Universidade do Minho

Departamento de Informática

Laboratórios de Informática III

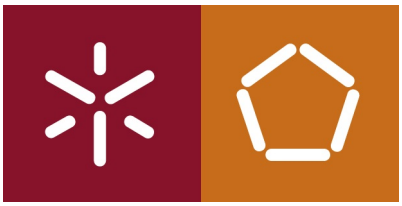
Relatório Projeto LI3 - 2ª Fase

Grupo 25
Gil Cunha Nuno Faria Tânia Silva

Resumo

Este relatório irá abordar a segunda fase do projeto de LI3 (JAVA). Falará quais foram as escolhas feitas para a sua realização bem como as justificações necessárias e os diferentes desafios encontrados.

10 de Junho de 2017



Conteúdo

1 Introdução

Neste relatório iremos falar sobre a realização da 2ª fase do projeto de Laboratórios de Informática III. É semelhante ao trabalho realizado na primeira fase, à excepção que este é feito em *Java*. Abordaremos a estrutura utilizada, a maneira como chegamos à solução e os diferentes caminhos que percorremos para tentar minimizar o tempo de execução, bem como as dificuldades que foram surgindo.

2 Parse

O *parse* que foi usado na primeira fase (*libxml2*) não podia ser utilizado nesta, visto que não foi desenvolvido para *Java*. Por isso, tivemos que procurar um novo.

Das várias alternativas que existiam, salientavam-se algumas:

- `SAX` - ler apenas quando era necessário, descartando o que leu anteriormente à medida que percorre o ficheiro;
- `StAX` - semelhante a `SAX`;
- `DOM` - cria uma árvore do *parse* feito, podendo aceder a qualquer parte do ficheiro a qualquer altura.

Como o tipo de *parse* de `DOM` era semelhante ao `libxml2` (no sentido em que ambos criam uma árvore do ficheiro) optamos primeiramente por este. Contudo, o que se sucedia era que como um ficheiro era todo carregado em memória, existiam problemas de `OutOfMemory`, sendo por isso este método descartado.

Decidimos por isso usar o `StAX`[1]. Tínhamos no entanto duas opções: `StAX` por `events` ou `StAX` por `streams`. Tentamos as duas opções, concluindo que o `StAX` por `streams` era muito mais rápido que a alterativa (verificamos uma descida no tempo de quase 8s).

Escolhendo finalmente a forma de como o *parse* seria feito, precisamos agora de percorrer o ficheiro. Como neste tipo de parse **não existe hierarquia** (por exemplo, não é possível fazer `current_node.getChildren()` ou algo do género), temos que criar booleanos que nos indicam em que parte do artigo estamos. Por exemplo, para saber que a próxima *tag* a ler é o texto de um artigo, as variáveis `rev` e `text` tem que ser `true`. Esta foi por isso a maior dificuldade no *parse*.

[1] `https://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/tutorial/doc/SJSXP2.html`

3 Modularidade

A estrutura do *package* `engine` tem o seguinte diagrama:

