TRABALHO PRÁTICO 1:
Arquivos Invertidos

Felipe Moraes Gomes

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

felipemoraes@dcc.ufmg.br

Resumo. *Este trabalho tem como objetivo implementar um indexador de páginas web produzindo um arquivo de índice invertido e um processador de consultas booleanas.*

1. INTRODUÇÃO

Este trabalho teve como objetivo projetar e implementar um sistema de programas para recuperar de forma eficiente informação em grandes arquivos armazenados em memória secundária, utilizando um tipo de índice conhecido como arquivo invertido.

Para exemplificar o que é um arquivo invertido considere a definição a seguir. Considere um conjunto de documentos. A cada documento é atribuído um conjunto de palavras-chave ou atributos. Um arquivo invertido é constituído de uma lista ordenada de palavras-chave, onde cada palavra-chave tem uma lista de apontadores para os documentos que contêm aquela palavra-chave [Baeza-Yates and Ribeiro-Neto 2011].

Também, neste trabalho um processador de consultas booleano (and ou or) foi implementado para fins de testes da recuperação de documentos que satisfação a consulta que contêm uma ou mais palavras. Este buscador é um esboço para os próximos trabalhos.

O restante deste relatório é organizado da seguinte forma. A Seção 2 descreve a solução proposta para este trabalho. A Seção 3 trata de detalhes específicos da implementação do trabalho. A Seção 4 contém a avaliação experimental do indexador de arquivos invertidos e da busca booleana. A Seção 5 conclui o trabalho.

2. SOLUÇÃO PROPOSTA

A solução proposta deste trabalho foi baseada no modelo de programação MapReduce. A primeira fase Map lida com separar ocorrências de termos de forma ordenada em vários arquivos. A segunda fase Reduce lida com agrupar todos arquivos gerados na fase anterior em um único arquivo. Desta forma, a solução proposta possui duas Classes principais Mapper e Reducer descritas na seção 2.1 e 2.1, respectivamente. Na seção 2.3 descrevemos de forma global a solução proposta utilizando essas duas classes.

2.1. Mapper

A classe Mapper possui um método principal que é chamado $process\_page$ e um outro método chamado $process\_frequencies$. Como atributos importantes, ela guarda um vocabulário, um buffer limitado pelo tamanho da memória principal a ser utilizada e um vetor da classe auxiliar File. A complexidade de espaço desses atributos é $O(n + m + k)$ onde n é o tamanho do vocabulário, m o tamanho da memória principal e k o número de arquivos gerados.

O método $process\_frequencies$ recebe um texto como entrada removendo os acentos e convertendo letras maísculas para mínusculas. Essa primeira fase é feita em $O(n)$ onde n é o tamanho do texto pois é necessário processar cada carácter do texto. Depois disso o método tokeniza o texto em tokens e para cada token do texto ele adiciona sua posição do texto em uma hash. Além disso, ele não inclui o termo se é um termo muito frequente (stopword). Essa segunda fase é feita em $O(n)$ para o tokenizer e $O(m*k)$ para a remoção das stopwords onde n é o tamanho do texto, m número de tokens e k número de stopwords. A complexidade final é de $O(n) + O(m * k)$.

Já o método $process\_page$ lida com chamar o $process\_frequencies$ e escrever cada termo na hash resultante em um buffer. A complexidade deste método é $O(n) + O(m)$ para o tamanho do texto n e número de termos m.

Para lidar com a escrita dos termos no buffer e realizar a separação destes termos em vários arquivos, papel do Mapper, é chamado um método chamado $flush$ para cada inserção de um termo para escrita. O $flush$ apenas verifica se o tamanho do buffer foi ultrapassado e escreve o buffer todo em um arquivo binário ao chamar o método $exec$ se tiver sido ultrapassado. Este último método tem complexidade $O(nlogn)$ possui realiza uma ordenação antes de mandar escrever em disco onde n é o tamanho do buffer.

As tarefas dessa classe terminam com a execução de todas as páginas e as ocorrências de cada termo terem sido salvas em vários arquivos. Todos os atributos alocados dinamicamente são esvaziados após o termino do programa. Um método $dump$ é utilizado para escrever o vocabulário em um arquivo texto separado.

