Universidade do Minho

Mestrado Integrado em Engenharia Informática

2º Ano, 2º Semestre

Unidade Curricular de Laboratórios de Informática III

Ano Letivo de 2017/2018

2.º Trabalho Prático de LI 3

(Processamento de ficheiro XML Stack Overflow)

Grupo 57

André Gonçalo Castro Peixoto (a82260)

Diogo Emanuel da Silva Nogueira (a78957)

Diogo Francisco Araújo Leite da Costa (a78485)

Luís Filipe da Costa Cunha (a83099)

Índice

1. Introdução

Este projeto foi enunciado pelos docentes da Unidade Curricular Laboratórios de Informática 3 e o seu principal objetivo foca-se no desenvolvimento de um sistema capaz de processar ficheiros XML que armazenam as várias informações presentes no *Stack Overflow*.

Para esta segunda fase teremos um processamento dos ficheiros XML baseado na linguagem e tecnologia JAVA sendo que assim existe uma ênfase no modelo da Programação Orientada aos Objetos que foi pensado desde raiz com uma modularidade, opacidade e encapsulamento do projeto e das suas classes. Coincidentemente a este objetivo principal, existe também a ideia de se consolidar todos os conhecimentos adquiridos em Unidades Curriculares anteriores como a Programação Orientada a Objetos e Algoritmos e Complexidade.

Assim, aliando uma programação mais complexa e em larga escala, através do processamento duma grande escala de dados, pretende-se criar um programa que faça jus ao pretendido e que vá ao encontro com todo o universo da Modularidade e Encapsulamento de Dados que esteve sempre presente na filosofia da linguagem JAVA.

2. Estruturação das Classes

Para o projeto de Java foi utilizado uma estrutura de *packages* recomendado tanto pelos professores como pela própria ferramenta de compilação automática *Maven*. Desta forma, através dum *pom.xml*, podemos colocar todas as dependências que o trabalho necessitou, tal como o *Log4J*, entre outros. A estrutura manteve-se numa pasta *src* que contém os *packages* *common*, *engine*, *estruturas* e *li3*. Cada uma destas pastas tem classes que foram necessárias para responder a todas as interrogações impostas pelo enunciado geral. Para efetuar o *parse* dos XMLs necessários para responder às *queries* foi utilizado a API Java STAX de forma a ser o mais nativo e conciso possível. A escolha desta API foi dado que toda a sua estrutura sendo mais recente e usando iteradores torna-se mais eficiente. O nosso trabalho teve um desempenho melhorado à escala de quase um minuto de *parse* para apenas segundos.

2.1. Estrutura *hashPosts*

Esta estrutura foi criada com o intuito de armazenar todos os *posts* contidos no ficheiro Posts.xml. Com o suporte das coleções nativas do Java utilizamos um *HashMap* no qual temos o *long id* como a chave e o valor correspondente uma instância da classe Posts que contém como variáveis de instância os mais diversos argumentos XML que foram efetuados no *parse*.

Assim, a classe *Posts* funciona como um esqueleto fundamental entre o *parse* até guardar as informações na coleção Java. Esse esqueleto terá as informações/atributos que serão necessários para as *queries* pedidas, como o *id*, *score*, *ownerUserId, tag,* etc. Nesse *package* também teremos como classes outras sub-coleções que nos facilitaram o trabalho de certas interrogações, tal como árvores dos *posts* ordenadas consoante diversos comparadores de forma a responder de forma mais eficiente às mais diversas *queries*.

Dessa forma, em vez de existir um *parse* sempre que seja necessário para uma *query*, agrega-se e filtra-se a informação relativa aos *posts* logo de início para otimizar depois o tempo da resposta à *query* pedida.

2.2. Estrutura *hashUsers*

A outra estrutura utilizada em peso foi a *hashUsers* que focar-se-á no armazenamento do *parse* das informações do XML dos utilizadores. Foi utilizada a mesma técnica de armazenamento com a ajuda das coleções da API Java que encapsulam, tratam e otimizam o armazenamento dos dados de forma eficiente e autónoma. Uma classe auxiliar foi criada, chamada *User* que receberá os atributos XML já com o *parse* efetuado, tais como o *id, displayName, aboutMe, reputation, etc.*

Tal como na estrutura anteriormente falada, estes argumentos foram pensados para responder automaticamente às interrogações pedidas, dado que a informação estaria logo disponível a partir do *parse* inicial ao invés de estar a correr apenas quando a interrogação fosse pedida.

Toda esta instância de *User* será colocada como "valor" na *hashUsers* e como chave o *id* equivalente ao utilizador equivalente. Desta forma conseguimos um desempenho excelente dado que ao buscar a informação dum certo *id* de utilizador temos logo a informação otimizada de forma imediata dado que não é necessário percorrer um *HashMap*, porque a procura da mesma baseia-se num índice de *hashes* e assim torna todas as interrogações mais eficazes e céleres e disponíveis quase instantaneamente.

