universidade de aveiro

DETI | ua

1º Relatório
Recuperação de Informação 2018-2019 1ºS

21-10-2018

Professor: Sérgio Matos

Aluno: André Cardoso, 65069, (marquescardoso@ua.pt)
Ivo Angélico, 41351, (ivoangelico@ua.pt)

Índice:

Apresentação do 1º Trabalho Prático

Este trabalho consistiu no desenvolvimento de um indexer de documentos constituído por um corpus reader, tokenizer e indexer. O Corpus a ser indexado é parte do Amazon Customer Reviews Dataset disponível aqui. Cada review será considerada um documento devendo ser indexados os campos *product_title*, *review_headline* e *review_body*.

O diagrama de classes da aplicação desenvolvida encontra-se no Anexo I.

Aplicação

A aplicação inicia criando os objetos Memory, Timer, Corpus Reader, DocCollection e Indexer. O objecto Memory é inicializado com um argumento que representa a quantidade de memória disponível para a JVM e outro 80% que representa a percentagem de memória ocupada a partir da qual é necessário escrever informação para o disco de forma a poder libertar memória. O objeto Timer é utilizado para coletar informações sobre o desempenho temporal da aplicação. Os restantes objectos serão analisados posteriormente neste relatório.

O corpus a ser lido encontra-se em disco num caminho que é conhecido pela aplicação. A aplicação corre um ciclo em que lê cada um dos documentos até não haver mais documentos disponíveis. A operação de leitura é executada pelo Corpus Reader.

Corpus Reader

O Corpus Reader utiliza um BufferedReader para ler o ficheiro, lendo um documento de cada vez. O Corpus Reader retorna um objecto Doc que contém um *document identifier* que é um inteiro único atribuído sequencialmente a cada objecto. O objecto Doc tem ainda o conteúdo do documento, já expurgado das secções irrelevantes. Se neste trabalho fosse necessário executar queries o Doc deveria ter guardadas informações adicionais que seriam necessárias para a resposta às queries tais como customer_id, review_id e product_id.

Após se ler um documento este será processado por um tokenizer. Dependendo dos argumentos de entrada o tokenizer será um dos dois seguintes.

Simple Tokenizer

Este tokenizer separa o documento em cada espaço, transformar os caracteres alfabéticos para lowercase, remover os caracteres não alfabéticos e remover os tokens com menos que 3 caracteres.

Improved Tokenizer

Este tokenizer separa o documento em cada espaço e transformar os caracteres alfabéticos para lower case.
Após isso verifica uma série de regras:

- Verifica se o token tem um arroba @. Se tiver verifica se tem pelo menos um caracter à esquerda, pelo menos 3 caracteres à direita e se um desses 3 caracteres. Se essa condição for preenchida assume-se que este termo é um email e por isso deve ser mantido.

- Verifica se o termo tem uma apóstrofe ‘ e se tiver divide o termo em duas partes. Se a segunda parte tiver até 2 caracteres assume-se que se trata de uma contracção e por isso é removida ficando apenas a primeira parte do termo. Os restantes caracteres não alfabéticos são removidos. Se não, a apóstrofe é removida bem como os restantes caracteres não alfabéticos.
- Verifica se o termo tem um hífen - e se tiver divide o termo em duas partes. Se a primeira parte tiver até 2 caracteres assume-se que se trata de um prefixo e por isso a primeira parte é removida. Os caracteres não alfabéticos são removidos.

Após estas regras é verificado se o token faz parte de uma lista de stopwords. Esta lista foi constituída a partir da lista presente aqui, à qual foram aplicadas as regras anteriores de forma aos termos que tinham sido normalizados serem devidamente identificados.

Finalmente os tokens passam pelo Porter Stemmer e são depois adicionados a uma lista de termos.

Indexer

O indexer foi implementado segundo a estratégia *Blocked Sort-Based Indexing* pois o dicionário é relativamente pequeno e cabe em memória mas as *posting lists* ocupam demasiado espaço sendo necessário escrever em disco. Nesta secção iremos analisar a construção dos sub-segmentos do indexer que estarão ordenados e na secção iremos analisar a junção dos segmentos num index final.

O indexer recebe um docId e uma lista de termos. Os termos são corridos e é calculada a sua frequência. A seguir é verificado se o termo já existe na hashmap do indexer onde estão os termos e posting list. Se não existir é criada uma *posting list* nova e adicionada ao hashmap. Se existir, a *posting list* existente é alterada para acrescentar este documento e respectiva frequência.

Após isto é verificado quanta memória está a ser utilizada. Se o valor for superior ao definido como máximo aceitável o hashmap é convertido num **treeMap** de forma aos termos ficarem ordenados e é salvo para o disco com recurso a um newBufferedWriter. Foi estudada a possibilidade de ser criado de origem um treeMap em vez do hashmap mas foi verificado que nesse caso a operação de indexação é aproximadamente 30% mais lenta do que quando era criado um hashmap e depois convertido para treeMap. Após se escrever em disco, o hashmap é limpo de forma a se começar uma nova lista. O garbage collector é chamado para se limpar a memória.