2.2. Reducer

A classe Reducer possui dois métodos principais $merge$ e $reduce$. A classe $merge$ realiza uma ordenação externa de múltiplos caminhos. Primeiro, a classe $merge$ chama para cada 1024 arquivos o método $kmerge$, se o número de arquivos for menor que 1024 ele chamará esse método apenas uma vez. Esse número é o número de arquivos padrão que um programa em um sistema operacional pode manter em aberto.

O método $kmerge$ realiza uma ordenação de múltiplos caminhos utilizando um heap. Um heap é uma estrutura que mantém em sua primeira posição sempre o menor elemento. Nessa estrutura precisamos fazer $O(nlog(m))$ operações de inserção e remoção no heap, onde n é o número de termos de todos arquivos e m o tamanho do heap que é o número de arquivos gerados na fase de Map. A complexidade final deste método é de $O(nlog(m))$.

Já o método $reduce$ tem a tarefa de agregar o arquivo ordernado produzido pelo método de $merge$. Esse método tem complexidade linear em relação ao número de ocorrências do arquivo ordenado. O método termina produzindo o arquivo de índice invertido.

2.3. Algoritmos principais

Essa seção descreve os dois algoritmos principais baseados nas classes descritas anteriormente. A seção 2.3.1 descreve o indexador e a seção 2.3.2 descreve o buscador booleano.

2.3.1. Indexer

Definido as duas classes anteriormente, abaixo segue um pseudocódigo que representa o fluxo de execução do algoritmo principal. O laço interno possui complexidade $O(nlog(m))$ que é a complexidade do método $kmerge$ da classe Mapper, o mais caro deste laço, com exceção do parser de páginas HTML, que não sabemos a complexidade envolvida. Por fim a complexidade final deste algoritmo é de $O(K * nlog(m))$ onde n é o número de ocorrência de termos, m o tamanho do buffer e K o número de páginas HTML a serem processadas.

Algorithm 1 Algoritmo de texto

Input: páginas: Arquivo comprimido com páginas web em HTML
Output: index: Arquivo de índice invertido
while Ainda existe página em páginas **do**
1. Realiza o Parse da página
2. Chama $Mapper.process\_page$
3. Chama $Reducer.merge$
4. Chama $Reducer.reduce$
5. Escreve vocabulário em disco

end while

2.3.2. Searcher

Foi implementado um buscador booleano que utiliza de uma estrutura de dados implementado com a classe Vocabulary. Essa classe carrega um vocabulário de um arquivo texto gerado pelo indexador com seus identificadores. Para deixar a busca mais eficiente, nesse mesmo arquivo foi adicionado o seek dos termos no vocabulário. A busca tem complexidade $O(1)$ no arquivo de índice e tem complexidade no melhor caso $O(1)$ para interseção e união das listas de documentos de consulta com apenas um termo. No caso médio a complexidade é de $\sum_{i=0}^{n-1} O(2(list_i + list_{i+1}))$ onde n é a quantidade de termos da consulta e $list_i$ é o tamanho da lista do termo i. Para melhorar a busca no caso do conector lógico and começamos a interseção com a menor lista. Já para o conector lógico or nenhuma melhoria poderia ser realizada pois a união das listas independe da ordem.

3. IMPLEMENTAÇÃO

Nesta seção descrevemos a estrutura de organização deste trabalho seção 3.1, seção 3.2 as etapas de compilação e seção 3.3 as etapas de execução.

3.1. Estrutura

Abaixo, segue a estrutura de organização deste trabalho. Temos um diretório para as bibliotecas, a biblioteca cedida para ler as páginas comprimidas ($riCode$), uma biblioteca de funções e classes do indexador e uma biblioteca de funções e classes comuns. Os arquivos fontes das funções principais que geram os binários de execução encontram-se no diretório src. Também um diretório $util$ é fornecido com um diretório para listas de stopwords.

```
search-engine
├── lib
│   ├── common
│   ├── index
│   └── riCode
├── src
│   ├── indexer
│   └── searcher
├── util
│   └── stopwords
└── test
```

3.1.1. Lib

Este diretório contem um diretório com funções e classes que são comuns ao indexador, buscador e um diretório com om funções e classes do indexador como a Mapper e Reducer.

