Projeto LI3 em Java

Francisco Oliveira Raul Vilas Boas Vitor Peixoto

Grupo 33

Braga, 3 de Junho de 2017

Laboratórios de Informática III

Mestrado Integrado em Engenharia Informática

Universidade do Minho

Conteúdo

Capítulo 1

Introdução

Foi-nos proposto criar um projeto idêntico ao que foi criado previamente, desta vez em *Java*. O sistema deve ser capaz de extrair informação útil de vários backups da Wikipedia, tais como maiores contribuidores, artigos com mais palavras, etc.

Imediatamente após a inicialização deste projeto, os principais problemas com que nos deparamos foram:

- Como conseguir ler os backups *XML* com *Java*. Quais os *parsers* que existem e quais os melhores para o nosso caso?
- Que estruturas usar para guardar a informação obtida dos backups e como trabalhar com essas estruturas?
- Será apenas uma estrutura suficiente para responder a todas as questões?
- Como podemos responder a determinadas questões fazendo proveito do pacote *Java Stream*?

Este relatório está orientado por três secções: os problemas detetados já falados nesta introdução, uma concepção da solução onde será explorada a nossa abordagem para as resoluções dos problemas detetados e uma breve conclusão sobre o resultado final.

Capítulo 2

Concepção da Solução

2.1 Parsing

Para começar o projeto pesquisamos acerca dos *parsers* de *Java* e optamos por aplicar o DOM, inicialmente num pequeno XML de exemplo, tendo depois aplicado a um backup. Infelizmente este método, devido ao facto deste *parser* carregar a árvore XML toda para a memória trouxe problemas de "Out of Memory". Assim optamos por outro método, o StAX. Este *parser* tem uma pegada na memória muito menor traduzindo-se numa velocidade superior, no entanto não permite um acesso aleatório ao XML visto que corre o ficheiro de início a fim "removendo"informação conforme é pedido.

O nosso `XMLStreamReader` é a peça essencial do StAX. Ele representa um cursor que percorre o documento de início a fim. Este cursor vai apontar para aquilo que pedirmos: um nó, uma tag de inicialização, um comentário, etc. Este cursor nunca anda para trás.

O nosso leitor do XML começa usando a classe `XMLInputFactory` para carregar os backups:

Excerto 2.1: Carregar os documentos XML.

```
for(int i=0; i<nsnaps; i++){
      XMLInputFactory factory = XMLInputFactory.newInstance();
      factory.setProperty(XMLInputFactory.IS_COALESCING, true);
      XMLStreamReader reader = factory.createXMLStreamReader(new
            FileInputStream(snaps_paths.get(i)));
            ...
}
```

O *parser* StAX tem uma maneira bastante peculiar de trabalhar e que demorou algum tempo até entendermos. Para o StAX saber em que tipo de *node* se encontra é necessário invocar o método `getEventType`. No nosso caso vai ser necessário saber o início, o conteúdo e o fim de uma *tag*, logo iremos precisar dos eventos `START_ELEMENT`, `CHARACTERS` e `END_ELEMENT`.

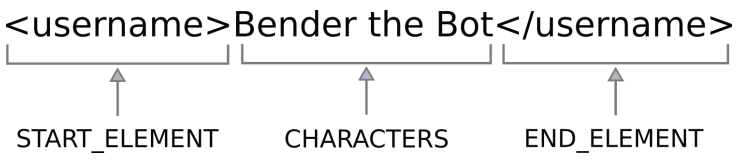

Figura 2.1: *Método* `getEventType`.

Assim sendo torna-se mais fácil trabalhar usando um conjunto de *flags* demonstrado no excerto abaixo:

Excerto 2.2: Como recolher o título do artigo.

```
case XMLStreamConstants.START_ELEMENT:
      String qName = reader.getLocalName();
      if(qName.equalsIgnoreCase("title")){
            bTitle = true;
      } break;

case XMLStreamConstants.CHARACTERS:
      if(bTitle){
            title = reader.getText();
            bTitle = false;
      } break;
```

Para toda a informação que quisermos retirar do XML temos de aplicar o que foi feito para o título do artigo no excerto acima. Então como vamos guardar a informação guardada quando chegamos ao fim de um artigo?