2.3. Estrutura *hashTags*

Como uma última estrutura principal temos a *HashTags* que suportará o último *parse* do ficheiro das Tags em XML. O procedimento foi semelhante sendo que foi criada uma classe que albergou todos os atributos das tags, tais como *id, tagname, count*. Essa classe acaba por ser o valor, ou seja, a sua instância fica ligada com a chave, que é o id da tag e assim temos de forma eficiente e nativa todas as ferramentas da API Java da coleção *HashMap*.

2.4. Outras Estruturas

Para responder de forma mais eficiente a algumas *queries* foram também criadas duas estruturas auxiliares chamadas *TreePosts* e *TreeUserNPosts*. A primeira é usada para criar uma instância de uma árvore ordenada por datas tanto de perguntas como respostas, para responder de forma mais eficiente a *queries* que usem em peso a informação de datas e limites das mesmas. A segunda é usada para criar uma árvore ordenada de forma decrescente do nº de posts e o utilizador associado. Dessa forma temos logo o top N de utilizadores mais ativos (ou seja, com mais *posts*) do Stack Overflow.

3. Interrogações

3.1. Primeira Interrogação

Nesta primeira interrogação começa-se por obter toda a informação associada ao ID do *Post* que se pretende procurar, por recorrência ao método *obtemPost* previamente declarado na classe *HashPosts*. **Tendo-se toda a informação (*value*) do *Post* pretendido surgem duas opções:**

- Caso se verifique que o *Post* se trata de uma pergunta, será necessário obter o *ownerUserID* que identifica o ID do utilizador desse Post. Dessa maneira, usa-se o método *obtemUser* previamente declarado na classe *HashPosts*. Dessa forma obtém-se a informação do utilizador completa. Após isso, facilmente se consegue também o título do *Post* e o nome do *User* das instâncias já buscadas às coleções previamente.
- Caso o *Post* seja uma resposta, é necessário pesquisar a pergunta a que corresponde, através do *parentID* que identifica o ID da pergunta a que diz respeito na coleção *HashPosts*. Com isto, devolve-se o título e o nome de utilizador do criador dessa mesma pergunta, de forma idêntica ao algoritmo supracitado.

3.2. Segunda Interrogação

Na segunda interrogação faz-se uso da classe *TreeUserNPosts* que se encontra ordenada pelo número total de *posts* correspondentes aos vários utilizadores existentes. Nesta árvore a *key* corresponde ao nº de posts o valor possui uma lista de *Users* cujo nº de *posts* é igual a essa mesma *key*.

O que acontece é que se recorre a um ciclo *while* que existe enquanto o número N de utilizadores que se pretende obter já estiver finalizado. Dentro desse ciclo colocamos as instâncias dos *User* correspondentes numa *List<Long>* que será o retorno da interrogação.

3.3. Terceira Interrogação

Nesta terceira interrogação, é imperativo recorrer a um método usado aquando a criação da classe *TreePosts*, que é *obtemNumero(data,data)*. Dado que no momento do *parse* foi colocado uma árvore de perguntas e outra de respostas ambas ordenadas por datas, dessa forma, ao dar os limites de inicio e fim, fornecidos pela *query*. Este método utiliza uma *stream* que vai retirando os valores que estejam antes, após e nessas datas dadas como argumento, sendo que depois soma as respostas e perguntas desse intervalo pedido e retorna esse número. Utilizou-se aqui *Java Streams* para melhorar a interrogação sendo que a mesma está nos nossos testes com 0ms de tempo corrido.

3.4. Quarta Interrogação

Esta quarta interrogação faz uso do método também introduzido na classe *TreePosts*. Dessa forma, com o *obtemTagsIntervalo(tag, inicio, fim)* nós criamos um *ArrayList<Long>* que irá ficar com todas os ids das perguntas que usaram a tag durante o intervalo passado como argumento. Usando um ciclo *for* semelhante ao utilizado na interrogação anterior, caso a pergunta esteja entre as datas indicadas inclusive, verifica-se depois se essa pergunta contém a *tag* pedida. Se isso acontecer, coloca-se o *ID* da pergunta no ArrayList que depois será devolvido como a resposta da quarta interrogação.

3.5. Quinta Interrogação

A quinta interrogação recebe como o argumento o *id* dum utilizador e pede a biografia e os *IDs* dos últimos N *posts*.

Para a biografia apenas foi necessário um *get* da *HashUsers* já definida e depois um *getBio()*, desse mesmo utilizador.

Para os N últimos *posts* foi-se à *TreePosts* que esse utilizador tinha e de forma reversa (ou seja, os mais recentes posts), limitou-se para N posts e colecionou-se os IDs desses mesmos posts para uma Lista para dar de resposta à quinta interrogação.

3.6. Sexta Interrogação

Para o desenvolvimento desta query, o grupo manteve desde início a ideia de que o número de votos equivale ao *score* de uma determinada pergunta/resposta. Esta perceção foi fundamental para que se concluísse que era desnecessário fazer o parse de outros *xml* que não os inicialmente idealizados perante as *queries* pedidas.