3.1.2. Src

Este é o diretório onde os arquivos de algoritmos principais estão localizados como o do indexador e do buscador, com dois diretórios respectivamente. Cada um dos diretórios contem um arquivo main.cc.

3.1.3. Util

Este diretório contem utilidades para serem usadas no indexador como por exemplo as listas de stopwords (palavras mais frequentes de um certo idioma). Neste diretório contem um diretório com listas de stopwords em português, inglês e espanhol.

3.2. Compilação

Este trabalho foi desenvolvido com Cmake, um sistema multiplataforma para realizar geração automatizada. É comparável com o programa Unix Make no qual o processo de geração é, ao final, controlado pelos arquivos de configuração, no caso do CMake chamados de arquivos CMakeLists.txt.

Uma única biblioteca externa foi utilizada, a do parser de páginas HTML. O parser escolhido foi o Gumbo Parser [1]. Para prosseguir com a compilação é necessário ter essa biblioteca instalada.

Para realizar a compilação em um ambiente Unix, os seguintes passos são necessários:

```
# cmake -G "Unix Makefiles"
# make
```

[1] https://github.com/google/gumbo-parser

3.3. Execução

Após a compilação um diretório bin é criado com dois arquivos executáveis: indexer e searcher. A seguir segue uma lista de parâmetros do indexer:

- **–directory ou -d**: Indica o diretório que está localizada a coleção de documentos a ser indexada.
- **–runSize ou -r**: Quantidade máxima do buffer (triplas ¡t, d, fd,t¿) que devem ser armazenadas em memória principal. Se não fornecido assume o valor padrão de 5.
- **–output ou -o**: Diretório que índice final deve ser armazenado. Se não fornecido assume como padrão a pasta de execução do programa.
- **–fileName ou -f**: Indica o nome do arquivo de índice da coleção de documentos. Se não fornecido, assume o valor padrão "index.txt".
- **–numDocs ou -n**: Quantidade de arquivos a ser indexados (se não fornecidoassume o valor padrão 999.999.999)
- **–stopwords ou -s**: Diretório que está localizado as listas de stopwords.

Abaixo os paramêtros do searcher:

- **–directory ou -d** Indica o diretório que está armazenado o índice gerado pelo indexer.

O buscador recebe da entrada padrão um inteiro n representando o número de consultas seguido do tipo de conector lógico a ser utilizado, and ou or. Em seguida para cada consulta um inteiro m é passado representando o tamanho da consulta e logo depois m termos da consulta. No diretório test encontra-se dois arquivos de testes de exemplo com consultas.

4. AVALIAÇÃO EXPERIMENTAL

Nesta seção avaliamos a solução proposta em termos do tempo de execução e também em tamanho dos arquivos de índice e vocabulário. Os experimentos foram executados em um computador com sistema operacional Ubuntu 14.04 LTS, processador Quad-Core AMD Opteron(tm) 2376 e 32GB de memória principal.

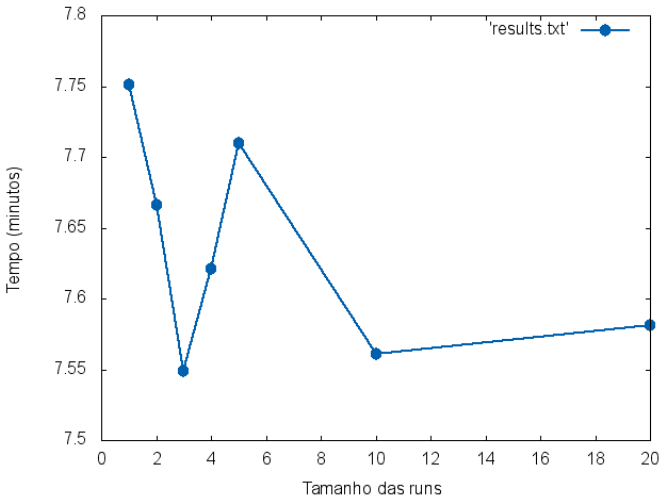
4.1. Tempo de Execução

Esta seção descrevemos os tempos de execução dos programas implementados neste trabalho. A seção 4.1.1 analisa os tempos do indexador e do a seção 4.1.2 do buscador.

