Trabalho Prático 1
Recuperação de Informação

Luís E. O. Lizardo[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Caixa Postal 702 – 30.123-970 – Belo Horizonte – MG – Brasil

lizardo@dcc.ufmg.br

Resumo. *Máquinas de busca revolucionaram a Internet ao possibilitarem que usuários tenham acesso aos conteúdos mais diversos e de forma rápida. Devido ao grande tamanho da Web, essas máquinas precisam ser eficientes em tempo e espaço. Neste trabalho foi desenvolvido uma máquina de busca simplificada para analisar o processo de construção de um índice invertido. Para isso três programas foram criados: o Index_writer, Index_sorter e o Index_builder. Experimentos mostraram que é possível construir um índice invertido em aproximadamente 70 minutos para uma coleção de quase 1 milhão de documentos.*

1. Introdução

Máquinas de busca são mecanismos importantes na era da Internet. Elas permitem procurar por páginas Web tendo como base palavras que essas páginas possam conter. Para que uma máquina de busca possa fazer uma consulta de forma rápida, ela precisa que as páginas tenham sido previamente indexadas. Devido ao grande volume de conteúdo que a Internet tem atualmente, é necessário que o processo de indexação de documentos seja otimizado para tempo e espaço. Neste trabalho é estudado técnicas e algoritmos de indexação de páginas Web.

O objetivo deste trabalho é desenvolver uma máquina de busca simplificada para indexar e recuperar eficientemente informação de grandes arquivos armazenados em memória secundária, utilizando um índice de arquivo invertido. O desenvolvimento da máquina de busca simplificada pode ser dividido em quatro etapas: (1) *parse*, consiste na extração do conteúdo texto das páginas Web e armazenamento das informações extraídas como tuplas em um arquivo temporário; (2) ordenação, consiste na ordenação do arquivo temporário, executada em disco. (3) indexação, consiste em criar um índice invertido em disco a partir das tuplas ordenadas armazenadas no arquivo temporário. (4) consulta, processamento de consultas por palavras chaves e recuperação dos documentos que as contém.

Neste trabalho foram criados três programas: **index_writer**: extrai o conteúdo texto de páginas web, exclui informações não relevantes como marcadores HTML e *scripts*, armazena as informações de cada termo extraído como uma tripla (termo, documento, frequência) em um arquivo no disco e cria um dicionário de termos, etapa 1; **index_sorter**: ordena o arquivo de triplas gerado pelo *index_writer* utilizando o algoritmo de ordenação externa *Multiway Merge Sort* com escritas *in-place*, etapa 2; **index_builder**: constrói o índice invertido em arquivo, gera o vocabulário em memória utilizando uma função hash perfeita mínima, e permite ao usuário realizar consultas booleanas de termos, etapas 3 e 4.

O restante do artigo está organizado da seguinte forma. A Seção 2 discute um tema relacionado ao trabalho. A Seção 3 descreve a solução proposta, avaliada experimentalmente na Seção 5. A Seção 4 trata de detalhes específicos da implementação do trabalho. A Seção 6 traz conclusões e lista trabalhos futuros.

2. Referencial Teórico

Uma função hash perfeita (PHF) mapeia um conjunto n de chaves para um conjunto de m números inteiros sem colisões., onde m é maior ou igual a n. Se m é igual a n, a função é chamada de mínima (MPHF). MPHF são amplamente utilizadas para o armazenamento eficiente em memória e para a rápida recuperação de itens de conjuntos estáticos, como palavras em linguagem natural.

Existem muitas aplicações para MPHF em diferentes áreas da computação. Em Recuperação de Informação o MPHF pode ser utilizado na construção do vocabulário, que é uma estrutura em memória utilizada para mapear cada palavra à posição da sua lista invertida comprimida em disco.

Neste trabalho foi utilizado o algoritmo BDZ desenvolvido por [Botelho and Ziviani 2008]. Esse algoritmo é simples, eficiente e quase ótimo em espaço. Ele é utiliza 3-grafos aleatórios acíclicos em sua construção, sendo 3-grafo uma generalização de um grafo onde cada aresta conecta 3 vértices. Essa MPHF gasta aproximadamente 2,6 bits por chave ao ser gerada.

