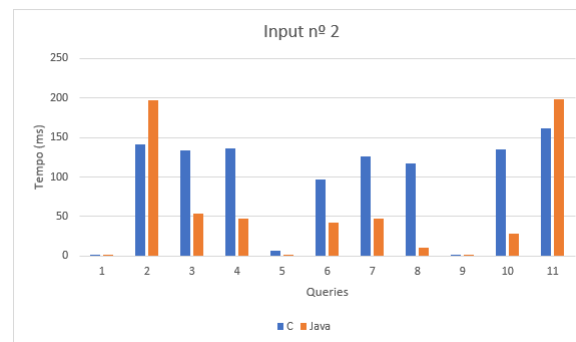


Figura 3: Tempos de Execução - Queries (Input 2)

5 Conclusão

No que toca à parte gráfica da aplicação, procuramos implementar ao máximo o modelo MVC, em que o Model é a estrutura TCD, o View é a classe Menu e o Controller é a classe que definimos com o mesmo nome. Com isto queremos dizer que, toda a interação do programa é feita no Menu, enquanto que é o Controller que faz a ligação entre o I/O e o nosso sistema, isto é, a nossa estrutura de dados, a TCD.

A elaboração deste projeto foi de certa forma mais simples e rápido de realizar do que o projeto em *C*, uma vez que o raciocínio para a realização das Interrogações já estava feito, e foi apenas necessário modificar alguns aspetos da implementação. Além disso, a existência de uma imensidão de APIs facilitou imenso a concretização do nosso raciocínio, resultando numa redução de tempo a definir funções auxiliares que seriam (e foram) necessárias no projeto em *C*.

Este projeto foi extremamente enriquecedor para o grupo, uma vez que nos permitiu trabalhar com um tipo diferente de dados, o XML, e nos ensinou a trabalhar com ele. Tivemos que nos esforçar bastante para conseguirmos

atingir os objetivos estabelecidos, tendo sempre em atenção o encapsulamento do código e a sua eficiência. O trabalho permitiu-nos também aprender a desenvolver as estruturas de que precisamos, consoante as dificuldades, exigências e necessidades que nos foram sendo impostas com o desenrolar do projeto.

Os resultados que obtivemos, ao correr o programa, foram iguais aos dos testes fornecidos pelos docentes, pelo que podemos concluir que a nossa implementação está correta. Além disso, achamos que cumprimos o objetivo de criar um programa eficiente, visto que os tempos que obtivemos para cada interrogação, utilizando os *dumps* do Ubuntu e os mesmos parâmetros utilizados pelos professores nos testes, foram os seguintes (aproximadamente):

Em suma, tendo em conta a eficiência e os resultados que obtivemos, podemos concluir que conseguimos atingir as metas propostas com este projeto, dado que os tempos de execução, tanto do *load* como das interrogações, foram bastante positivos. Com os *getters* e *setters* por nós definidos para as nossas estruturas, conseguimos sempre, com a modulação e separação dos ficheiros que fizemos, respeitar o encapsulamento da informação.