4.1.1. Indexação

O tempo de indexação total da melhor configuração proposta foi de 55 minutos limitando o tamanho da memória principal em 45Mb. Para entender um pouco mais o gargalho do indexador foi realizado um Profile do código utilizando o Instruments, um software que faz parte da IDE Xcode. Esse software mostrou que o gargalho da execução do indexador encontra-se no parser contribuindo com mais de 60% do tempo de execução. Ao utilizar o parser gumbo, ganhamos 5 minutos com o tempo de execução comparado com o parser htmlcxx.

Figura 1. Example



Na figura 1, temos um gráfico do tempo de execução do indexador fixando o número de documentos e variando o tamanho das runs. Cada run possui 250 mil ocorrências, então o 1 no eixo x representa 250 mil ocorrências e o eixo y o tempo de execução em minutos. Como podermos perceber o tempo de execução cai com o tamanho das runs. Isso mostra que quanto mais memória disponível para executar este trabalho melhor é o tempo de execução, porém o gargalho permanece no parser quando o gráfico começa a subir o tempo de execução indenpendete do tamanho das runs.

4.1.2. Busca

Nesta seção avaliamos o tempo de busca para diferentes consultas. Primeiramente, a busca precisa carregar todo o vocabulário em memória, esse tempo foi desprezado e avaliamos apenas a busca pelos dois operadores lógicos and e or em relação ao número de termos da consulta. Na figura 2 temos um gráfico do tamanho de uma consulta e o tempo da busca realizada. A linha azul representa o conector lógico and e a linha vermelha o or.

Como podemos perceber, o tempo de busca segue uma complexidade quase constante independente do conector lógico, é difícil analisar esse tipo de desempenho sendo como objetivo dos próximos trabalhos esse estudo.

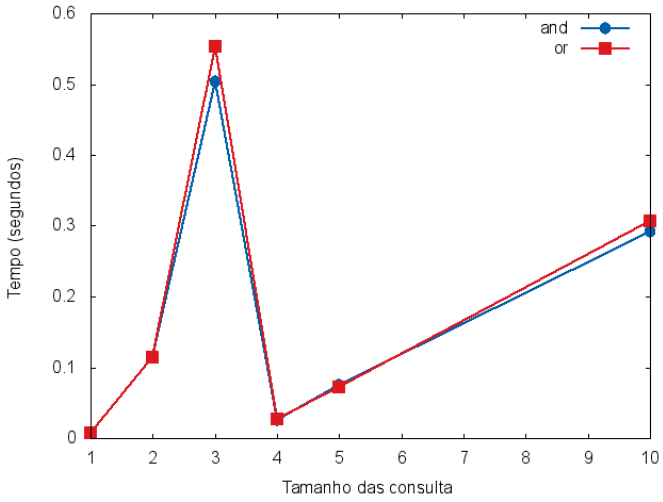
4.2. Tamanhos

O arquivo final de índice invertido tem tamanho de 2.8GB. O tamanho do índice foi reduzido ao utilizar o parser gumbo e também ao remover stopwords do vocabulário que possui 4269136 termos distintos.

5. CONCLUSÃO

Este trabalho teve como objetivo a implementação de um programa para produzir um arquivo de índice invertido. A solução proposta foi baseada no modelo de programação MapReduce onde a fase Map produz em vários arquivos ocorrências de termos ordenadas e a fase Reduce ordena externamente esses arquivos e agrega para produzir o arquivo de índice final.

Figura 2. Example



O trabalho atingiu seus principais objetivos: a prática da implementação de uma parte das tarefas de um buscador web. Uma importante decisão importante para este trabalho foi modelar o problema em cima do modelo de programação MapReducer mantendo uma organização clara do trabalho. A aprendizada adquirida neste trabalho foi muito grande, principalmente no conhecimento adquirido sobre com indexador eficiente e das estruturas padrões existentes na linguagem C++.

Algumas melhorias que poderiam ser consideradas neste trabalho são:

- Uma modelagem mais simples do problema proposta para ser capaz de realizar testes unitários;
- Remoção de letras e números do vocabulário.
- Redução do vocabulário ao passar certas palavras do plural para o singular.

Referências

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (2011). Modern information retrieval - the concepts and technology behind search, second edition.