Laboratórios de Informática III - Java

Diogo Gonçalves(a81860), Pedro Gomes(a82418), Pedro Lima(a80785)

June 12, 2018

Abstract

No âmbito da unidade curricular de LI3, foi proposto um projeto inicialmente em C e, posteriormente, em Java, de modo a comparar os diferentes paradigmas de programação. Este relatório aborda a implementação do trabalho em Java e a respetiva arquitetura de classes, discutindo-se, posteriormente, os aspetos relacionados com o compromisso entre simplicidade de desenvolvimento, controlo de código e desempenho do programa. Neste relatório serão apresentadas as soluções implementadas em Java para cada *query*, a respetiva eficiência, efetuando-se, por fim, uma comparação global do problema do ponto de vista da utilização de linguagens diferentes. Salienta-se ainda a utilização de um controlador MVC e a documentação de código em *Java Doc*.

1 Introdução

O *Stack Overflow* é uma das maiores plataformas de suporte à comunidade informática, onde qualquer utilizador pode ver as questões colocadas e respondidas anteriormente ou colocar ele próprio a sua questão, de modo a obter respostas objetivas ao problema apresentado.

A ideia subjacente ao projeto de Laboratórios de Informática III (LI3) proposto é criar um programa capaz de suportar os metadados desta plataforma e efetuar 11 *queries* diferentes sobre os mesmos, tendo estes de ser acedidos e retornados o mais eficientemente possível, de modo a reduzir o tempo de execução.

Este trabalho foi implementado em duas linguagens distintas, Java (o foco principal deste relatório) e C, de modo a comparar os diferentes paradigmas. Uma vez que C é uma linguagem de baixo nível, espera-se uma maior eficiência, tendo em conta o maior controlo que o programador possui. Em Java abdica-se desse controlo, por uma simplicidade de código, tornando-se este bem mais simples, mas ao mesmo tempo mais ineficiente, algo que será abordado mais à frente no relatório.

Salientamos ainda que a implementação em Java realizada tem um controlador MVC e está toda devidamente documentado em *Java Doc*, para uma melhor interpretação por terceiros.

Este relatório está organizado da seguinte forma: na Secção 2 é descrita a arquitetura de classes e a principal (TAD_comunity); na Secção 3 são explicadas as *queries* individualmente; na Secção 4 são explicadas as decisões tomadas no trabalho numa perspetiva de otimização do desempenho do programa; na Secção 5 é feita uma comparação entre a implementação em Java e em C; na Secção 6 são expostas as conclusões do trabalho realizado.

2 Arquitetura de Classes

Neste projeto foram utilizadas múltiplas classes , de modo a guardar da forma mais acessível a informação. As principais classes são:*User* - que guarda toda a informação necessária de um utilizador; *Post* - que guarda toda a informação de uma publicação; *Tag* - que guarda a informação necessária de uma *tag* para uma realização plena da *query* 11. No entanto, existe uma classe principal, a TAD_comunity, que agrupa todas as outras classes nas suas variáveis de instância: 3 *maps* e 1 *set*.

Em mais detalhe:

- os *maps* são usadas para: (i) os *posts*, onde o *id* do *post* é a *key*, que permite encontrar um *post* completo; (ii) os *users*, onde o *id* do *user* é a *key*, que permite encontrar um *user*; (iii) as *tags*, onde o nome da *tag* é a *key*, que permite encontrar uma *tag*;

- o *set* guarda todos os *posts* presentes no *map* organizados por ordem cronológica.

3 Queries

Nesta secção serão explicadas todas as *queries* individualmente, discutindo-se a solução implementada e o seu grau de eficiência.