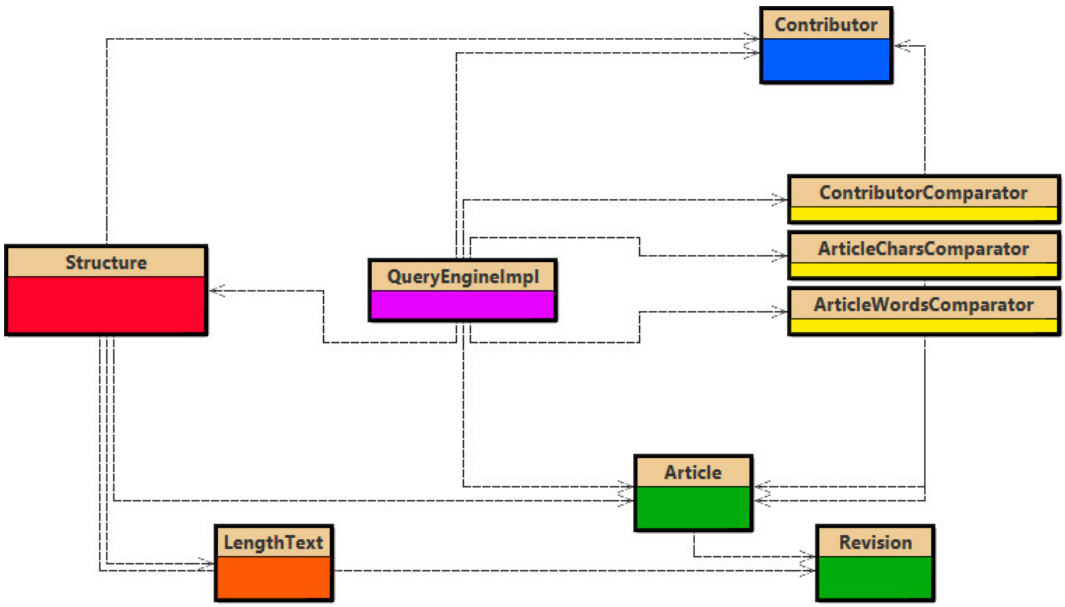


Figura 1: Estrutura de `engine`

Para guardar a informação de cada artigo teremos a classe `Article`, sendo composta pela classe `Revision`. Em `Contributor` guardamos a informação de um contribuidor.

`Structure` armazenará todos os artigos e contribuidores, utilizando por isso `Article` e `Contributor`. É composta também pela classe que conta caracteres e palavras `LengthText`.

Por ultimo temos a classe que implementa a interface `Interface` (que contem o conjunto das *queries*), fazendo uso da estrutura, artigos, contribuidores e dos comparadores `ContributorComparator`, `ArticleCharsComparator` e `ArticleWordsComparator`

Não houve nenhuma modificação dos *packages* `li3` e `common`.

4 Classes e Estrutura

4.1 `Structure`

Classe que contem a estrutura geral do projeto. Como concluímos na primeira fase que as tabelas de *Hash* era uma boa forma de organizar os dados, optamos novamente por essa estrutura.

Temos assim as seguintes variáveis de instância:

```
1 private long numberOfPages; ///< Artigos totais
2 private long uniqueArticles; ///< Artigos unicos
3 private long totalRevisions; ///< Revisoes totais
4 private HashMap<Long, Article> articles; ///< Artigos
5 private HashMap<Long, Contributor> contributors; ///< Cont.
6 private HashMap<Character, ArrayList<String>> titles;
   ///<Titulos
```

As tabelas de artigos e contribuidores serão indexadas a partir do respetivo `ID`. A tabela que contem os títulos é indexada pelo caractere inicial.

4.2 Article

Classe que organiza a informação de um artigo. É composta pelas seguintes variáveis:

```
1 private long id; ///< ID do artigo
2 private long lastRevision; ///< ID da ultima revisao
3 private long length; ///< Tamanho em carateres
4 private long lengthW; ///< Tamanho em palavras
5 private String title; ///< Titulo
6 private LinkedList<Revision> revisions = new LinkedList<>();
   ///< Revisoes
```

Cada artigo irá conter também o histórico de todas as revisões feitas, sendo composto pela classe `Revision`.

4.2.1 Revision

Guardará todas as revisões de um artigo. Terá apenas o id e a data, não sendo necessário guardar títulos de versões anteriores (todas as *queries* que fazem uso dos títulos usam sempre a versão mais recente). Tem por isso as seguintes variáveis:

```
1 private long id; ///< ID
2 private String timestamp; ///< Data
```

4.3 Contributor

Organizará a informação de um contribuidor. Só precisamos de guardar o seu ID, nome e o número de contribuições:

```
1 private long id; ///< ID do contribuidor
2 private long count; ///< Numero de contribuicoes
3 private String username; ///< Nome do contribuidor
```

4.4 LengthText

Precisamos de contar o número de caracteres e palavras em cada texto. Para não estar a fazer isso no *parse*, criamos uma classe à parte. Será constituída por métodos de classe, não sendo necessário criar uma instância.

É composto pelas seguintes variáveis,

```
1 private static long lengthChars; ///< Numero de caracteres
2 private static long lengthWords; ///< Numero de palavras
```

e os seguintes métodos

```
1 private static int utf8ToBytes (char c); //N bytes em 'c'
2 public static void count(String text); //Conta tamanhos
3 public static long getLengthChars(); //Devolve n caracteres
4 public static long getLengthWords(); //Devolve n palavras
```

A sua utilização é feita da seguinte forma: é invocado o método `count` sobre uma *string*, ou seja, `LengthText.count(text)`. Quando precisamos dos tamanhos, simplesmente fazemos `LengthText.getLengthChars()` ou `LengthText.getLengthWords`.