Indexer merging

O processo anterior é executado continuamente até não haver mais documentos para indexar. Aí é escrito para memória o último hashmap. Temos agora em disco uma série de documentos com segmentos do indexer final cujos membros necessitam de ser reunidos em ordem alfabética. O indexer final será composto com base num LinkedHashMap.

O primeiro termo de cada segmento é lido. Após isso entra-se num ciclo até todos os termos de todos os segmentos terem sido indexados. Para decidir qual o próximo termo a ser inserido no dicionário, são comparados um elemento de cada segmento até ser escolhido qual o de maior ordem lexicográfica. Esse elemento é adicionado ao LinkedHashMap ou a posting list é fundida com a já existente se o termo já tiver sido adicionado. É lido um novo elemento do segmento (ou segmentos caso o mesmo estivesse disponível em vários segmentos) de onde o termo foi lido (caso ainda haja termos para serem indexados nesse segmento). É realizado o mesmo processo de salvar em disco o LinkedHashMap quando a memória disponível é baixa. Em memória é mantido um dicionário com o termo, um id do documento com o segmento do index final onde está guardado esse termo e a linha em que ele está.

Resultados

O documento inicial tem 3.9GB. É possível verificar na Figura 1 que quando a utilização de memória ultrapassa os 1.2GB (80% de 1.5GB) há uma queda da utilização da mesma e na Figura 2 é possível verificar que há actividade do garbage collector nesses instantes. Isto é o esperado pois, após se ler cada documento é verificado se já atingiu esse nível de memória e se for o caso a informação é passada para disco e a memória libertada.

Figura 1 - Utilização da memória com o simple tokenizer

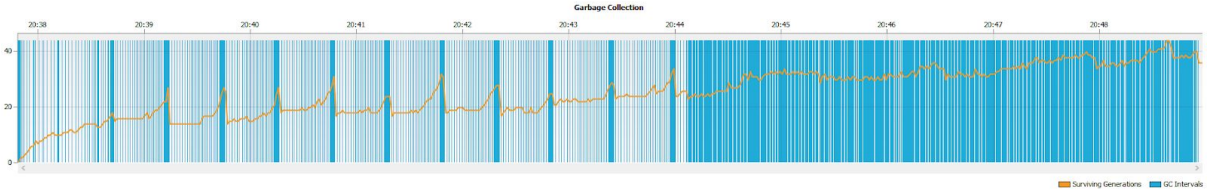

Figura 2 - Utilização do garbage collector com o simple tokenizer

Foram compilados resultados utilizando cada um dos tokenizers.

Simple tokenizer

Com o simple tokenizer obteve-se um indexer parcial de 3.04GB, com 9002021 tokens e um indexer ordenado com 3.02GB. O dicionário tem um tamanho de 40.5MB. No final obtiveram-se <u>1096116 termos (2.3a).</u> A geração dos segmentos demorou 6:18min e o merge 4:48min num total de 11:06min.

Os primeiros 10 termos que aparecem apenas num documento são (2.3b):

1. aaaaaaaaaaaaaa=[7498274:1]
2. aaaaaaaaaaaaaaaaa=[1168337:1]
3. aaaaaaaaaaaaaaaaaaa=[2374108:1]
4. aaaaaaaaaaaaaaaaaaaaaa=[6408078:1]
5. aaaaaaaaaaaaaaaaaaaaaaa=[4258952:1]
6. aaaaaaaaaaaaaaaaaaaaaaaaaaa=[7370868:1]
7. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa=[5314407:1]
8. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa=[5022344:1]
9. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa=[5114109:1]
10. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaa=[4663841:1]

O documentos com maior número de frequência são (2.3c):

1. 4065373=case
2. 3975339=with
3. 3755690=this
4. 3086380=phone
5. 3004789=iphone
6. 2557475=great
7. 2222835=samsung
8. 2212414=that
9. 2094161=galaxy
10. 2022026=have

Improved tokenizer

Para o improved tokenizer, verificamos que a memória ocupada pela aplicação era maior (por causa do stemmer). Para evitar que fossem criados um número excessivo de documentos aumentou-se a memória para 3GB, mantendo-se os 80% de percentagem de memória utilizada.

Com o improved tokenizer obteve-se um indexer parcial de 2.34GB, com 9002021 tokens em 14:24 min. O merge dos indexers parece estar funcional mas último segmento do indexer não está a ser escrito para o disco por motivos que ainda desconhecemos. Por esse motivo não conseguimos apresentar resultados para este indexer.

Os primeiros 10 termos que aparecem apenas num documento são (2.3b):

1. !#@$!#@.=[5450222:1]
2. #joeythepcguy@facebook.=[5247997:1]
3. #mckeesavage@yahoo.com=[1126528:1]
4. $!*@house...=[2215229:1]
5. $%@**#@...=[8476457:1]
6. $.5:5,98)?@&&!8(8(6.5:/(()8?8((808?=[6653178:1]
7. $8.00..sassyswassy@aol.com...let=[7806491:1]
8. %*#$!@#$(*.=[1938463:1]
9. "&%#%**$%#@$^gjhvxr^*(%#q@r%%#@d".=[741592:1]
10. "@feelbug.yoon"=[4052887:1]

Fica claro que a regra que define se um termo é um email tem de ser melhorada.