Dessa forma, através dum ciclo *for* para intercalar os limites temporais dos *posts*, adicionamos todos os *posts* desse intervalo numa lista e depois utilizando um comparador específico para os scores, tornamos essa lista ordenada por scores dos *posts*. Após isso, apenas basta obter os *IDs* dos N *posts* pedidos na interrogação e fornecer esse *ArrayList<Long>*.

3.7. Sétima Interrogação

A sétima interrogação utiliza um algoritmo pensado de forma a responder de forma correta à questão. Através da criação dum comparador inverso e duma *TreeMap<Integer,List<Long>>*, coloca-se neste todas as perguntas que estejam dentro do intervalo de datas definido pela interrogação. Depois nessas perguntas analisamos o tamanho das respostas que essas perguntas têm e adicionamos a essa TreeMap criada para o propósito que irá ter como *key* o tamanho e o valor será uma lista de IDs de perguntas que tenham esse tamanho de respostas associadas ao mesmo. Com a TreeMap feita, apenas se fez um ciclo *for* para apenas colocar os N IDs como pedido pela interrogação.

3.8. Oitava Interrogação

Esta sétima interrogação faz uso do método também introduzido na classe *TreePosts*. Dessa forma, com o *obtemIDsPalavra(N,palavra)* nós criamos um *ArrayList<Long>* que irá ficar com todas os ids das perguntas que tenham a palavra pedida no título do mesmo. Usando um ciclo *for*, verifica-se o título do post contém a palavra pedida. Caso contenha, adiciona-se esse *post* a um *ArrayList<Post>* temporário. Após todos os posts serem analisados, faz-se reverse a este *ArrayList* temporário para ficar de forma cronologicamente inversa. Depois basta adicionar os IDs deste post para um *ArrayList<Long>* que será a resposta da *query*.

3.9. Nona Interrogação

A nona interrogação envolve dois utilizadores, portanto a primeira parte foi instanciar e ir buscar os utilizadores com os ids fornecidos pela interrogação. Após ter os Utilizadores, e dado que cada um tem uma árvore de posts seus, manipulou-se com um algoritmo ambas as árvores de modo a responder à pergunta.

Através de dois ciclos *for* conjuntos, percorremos todos os *posts* do utilizador 2 e comparamos para todos os *posts* do utilizador 1 e verificamos se são uma pergunta e resposta, se são resposta e pergunta, ou ainda, se são ambas resposta de uma pergunta. Dessa forma, certificamos as três maneiras que os dois utilizadores pudessem ter interagido entre eles. Após isso limitou-se pelo número dado esses ids de posts que houve interação.

3.10. Décima Interrogação

Dado o *ID* de uma pergunta, o objetivo passa por obter a melhor resposta através do cálculo de uma fórmula criada para o efeito. Para se poder chegar a esta conclusão é preciso vários passos de programação. A primeira coisa que fizemos foi enviar para o método tanto o *HashPosts* como o *HashUsers*. Dessa forma, tendo o *post* concreto vindo da *HashPosts*, fez-se um *getRespostas()*, dado que cada *Post* tinha previamente uma variável *TreePosts* que era todas as respostas a esse mesmo post. Assim percorre-se todas as respostas desse post e para cada uma faz-se o cálculo da equação. Quando o cálculo for máximo devolve-se esse id da resposta em causa, dado que é a melhor resposta para esse post pedido pela interrogação.

3.11. Décima Primeira Interrogação

A primeira parte do algoritmo para a última interrogação foi buscar de forma inversa à *HashUsers* os N Users com melhor reputação. Depois a cada um desses Users, foi-se à sua árvore de posts e coleciona-se todas as tags e a sua contagem durante o tempo dado como argumento e esse é colocado num *HashMap<String,Integer>* sendo que a *key* é a tag e o Integer é a contagem dessa tag nos posts todos dos users com melhor reputação. Após todas as tags estarem contabilizadas, passou-se para uma lista de forma a reverter a ordem, comparando os valores, ou seja, colocando de forma decrescente. Por fim, e de forma a responder à interrogação foi procurado todos os IDs dessas tags e colocado numa lista para responder.

4. Otimizações da velocidade

Neste projeto Java tomou-se como iniciativa desde inicio que a otimização, encapsulamento seria feito desde a raiz do código. Com um LOAD apenas de 17 segundos e com todas as interrogações feitas num espaço de milissegundos, foi um fruto de uma programação que preferiu a utilização da API Java que está testada pelos milhões de programadores que a usam. Desde as Coleções, como ArrayList, HashMap, TreeMap, mas também a API Stream Java para fazer certas iterações por forma de programação funcional e assim poupar imenso tempo na análise dos dados para responder às perguntas. Mesmo a escolha do Java StAX, que é uma API mais atual e utiliza no seu núcleo *iteradores* e *streams*, tornou-se logo a nossa escolha para fazer o *parse* do XML em vez do SAX. Em comparação o SAX fazia um *load* em quase 2 minutos enquanto o StAX demora cerca de 17 segundos na mesma máquina.

Apesar que o Java é uma linguagem de programação mais abstracta e de maior nível que C, tentamos ao máximo obter o maior desempenho e otimização nesta segunda fase do projeto sendo que foi o nosso foco desde o princípio do mesmo.