4.5 Comparadores

Visto que precisamos de calcular *tops*, será necessário fazer a ordenação de artigos e contribuidores. Visto que estamos em *Java*, podemos fazer uso de *TreeSet*. Para isso, criamos comparadores para indicar à estrutura de que forma terá que fazer a ordenação.

Fizemos por isso três comparadores;

- `ArticleCharsComparator` - comparação de número de caracteres;
- `ArticleWordsComparator` - comparação de número de palavras;
- `ContributorComparator` - comparação de número de contribuições;

4.6 `QueryEngineImpl`

Classe que irá implementar todas as *queries* do projeto, bem como o `init`, `load` e `clean`. Contem apenas uma variável, `qs`, que guarda a estrutura.

```
1 private Structure qs;
```

4.6.1 `all_articles, unique_articles, all_revisions`

Como à medida que percorremos o ficheiro iremos automaticamente guardar o número de artigos totais, artigos únicos e revisões, a solução destas três *queries* é trivial, sendo apenas preciso retornar a respetiva variável.

4.6.2 `contributor_name, article_title, article_timestamp`

Ao organizar os artigos e contribuidores por tabelas de *Hash*, garantimos que a sua procura será feita em $O(1)$. Sendo assim, estas *queries* tornam-se simples, acendendo à sua posição respetiva (através de *`tabela`*`.get(`*`id`*`)`) e devolvendo o parâmetro pretendido (no caso da data do artigo precisamos ainda de percorrer o *arraylist* das revisões).

4.6.3 *Tops*

Para a realização das *queries* de *tops* decidimos fazer uso dos `TreeSet` do *Java*. Criamos os comparadores para cada um e depois foi só preciso percorrer as tabelas e fazer a inserção na árvore.

Para tornar as inserções mais eficientes, foram feitas algumas alterações, que iremos falar mais à frente.

4.6.4 `titles_with_prefix`

Nesta *query* teremos que retornar todos os títulos que tenham um determinado prefixo. De modo a tornar a procura mais rápida, decidimos separar os artigos pelo seu **caractere inicial**. Fizemos essa separação num `HashMap`, onde a chave será o *char* inicial e os valores serão `ArrayList` de títulos.

A inserção de títulos é feita no fim do *parse*, visto que toda a informação já foi inserida e não haverá mais nenhuma alteração dos títulos.

Através desta simples forma de organização, o tempo de execução será muito menor comparativamente à procura pelos títulos todos.

5 Melhoramento do desempenho

Neste secção iremos abordar algumas medidas que tomamos para tornar a execução do programa mais eficiente.

5.1 Leitura de textos

A maior parte do conteúdo de um *snapshot* é constituído pelo texto do artigo. Por isso, uma poupança de tempo pequena na leitura de cada texto irá acabar por poupar muito tempo no final.

Temos que calcular o número de caracteres e palavras. Para isso, não há alternativa se não percorrer o texto todo. Inicialmente, como uma *String* em *Java* guarda o tamanho dos caracteres, apenas precisamos de percorrer o texto para contar palavras (havendo menos uma variável a incrementar por cada ciclo). Contudo, o resultado obtido era errado. Após alguma análise do que se tava a passar, verificamos que a codificação do que era lido estava em `UTF-8`.

Para ultrapassar este desafio, simplesmente fazemos `getBytes().length`, obtendo assim o tamanho do texto em `ASCII`. Porém, isto acabou por trazer mais outro problema: o desempenho piorou em alguns segundos. O facto de se estar a fazer `getBytes()` e depois percorrer o texto para contar as palavras, significaria que o trabalho era duplicado (e o tempo também). Tentamos depois mudar a codificação da leitura, através de `input.createXMLStreamReader(stream_input), "ASCII"`, que apesar de ser apenas preciso `length()`, o tempo tornou-se ainda pior.

Teria que haver uma forma de conseguir os dois tamanhos com apenas uma travessia do texto. Se conseguíssemos descobrir tamanho em *bytes* de cada caractere, podíamos somar esse valor ao total (porém, não existe um método `getBytes()` para *char*). Descobrimos que, tal como em *C*, o tipo *char* é representado por um inteiro, no intervalo `[0, 65535]`[2]. Podemos então analisar o número que representa cada um e calcular a quantidade de *bytes*. A partir da seguinte tabela[3] podemos descobrir o tamanho de cada caractere, conseguindo assim calcular os dois valores de uma vez só.