O nosso método para resolver este problema foi criar uma `writeFlag` para sinalizar quando devemos escrever numa estrutura a informação recolhida do XML. Essa `writeFlag` é ativada quando recolhemos a última informação que desejamos do artigo, ou seja, após recolhermos o tamanho e palavras do texto.

Esta `writeFlag` é imediatamente "desligada"quando chegamos ao fim da *tag page* fazendo uso do evento `END_ELEMENT`.

Excerto 2.3: Funcionamento da `writeFlag`.

```
case XMLStreamConstants.START_ELEMENT:
      String qName = reader.getLocalName();
      if(qName.equalsIgnoreCase("text")){
            bTextSize = true;
            bTextWords = true;
            writeFlag = true;
      } break;

case XMLStreamConstants.CHARACTERS:
      if(writeFlag){
            art = new ArticleStructure(title,id,revid,time,name,contid,
                  size,words);
            return art;
      } break;

case XMLStreamConstants.END_ELEMENT:
      String qNames = reader.getLocalName();
      if(qNames.equalsIgnoreCase("text")){
            writeFlag = false;
      } break;
```

2.2 Implementação da Estrutura

Para guardar as informações relativas a um artigo criamos uma classe `ArticleStructure`. Nela vamos guardar toda a informação que necessitamos relativamente a um artigo:

Excerto 2.4: Classe `ArticleStructure`.

```
public class ArticleStructure{
      private String title;
      private long id;
      private long revid;
      private String timestamp;
      private String contributor;
      private long contid;
      private long size;
      private long words;
      ...
```

Tal como foi feito no projeto em *C* criamos também uma classe de contribuidores com menos informação (ID, ID da revisão, Contribuidor, ID do contribuidor) que vai servir para criar uma hashmap orientada pelos ID's dos contribuidores que vai ajudar na resolução de 2 questões.

Neste projeto voltamos a recorrer a uma tabela de hash para guardar as informações. Este método revela-se útil pois é muito mais fácil encontrar o que pretendemos quando temos a *key* da hash onde procurar. Aqui, as tabelas de hash serão orientadas pelos ID's de artigo na hash de artigos e pelos ID's de contribuidor na hash de contribuidores.

Excerto 2.5: Criar tabelas de hash.

```
artHash = new HashMap<Long,ArrayList<ArticleStructure>>();
contHash = new HashMap<Long,ArrayList<ContributorStructure>>();
```

O ficheiro *Load.* é o responsável pela leitura do XML e sua organização nas respetivas estruturas. Neste ficheiro temos uma função principal, a `Load` que carrega os backups e que por cada artigo vai invocar a função `buildArt` que lê o XML e cria as estruturas e a insere-as nas respetivas tabelas de hash através das respetivas funções de inserção (`insertArt` e `insertCont`).

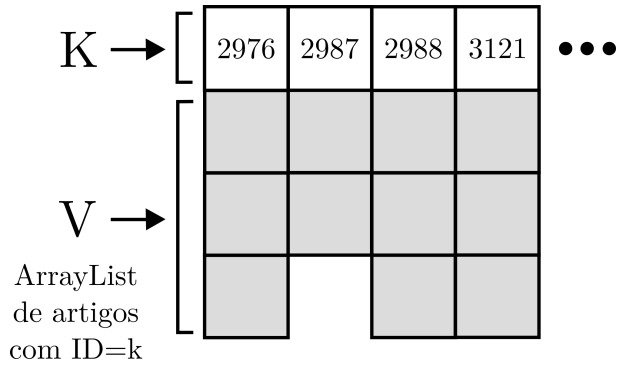

Figura 2.2: *Organização da hash dos artigos. A hash de contribuidores será igual, sendo que o K será o ID do contribuidor*

2.3 Operações à Estrutura

No ficheiro *QueryEngineImpl.java* temos três funções para além das interrogações, a `init`, `load` e a `clean`.

A `init` tem a função de inicializar as duas tabelas de hash por omissão.

A `load` carrega as tabelas de hash com a informação lida dos XML na classe `Load`.

A `clean` invoca o método `clear()` da API de *Hashmap* e limpa as estruturas de hash. Após esta função os mapas estarão vazios.