3. Metodologia

Neste trabalho foram desenvolvidos três programas que executam partes distintas mas totalmente dependentes no processo de construção de uma máquina de busca. Os detalhes do *Index_writer*, responsável por extrair o conteúdo texto relevante de páginas Web, estão descritos na Seção 3.1. Os detalhes do *Index_sorter*, programa que ordena o arquivo de triplas, está descrito na Seção 3.2. A Seção 3.3 descreve os detalhes do *Index_builder*, responsável por construir o vocabulário, criar o índice invertido e processar consultas. A Figura 1 esquematiza o processo completo de construção do índice invertido da máquina de busca simplificada.

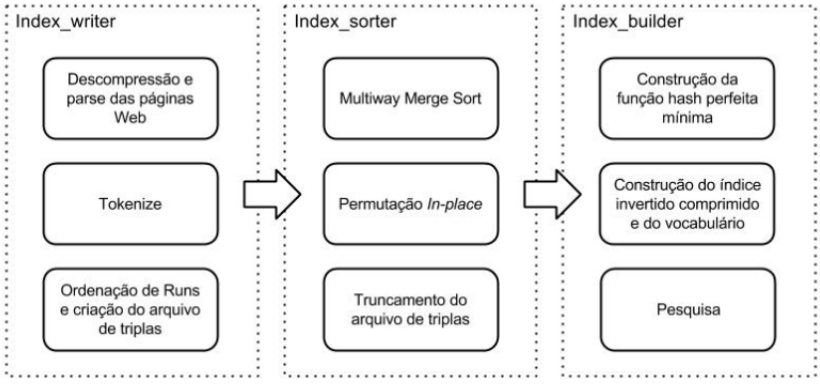

Figura 1. Fases do processo de indexação separadas por programa. Dentro de cada programa as fases seguem um fluxo de cima para baixo.

3.1. Index_writer

O Index_writer tem como função ler as páginas Web que estão em uma coleção comprimida; extrair o conteúdo texto relevante de cada página; separar o texto em palavras ou termos; e criar dois arquivos em disco, o primeiro é o arquivo de triplas contendo, para cada palavra extraída da coleção, uma tripla da forma $(term\_id, doc, freq)$, onde *term_id* é um número inteiro positivo atribuído ao termo encontrado, *doc* é um número inteiro positivo atribuído ao documento onde a palavra foi encontrada e *freq* é a quantidade de vezes que a palavra aparece dentro do documento. Cada item da tripla é representado por um *unsigned int* de 4 bytes. Assim a tripla tem no total 12 bytes. O outro arquivo é o dicionário, ele é texto e contém cada palavra extraída e o seu número inteiro atribuído.

Os detalhes de cada subprocesso do Index_writer estão explicados nas subseções seguintes.

3.1.1. Parser

O Parser tem como tarefa extrair todo o conteúdo de texto das páginas Web a serem indexados. Ele remove marcadores HTML, scripts, cabeçalhos e outros códigos sem conteúdo para seres humanos. Ele foi implementado utilizando a biblioteca do Google Gumbo Parser[1]. O uso dessa biblioteca foi optado por ela suportar HTML5, ter sido amplamente testada, e não ter dependências de outras bibliotecas.

Conforme é mostrado na Seção 5, o parser foi a parte mais lenta de todo o processo de indexação. Isso é provavelmente devido a um baixo percentual de texto relevante em meio aos marcadores HTML. A complexidade do Parser é linear no número de documentos.

3.1.2. Tokenizer

O papel do Tokenizer é retirar do texto os termos - palavras ou números, sem acentuação e símbolos, e em minusculo - que serão indexados na máquina de busca.

Como os textos dos documentos podem estar em qualquer codificação, o Tokenizer converte todo o texto para *wide strings*, onde cada caractere é armazenado com 4 bytes. Depois de convertido, o texto é divido por palavras e elas têm seus acentos removidos, assim como qualquer outro símbolo não presente na tabela ASCII. Depois que as palavras estão limpas, elas são convertidas novamente para *strings* com caracteres de 1 byte.

O Tokenizer faz três passagens sobre o texto. A primeira na conversão para *wide strings*, a segunda para separar as palavras e limpar o texto e a terceira para converter novamente para *strings* com caracteres de 1 byte, sendo está última menor, já que caracteres não indexáveis foram removidos. Assim sua complexidade é $O(n)$, onde n é o tamanho do texto em caracteres.