Number of Bytes	First Code Point	Last Code Point
1	U+0000	U+007F
2	U+0080	U+07FF
3	U+0800	U+FFFF

Tabela 1: Número de *bytes* em caracteres `UTF-8`

Teremos por isso a seguinte função:

```
1 private static int utf8ToBytes(char c){
2     if (c <= 0x7f) return 1;
3     if (c <= 0x7ff) return 2;
4     if (c <= 0xffff) return 3;
5     return 0;
6 }
```

[2] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
[3] https://en.wikipedia.org/wiki/UTF-8

5.2 *Tops*

Como o *Java* disponibiliza estruturas com `TreeSet`, a ordenação para obtenção dos *tops* seria muito mais fácil. Simplesmente criamos as classes de comparadores necessários, inicializamos um novo `TreeSet` da seguinte forma:
`TreeSet<Article> ord = new TreeSet<>(new ArticleWordsComparator())`
(exemplo para ordenação de artigos por número de palavras), e inserimos todos os artigos. No final, escolhemos, por exemplo, os 30 primeiros, ficando assim concluída a *query*.

É uma solução fácil e imediata. Contudo, não é muito eficiente, visto que irá fazer operações que não servirão de nada no resultado final. Podemos fazer melhor.

Analisemos o algoritmo de inserção do `TreeSet` : irá procurar a posição onde um elemento encaixa-se, colocando-o lá, isto em $O(lg(n))$. Neste caso, se quisermos o *top* m e a posição do elemento inserido for maior que m, ele já não serve para o nosso *top*, sendo uma operação de inserção a mais.

Para evitar as inserções que não são necessárias, podemos limitar o tamanho do `TreeSet`. Como não existe nenhuma maneira pré-definida para fazer isso, temos que ser nós a implementá-la. Enquanto o número de elementos inseridos for menor que m, continuamos a adicionar. A partir do momento que inserimos um número de elementos maior, começamos a **remover** o último elemento do `set`, através de `pollLast()` ($O(1)$). Assim já iremos melhorar o tempo de inserção. Contudo, se o elemento for menor que todos os outros, será inserido no final e depois removido, sendo realizadas *lg(m)+1* operações que não servirão para este caso. Isto também pode ser evitado, se no início de cada ciclo pusermos uma condição que verifica se o elemento atual é maior que o menor no `set`. Caso for, insere o elemento. Caso contrário, **não entra no ciclo**, reduzindo-se assim para *1*.

Tem-se assim o algoritmo:

```
1  (...)
2  TreeSet<Article> tree = new TreeSet<>(COMPARATOR);
3  int i = 0;
4
5  for (ELEMENT a: elements)
6      if (i==0 || a.getValue() >= tree.last().getValue()){
7          tree.add(a);
8          if (i > N) tree.pollLast();
9          else i++;
10     }
11 (...)
```

Após isto, verificou-se uma redução no tempo de mais de 4x.

6 Streams

Para este projeto foi pedido a implementação das *streams* do *Java 8*. Por isso, em vários métodos do nosso código, em vez de usar os ciclos-*for*, optamos pelos iteradores internos. Contudo, encontramos um problema.

Os tempos de execução pareciam-nos um bocado elevados. Para verificar se o problema era de causado pelas *streams*, decidimos comparar com os iteradores externos.

Top contribuidores

```
//Streams
    int N = 10;
    return qs.getContributors().values()
              .stream()
              .sorted(new ContributorComparator())
              .limit(N)
              .map(c -> c.getId())
              .collect(Collectors.toCollection(ArrayList::new));

//Iterador externo (nao otimizado)
    int N = 10, i=0;
    TreeSet<Contributor> tree = new TreeSet<>(new
       ContributorComparator());
    ArrayList<Long> top = new ArrayList<>(N);

    for (Contributor c: qs.getContributors().values())
        tree.add(c);

    for (int i=0; i<N; i++)
        top.add(tree.pollFirst().getId());

    return top;

//Iterador externo (otimizado)
    int N = 10, i=0;
    TreeSet<Contributor> tree = new TreeSet<>(new
       ContributorComparator());
    ArrayList<Long> top = new ArrayList<>(N);

    for (Contributor c: qs.getContributors().values())
        if (i == 0 || (c.getCount()) >=
           tree.last().getCount()){
            tree.add(c);
            if (i == N) tree.pollLast();
            else i++;
    }

    while (tree.size() > 0)
        top.add(tree.pollFirst().getId());

    return top;
```

Obtivemos os seguintes resultados:

```
Streams - 54ms
Iterador externo (não otimizado) - 14ms
Iterador externo (otimizado) - 5ms
```

Como podemos ver, o iterador interno é mais lento que os externos, não existindo possibilidade de o otimizar.