2.4 Interrogações

1. A função **`all_articles`** foi resolvida enquanto lemos os ficheiros XML. Como temos de saber sempre que um novo artigo começa para criar uma nova estrutura, criamos também uma variável que incrementa sempre que encontra um novo artigo e retornamos essa variável na função all_articles.
2. Para a **`unique_articles`** como a hash tem como *key* os ID's dos artigos e como *value* um *ArrayList* dos artigos desse ID, então o número de ID's únicos é o tamanho desta hash.
3. Na **`all_revisions`**, para cada *key* (ID) vamos criar um *ArrayList* com os ID's de revisão únicos daquele artigo. Quando chegarmos ao fim dos artigos daquele ID vamos adicionar o tamanho da lista criada, que será o números de revisões diferentes, a um contador. E passa para a próxima *key*. No fim o contador deverá ter o número total de revisões diferentes.
4. Para a **`top_10_contributors`** usamos a hash de contribuidores. Nesta hash criamos para cada contribuidor (*key*) um *Set* com os ID's de revisão únicos desse contribuidor e no fim dessa posição conta o tamanho do set e adiciona a um hashmap criado para a função, o ID do contribuidor como *key* e o número de revisões únicas como *value*. Após percorrer todas as posições da hash dos contribuidores iremos ter uma hash preenchida com todos os ID's de contribuidores e respetivo número de revisões. Usamos o pacote *Java Stream* para ordenar pelo nº de revisões e cortar pelo tamanho 10. A *key* da hash vai conter os ID's dos 10 maiores contribuidores, que será (e apenas a *key*) copiada para um *ArrayList*.
5. Na **`contributor_name`** fazemos proveitos das vantagens da hash. Uma vez que nos dão o ID do contribuidor basta ir para a *key* igual a esse ID na hash dos contribuidores e obter o nome desse contribuidor num dos artigos desse contribuidor.
6. A questão **`top_20_largest_articles`** cria um hashmap onde irá guardar na *key* o ID de artigo e no *value* o tamanho desse artigo, caso não exista ou substituindo caso o tamanho seja maior do que o que já existia com aquele ID. Depois de completo o preenchimento do hashmap fazemos o mesmo *stream* aplicado na `top_10_contributors` e iremos ter um *ArrayList* com os ID's ordenados pelos de maior tamanho.
7. Para a **`article_title`**, uma vez que nos é dado o ID do artigo que queremos, apenas temos de ir à *key* exata da hashmap de artigos e devolver o título do artigo mais recente (que irá ser o último elemento do *ArrayList* de artigos).
8. Na função **`top_N_articles_with_more_words`** usamos exatamente o mesmo método que na `top_20_largest_articles`, com a única diferença que o hashmap vai ser populado com o ID de artigo e respetivo número de palavras.
9. Na questão **`titles_with_prefix`** percorremos cada posição da hash e em cada uma verificamos se o título dos artigos tem como prefixo a *String* dada usando para isso a função predefinida em *Java* `startsWith()`. Se esta condição se verificar (ou seja, se a *String* dada for, de facto, um prefixo do título) é adicionado o título a um

ArrayList e verificado se este não está já inserido. No fim o *ArrayList* é ordenado alfabeticamente recorrendo a outra função predifinida, a `sort()`.

10. Para a última questão, a **`article_timestamp`**, usamos o facto de nos ser dado o ID do artigo para ir imediatamente para a posição da hash onde este se encontra e a partir daí procuramos qual artigo coincidia com o ID de revisão e retornamos a *timestamp* dessa revisão.

Capítulo 3

Conclusão

Após concluirmos o trabalho as primeiras impressões que temos é que o tempo de execução é bem mais demorado do que o projeto anterior desenvolvido em *C*. Verificamos que a maioria do tempo gasto é durante o processo de *Load*, mais especificamente nas funções de contagem de palavras e de tamanho do texto.

Por outro lado a implementação das estruturas e a resolução das interrogações é muito mais fácil e rápida, ocupando muito menos linhas.

O pacote *Java Stream* revelou-se bastante útil em algumas questões porque permite fazer em poucas linhas o que em *C* iria implicar a criação de ciclos e outros.

No geral os nossos objetivos foram concluídos, sendo que todas as perguntas foram concluídas e aperfeiçoadas na medida do possível, tendo sempre em vista um código bem organizado e limpo.