3.1 Query 1

Nesta *query*, de modo a obter o *output* desejado, o título do *post* e o *username* do criador, faz-se um *get* no *Map* dos *posts* com o *id* do *post* do *input*, obtendo-se assim toda a informação deste. Se for uma resposta (se entrar no *else*, como se pode ver na Figura 1), repete-se o processo para o *id* da pergunta original. Quando já se tiver o *post* com a pergunta desejada, obtém-se diretamente o título (primeiro elemento do par a retornar) e, seguidamente, faz-se um *get* no *Map* dos *users* com o *id* do criador da pergunta, obtendo-se assim o seu *username* (segundo elemento do par a retornar).

```
//Query1
public Pair<String,String> infoFromPost(long id) {
        Post x = this.posts.get(id);
        User y = this.users.get(x.getUserId());
        //É pergunta;
        if (x.getPostType() == 1)
            return new Pair<String, String>(x.getTitle(), y.getNickname());
        else{ //É resposta;
            Post pergunta = this.posts.get(x.getParentId());
            y = this.users.get(pergunta.getUserId());
            return new Pair<String, String>(pergunta.getTitle(), y.getNickname());
        }
    }
```

Figure 1: Código da implementação da *query* 1

3.2 Query 2

Nesta *query*, de modo a obter o *output* desejado, o top N de utilizadores com maior número de *posts*, faz-se inicialmente um *stream* ao *map* dos posts e a cada *post*, vai-se ao *user* criador dele e incrementa-se o número de posts por ele feito (como se pode ver na primeira linha de código da Figura 2). De seguida, percorre-se todo *map* dos *users* e ordenam-se numa lista por número de *posts* feitos, retornando apenas os primeiros N desejados.

```
// Query 2
public List<Long> topMostActive(int N) {
    return  this.users.values().stream()
            .map(p -> p.c())
            .sorted(new ComparatorUserCount())
            .map(e -> e.getId())
            .limit(N)
            .collect(Collectors.toList());
}
```

Figure 2: Código da implementação da *query* 2

3.3 Query 3

Nesta *query*, de modo a obter o *output* desejado, o número de perguntas e respostas num certo intervalo de tempo, percorre-se a estrutura dos *posts* organizada pelo tempo. Inicializa-se na data final e percorre-se até a data inicial, uma vez que o *output* é necessário ser em cronologia inversa, e verifica-se se o *post* é uma pergunta ou resposta, incrementando a varíavel do seu tipo. No fim, retorna-se os dois valores incrementados ao longo da fusão. A Figura 3 ilustra o código correspondente.

```java
// Query 3
public Pair<Long,Long> totalPosts(LocalDate begin, LocalDate end) {

    return new Pair<>(this.setTempo.stream().filter(e -> ((e.getDate().isAfter(begin) || e.getDate().equals(begin)) &&
                        (e.getDate().isBefore(end) || e.getDate().equals(end)))).filter(e -> e.getPostType() == 1).count() ,
                      this.setTempo.stream().filter(e -> ((e.getDate().isAfter(begin) || e.getDate().equals(begin)) &&
                        (e.getDate().isBefore(end) || e.getDate().equals(end)))).filter(e -> e.getPostType() == 2).count());
}
```

Figure 3: Código da implementação da *query* 3

3.4 Query 4

Nesta *query*, de modo a obter o *output* desejado, todas as perguntas que contêm uma determinada *tag* num certo intervalo de tempo, percorre-se o *set* dos *posts* organizado por tempo. Nesta, como podemos ver na Figura 4, filtra-se as perguntas, visto que, apenas estas possuem tags, seguidamente se esse post tem a tag desejada e, por fim, se está no intervalo de tempo desejado. No fim os posts que passaram por essas 3 condições, guarda-se o seu id numa lista, sendo posteriormente retornada.

```java
// Query 4
public List<Long> questionsWithTag(String tag, LocalDate begin, LocalDate end) {
    return this.setTempo.stream()
            .filter(e -> e.getPostType() == 1)
            .filter(e -> e.getTags().contains(tag))
            .filter(e -> ((e.getDate().isAfter(begin) || e.getDate().equals(begin)) && (e.getDate().isBefore(end) || e.getDate().equals(end))))
            .map(e -> e.getId())
            .collect(Collectors.toList());
}
```

Figure 4: Código da implementação da *query* 4

3.5 Query 5

Nesta *query*, de modo a obter o *output* desejado, a *short bio* de um utilizador e os seus últimos 10 *posts*, faz-se um *get* do *id* do *user* no *map* dos *users*, obtendo-se diretamente a *bio*. Relativamente aos seus últimos 10 posts, percorre-se o *set* dos *posts* organizado por tempo e verifica-se se o criador é o *id* do input. Como a estrutura é organizada com a cronologia inversa, as 10 primeiras ocorrências são as desejadas. A Figura 5 ilustra o código correspondente.

```java
// Query 5
public Pair<String, List<Long>> getUserInfo(long id) {
    return new Pair<String, List<Long>>(this.users.get(id).getShortBio(),
            this.setTempo.stream().filter(p -> (p.getUserId() == id)).map(p -> p.getId()).limit(10).collect(Collectors.toList()));
}
```