Como executar a aplicação

Antes de executar a aplicação devem ser criados na raíz do projecto o directório *testing* e dentro dele os diretórios *tmp* e *final.* Na pasta testing deve estar presente o ficheiro que vai ser indexado com o nome amazon_reviews_us_Wireless_v1_00.tsv. (Algo a melhorar no próximo trabalho.)

Executar o seguinte comando para a instalação e criação do ficheiro .jar "mvn package". O programa tem de correr com um argumento de entrada, -st (simple token) ou -it (improved token e stemmer). É possível correr o programa num editor com ambos os argumentos de entrada.

No entanto ao executar o ficheiro .jar só é possível com o argumento -st. O argumento -it não é possível devido a uma configuração, que pretendemos investigar mais a fundo, em que as bibliotecas do package Stemmer não estão a ser incluídas na geração no ficheiro .jar. Mas é possível correr o ficheiro .jar com o seguinte comando:
"java -jar [nome do jar] -st".

Conclusões

Conseguiu-se executar com sucesso todas as tarefas que estavam propostas. Analisando os tamanhos totais do indexer e do dicionário verifica-se que as opções de estrutura do programa no que concerne a gestão de memória foram as adequadas. A utilização da hashmap é aceitável nesta situação pois os documentos não irão ser actualizados.

Caso se pretenda melhorar o indexer no futuro poderão realizar-se algumas acções:

- Incluir novas stopwords com os termos mais frequentes (exemplo: wifi);
- Usar programação concorrente para potenciar a utilização do CPU;
- Criar buffers dos segmentos do indexer na memória para evitar ler esses ficheiros enquanto não se indexarem todos os termos que estão na memória.
- Melhorar o código: criar directorias necessárias caso não existam e melhorar a recepção de argumentos.
- Melhorar a execução do improved tokenizer.
- Melhorar o filtro dos emails

Para além disso foi feito um profiling das funções de forma a perceber-se onde vale a pena investir em otimização no futuro (ver Figura 3).

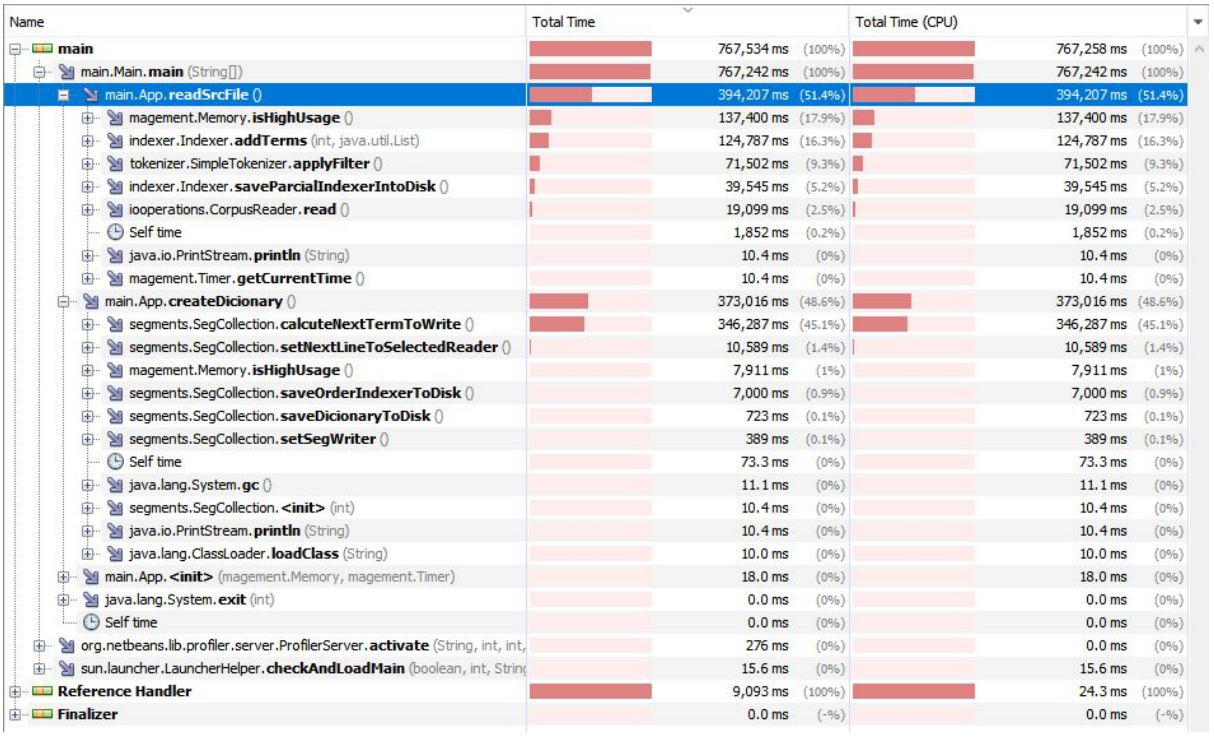

Figura 3 - Profiling das funções

Anexo I - Diagrama de Classes