Top artigos (caracteres)[4]

```
Streams - 13ms
Iterador externo (não otimizado) - 13ms
Iterador externo (otimizado) - 4ms
```

Apesar de neste caso o iterador interno ser igual ao externo (não otimizado), somos capazes de otimizar o último de forma a ser três vezes mais rápido.

Top artigos (palavras)[5]

```
Streams - 20ms
Iterador externo (não otimizado) - 15ms
Iterador externo (otimizado) - 4ms
```

Títulos com prefixo

```
1  //Streams
2      return qs.getTitles().get(prefix.charAt(0))
3                .stream()
4                .filter(t -> t.startsWith(prefix))
5                .sorted()
6                .collect(Collectors.toCollection(ArrayList::new));
7
8  //Iterador interno
9      TreeSet<String> tree = new TreeSet<>();
10     ArrayList<String> prefixedTitles = new
          ArrayList<String>();
11
12     for (String s: qs.getTitles().get(prefix.charAt(0)))
13         if (s.startsWith(prefix))
14             tree.add(s);
15
16     while (tree.size() > 0)
17         prefixedTitles.add(tree.pollFirst());
18
19     return prefixedTitles;
```

```
Streams - 3ms
Iterador externo - 0-1ms
```

[4]Código semelhante a *Top* contribuidores
[5]Código semelhante a *Top* contribuidores

6.1 Desvantagens

Como vemos pelos resultados obtidos, os iteradores internos são mais lentos que os externos quando aplicados à nossa estrutura. O seu tempo também varia muito, sendo que enquanto os externos (não otimizados) correm no intervalo 13-15ms, estes variaram entre 13-54ms.

Outra desvantagem é que não existe forma de otimizar o código[6] tal como fizemos nos iteradores externos. Temos muita mais liberdade quando implementamos o nosso código num ciclo-*for*.

6.2 Vantagens

Apesar do tempo de execução ser pior, podemos ver pelos exemplos acima que o código fica mais compacto e mais legível. Nem sempre compensa um programa ser alguns milissegundos mais rápido e ter o código menos legível.

Exitem muitas funcionalidades pré-definidas que ao tornar o código mais compacto, facilitam o *debug* e a leitura, tais como `filter`, `sum`, `collect`, `removeIf`, `sorted`, etc.

7 Conclusão

Esta segunda fase do projeto deu para aprender como deve ser feito um trabalho em *Java* onde devemos conseguir o menor tempo de execução. Tal como em *C*, devem ser implementadas estruturas eficientes, guardar apenas o necessário e não fazer cópias desnecessárias.

Em comparação à primeira fase, esta foi mais fácil. Não tivemos que criar tabelas de *Hash* ou listas ligadas, visto que o *Java* contem muitas estruturas pré definidas eficientes e relativamente mais fáceis de usar. Também não houve necessidade de libertar a memória alocada, visto que o *garbage collector* faz isso por nós. A maior dificuldade foi no *parse* dos ficheiros, sendo este mais difícil nesta fase do que na primeira.

Uma vantagem do projeto em *C* foi o seu tempo de execução. O tempo de execução desta fase, tanto no *parse* como nas *queries* foi lento, sendo que o *parse* é mais de 3 segundos mais lento e as *queries* em *Java* andam à volta de 1-5ms, enquanto que em *C* estão entre 0-0.5ms (a única *query* mais rápida nesta fase foi a `clean`).

Também podemos aprender a usar *streams* do *Java 8*. Vimos que apesar de serem ligeiramente mais lentas que os iteradores externos, tornam o código mais compacto, facilitando a leitura e o *debug*.

Tal como na primeira fase, consideramos que cumprimos os objetivos pretendidos, sendo que resolvemos todas as *queries* num tempo relativamente bom.

[6] Se tentarmos implementar a otimização por iteradores internos (através de *forEach*), não só estaremos a desaproveitar o objetivo para que as *streams* foram criadas (o código iria ficar igual aos iteradores externos) mas também seria ainda mais lento.