Figure 5: Código da implementação da *query* 5

3.6 Query 6

Nesta *query*, de modo a obter o *output* desejado, as N respostas com mais votos num determinado intervalo de tempo, percorre-se o *set* dos *posts* organizado por tempo. Inicialmente filtra-se por datas, ficando apenas os *posts* do intervalo desejado e seguidamente, filtra-se as apenas respostas. No fim organiza-se por *score*, guardando apenas os N primeiros desejados. A Figura 6 ilustra o código correspondente.

```java
// Query 6
public List<Long> mostVotedAnswers(int N, LocalDate begin, LocalDate end) {
    return this.setTempo.stream()
            .filter(e -> ((e.getDate().isAfter(begin) || e.getDate().equals(begin)) && (e.getDate().isBefore(end) || e.getDate().equals(end))))
            .filter(e -> (e.getPostType() == 2))
            .sorted(new ComparadorScore())
            .map(e -> e.getId())
            .limit(N)
            .collect(Collectors.toList());
}
```

Figure 6: Código da implementação da *query* 6

3.7 Query 7

Nesta *query*, de modo a obter o *output* desejado, as N perguntas com mais respostas num determinado intervalo de tempo, percorre-se o *set* dos *posts* organizado por tempo. Neste filtra-se os *posts* de modo a ficarem apenas os que estão dentro do intervalo desejado e os que são perguntas (como se vê na Figura 7). De seguida organizam-se por número de respostas que cada pergunta possui (é uma variável de instância da classe *post*) e no fim retornam-se os N primeiros desejados.

```java
// Query 7
public List<Long> mostAnsweredQuestions(int N, LocalDate begin, LocalDate end) {
    return this.setTempo.stream()
            .filter(e -> ((e.getDate().isAfter(begin) || e.getDate().equals(begin)) && (e.getDate().isBefore(end) || e.getDate().equals(end))))
            .filter(e -> (e.getPostType() == 1))
            .sorted(new ComparatorAnswerCount())
            .map(e -> e.getId())
            .limit(N)
            .collect(Collectors.toList());
}
```

Figure 7: Código da implementação da *query* 7

3.8 Query 8

Nesta *query*, de modo a obter o *output* desejado, todas as perguntas que contêm uma determinada palavra no título, percorre-se o *set* dos *posts* organizado por tempo. Nesta filtra-se as perguntas, visto que apenas estas têm título e, caso possuam a palavra desejada, é adicionada à lista final o *id* dessa pergunta(como se pode ver na Figura 8).

```java
// Query 8
public List<Long> containsWord(int N, String word) {
    return this.setTempo.stream()
            .filter(e -> (e.getPostType() == 1))
            .filter(e -> e.getTitle().contains(word))
            .sorted(new ComparatorDataReverse())
            .map(e -> e.getId())
            .limit(N)
            .collect(Collectors.toList());
}
```

Figure 8: Código da implementação da *query* 8

3.9 Query 9

Nesta *query*, de modo a obter o output desejado, as últimas N perguntas que dois utilizadores interagiram, percorre-se o *set* dos *posts* organizado por tempo, criando duas listas, guardando os ids das perguntas que cada um participou (quer porque criou a própria pergunta, ou porque a respondeu). De seguida guarda-se numa terceira lista, os ids em comum entre os dois users, que no fim é retornada retirando os elementos repetidos (caso um user responda mais que uma vez, ou faça a pergunta e também a responda), devolvendo apenas os N elementos desejados. A Figura 9 ilustra o código correspondente.

```java
// Query 9
public List<Long> bothParticipated(int N, long id1, long id2) {
    List<Long> myid2 = new ArrayList<>();
    myid2 = getPergunta(id2);
    List<Long> myid1 = new ArrayList<>();
    myid1 = getPergunta(id1);
    List<Long> myList = new ArrayList<>();

    for (Long e : myid1){
        if (myid2.contains(e))
            myList.add(e);
    }

    return myList.stream()
            .distinct()
            .collect(Collectors.toList());
}
```

Figure 9: Código da implementação da *query* 9

3.10 Query 10

Nesta *query*, de modo a obter o output desejado, a melhor resposta para uma certa pergunta, percorre-se o *set* dos *posts* organizado por tempo, filtrando para uma lista apenas as respostas cuja pergunta original é a desejada. A estas aplica-se a fórmula de cálculo para obter a "qualidade" da resposta, organizando no final por esse parâmetro e retorna-se a primeira (a melhor). A Figura 10 ilustra o código correspondente.

```java
// Query 10
public long betterAnswer(long id) {
        return this.setTempo.stream()
                .filter(p -> p.getParentId() == id)
                .map(p -> p.clone())
                .map(p -> calculateAnswer(p))
                .sorted(new ComparatorBestAnswer())
                .map(p -> p.getId())
                .collect(Collectors.toList())
                .get(0);
}
```

Figure 10: Código da implementação da *query* 10

3.11 Query 11

Nesta *query*, de modo a obter o output desejado, as N tags mais utilizadas pelos N utilizadores com mais reputação, dentro de um certo intervalo de tempo, percorre-se o *set* dos *posts* organizado por tempo, de modo a obter uma lista com os *users* de maior reputação que participaram dentro desse intervalo de tempo, quer por pergunta ou por resposta. Seguidamente cria-se uma lista de *strings*, com todas as *tags* utilizadas dentro desse intervalo, incrementando o seu número de aparições. No final cria-se uma lista com o *id* das *tags* mais utilizadas e retorna-se as N primeiras desejadas.

```java
// Query 11
public List<Long> mostUsedBestRep(int N, LocalDate begin, LocalDate end) {

    Map<String, Tag> tag_query = new HashMap<>();

    List<Long> myList = getTopUsers(N ,begin , end);
    List<String> myStringTag = getPostWithTags(myList , begin , end);

    StringBuilder all_tags_string = new StringBuilder();

    for (String e : myStringTag)
        all_tags_string.append(e);


    List <String> all_tags = SplitString(all_tags_string.toString());

    all_tags.forEach(p -> {
        if(this.tags.get(p) != null)
            if (tag_query.get(p) == null)
                tag_query.put(p, this.tags.get(p).clone());
            tag_query.get(p).incCount();

    });

    return  tag_query.values()
            .stream()
            .sorted(new ComparatorCount())
            .map(e -> e.getId())
            .limit(N)
            .collect(Collectors.toList());
}
```

Figure 11: Código da implementação da *query* 11

4 Análise de desempenho

Como este projeto em Java foi desenvolvido posteriormente ao de C, houve uma maior consciência para a eficiência do programa, visto que era algo fundamental na versão anterior e muitas soluções concebidas poderiam ser re-implementadas em Java. Assim sendo decidiu-se optar pelas "mesmas" estruturas, agora adaptadas para as coleções de *Map*, *List* e *Set*, que nos permitiram aproveitar ao máximo a eficiência delas e todo o código já pré-definido. A Figura 11 ilustra o código correspondente.

Em mais detalhe:

1. Uso de *Map* para guardar a informação proveniente do *parser* dos ficheiros. Uma vez que a busca de informação por *id* é algo muito presente no trabalho, esta coleção permite-nos um acesso instantâneo à informação usando esse parâmetro.

2. Uso de *Set* para guardar os *Posts* por ordem cronológica inversa. Uma vez que os intervalos temporais estão continuamente presentes nas queries é algo que permite um acesso mais eficiente à informação, visto que já está ordenada no formato desejado. Nesta versão, o uso desta coleção, não foi tão essencial uma vez que em algumas *queries* o *set* é percorrido totalmente e filtrado para dentro do intervalo, ou seja, é irrelevante se está organizado por datas ou não. Todavia, em *queries* como a 4, 5 e 8, que é necessário devolver os resultados por ordem cronológica inversa e apenas é necessário verificar uma certa condição, percorre-se o *Set* e se esta se verificar é adicionado a uma lista, que ficará automaticamente ordenada, evitando assim percorrer no final a lista para a ordenar.

Após todos estas decisões, o trabalho ficou com um desempenho que consideramos formidável, sendo os tempos de execução mínimos, na ordem dos milisegundos.

5 Java versus C

Nesta secção é feita uma reflexão sobre a elaboração deste projeto, primeiro em C e por fim em Java, comparando os diferentes paradigmas de programação.

Como seria de esperar, o trabalho desenvolvido em C possui um tempo de execução menor (ver Figura 12, uma vez que é uma linguagem de baixo nível. Isto permite-nos um maior controlo de todo o código, que foi implementado com o intuito de, neste caso, resolver 11 queries. Este é bastante

específico para o trabalho, fazendo-o da forma mais eficiente possível, mas impossibilitando a sua reutilização. Este é também mais difícil de desenvolver, uma vez que tem de ser feito tudo de raiz para o que é desejado, estando assim sujeito a vários problemas que têm de ser corrigidos pelo programador.

Em java, deparamo-nos com o compromisso entre o controlo do código e a eficiência do mesmo, que leva a piores tempo de execução (como pode ser observado na Figura 12). Este código, como implementa coleções já definidas pelo java, torna-se menos específico para o caso utilizado, o que apesar de tornar o seu desenvolvimento bem mais simples, não é o programador que tem de resolver todos os problemas, tem o preço de o tornar mais lento, por fazer operações extra.

```
QUERY 1 -> 0.004000 ms
QUERY 1 -> 0.002000 ms
QUERY 2 -> 0.002000 ms
QUERY 2 -> 0.006000 ms
QUERY 3 -> 0.676000 ms
QUERY 3 -> 8.238000 ms
QUERY 4 -> 0.919000 ms
QUERY 4 -> 0.828000 ms
QUERY 5 -> 0.003000 ms
QUERY 5 -> 0.001000 ms
QUERY 6 -> 0.979000 ms
QUERY 6 -> 0.598000 ms
QUERY 7 -> 0.640000 ms
QUERY 7 -> 45.194000 ms
QUERY 8 -> 50.161999 ms
QUERY 8 -> 50.695000 ms
QUERY 9 -> 48.800999 ms
QUERY 9 -> 49.868000 ms
QUERY 10 -> 0.001000 ms
QUERY 10 -> 0.001000 ms
QUERY 11 -> 3.956000 ms
QUERY 11 -> 69.587997 ms
-------------------------
```

```
LOAD -> 10777 ms
Query 1: -> 1 ms
Query 1: -> 0 ms
Query 2 -> 108 ms
Query 2 -> 79 ms
Query 3 -> 141 ms
Query 3 -> 51 ms
Query 4 -> 44 ms
Query 4 -> 39 ms
Query 5 -> 8 ms
Query 5 -> 5 ms
Query6 -> 53 ms
Query6 -> 35 ms
Query 7 -> 46 ms
Query 7 -> 43 ms
Query 8 -> 8 ms
Query 8 -> 15 ms
Query9 -> 51 ms
Query9 -> 47 ms
Query 10 -> 30 ms
Query 10 -> 26 ms
Query 11 -> 86 ms
Query 11 -> 103 ms
CLEAN -> 0 ms
```

Figure 12: Código da implementação da *query* 9

6 Conclusões

Este trabalho centrou-se no desenvolvimento de uma programa em Java capaz de responder a 11 *queries* relativas a metadados previamente fornecidos pelo stack overflow, no menor tempo de execução possível. Atendendo a que este trabalho já tinha sido realizado em C, o principal intuito com o mesmo, foi permitir obter uma perceção prática de como linguagens diferentes criam abordagens distintas ao mesmo problema.

Para cumprir este objetivo, apresentamos neste relatório as soluções implementadas em Java para dar resposta às *queries* da forma mais eficiente possível. Posteriormente, discutimos a eficiência das mesmas e finalizamos com uma comparação direta com a respetiva versão em C. Gostaríamos de realçar que, relativamente ao compromisso eficiência-simplicidade, consideramos Java a opção mais favorável, uma vez que os tempos de execução são bastante próximos e a simplicidade da implementação é substancialmente superior, resultando num programa mais elegante.

Por fim, realçamos o impacto que este trabalho teve nos nossos conhecimentos como programadores, capazes de olhar para um mesmo problema com duas abordagens claramente distintas.