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

3.1.3. Sort-based inversion

Na medida em que as triplas são criadas, elas são armazenadas em um *buffer* na memória. Quando o *buffer* fica cheio, as triplas são ordenadas, com Quicksort, em ordem ascendente do número do termo e depois do número do documento. Depois as triplas são salvas em um arquivo em disco, que é utilizado na construção do arquivo invertido. Esse processo de ordenação das triplas presentes no buffer, otimiza a construção do índice, pois evita, na etapa seguinte, que o *Index_sorter*, tenha que carregar todas as triplas na memória, ordenar até o tamanho limite do buffer, para então somente poder fazer o *Merge Sort* de todo o arquivo.

Cada conjunto de triplas armazenadas e ordenadas no *buffer* são chamadas de *Runs*. O número total de *runs* necessárias para criar o arquivo de triplas depende do número de triplas geradas e do tamanho do *buffer*. Todas devem ter o mesmo tamanho, e quando não é possível, bytes de *padding* são adicionados para completar o tamanho. Essa restrição é necessária para posteriormente facilitar a permutação in-place, explicada Seção 3.2. No final da execução do algoritmo *Sort-based inversion*, tem-se um arquivo de triplas parcialmente ordenado.

O pseudocódigo desse algoritmo pode ser encontrado em [Witten et al. 1999]. Sua complexidade de tempo depende do algoritmo de ordenação utilizado. Neste trabalho foi optado pelo Quicksort que é $O(n \log n)$ no caso médio e $O(n^2)$ no pior caso, onde n é o número de triplas.

3.2. Index_sorter

O papel do Index_sorter é ordenar o arquivo de triplas armazenado no disco. A ordenação desse arquivo é primordial para a construção do arquivo invertido realizado na etapa seguinte, do Index_builder. O desafio do Index_sorter é realizar essa ordenação sem poder carregar todo o arquivo para a memória, já que o arquivo pode ser muito maior que o espaço disponível.

Para ordenar todo o arquivo, o Index_sorter aproveita do fato das *runs* já estarem ordenadas, e cria o conceito de *Bloco*. Um bloco é um buffer pequeno em memória e muito menor que uma *run*. No total, são utilizados um número igual de blocos ao número de *runs* ordenadas pelo Index_writer, acrescido de um bloco para ser utilizado como saída, no armazenamento das triplas após serem mescladas. Cada bloco de entrada mapeia uma *run* e são lidos do disco seguindo a ordem ascendente das triplas. Neste trabalho um bloco possui 50 KB.

A ordenação de todo o arquivo é feita utilizando um Heap, que tem como entrada as triplas de cada bloco. Do topo do Heap são retiradas as triplas de menor *term_id* e *doc*, que são armazenadas no bloco de saída. Os bytes de *padding* têm a menor prioridade no Heap. Na medida em que os blocos de entrada ficam vazios, eles são substituídos por outros de mesma *run*. Quando o bloco de saída fica cheio, ele é escrito no disco em alguma posição vazia de outro bloco previamente lido para a memória. Quando não há mais blocos para serem lidos do arquivo de triplas e o Heap fica vazio, a ordenação termina.

No final desse processo, tem-se no arquivo de triplas um conjunto de blocos com triplas ordenadas, mas os blocos em si estão fora de ordem. Para colocar cada bloco no

lugar correto, é utilizado um algoritmo de permutação *in-place*. Mais detalhes sobre os algoritmos utilizados nessa etapa podem ser encontrados em [Witten et al. 1999].

Após a ordenação dos blocos, o arquivo é truncado, eliminando os bytes de *padding* adicionados pelo Index_writer para que cada bloco possa ter o mesmo tamanho. Esses bytes de *padding* são úteis para permitir a troca de um bloco por outro durante a permutação.

A complexidade de tempo do Heap é $n \log n$ no caso médio, para n igual ao número de *runs*, vezes o número de triplas. Em geral, para a coleção utilizada no trabalho e os tamanhos de memória utilizados nos testes, o número de *runs* foi baixo. A permutação *in-place* é feita em tempo linear, do número de blocos.

3.3. Index_builder

O Index_builder é o programa responsável por criar o vocabulário e o arquivo com as listas invertidas. Ele usa para isso o dicionário produzido pelo Index_writer e o arquivo de triplas ordenado pelo Index_builder. O vocabulário é um vetor em memória que armazena, para cada palavra da coleção o conjunto: $(term\_str, ptr, tam)$. *term_str* é a *string* da palavra, *ptr* é um ponteiro para a posição no arquivo onde a lista invertida da palavra está armazenada, e *tam* é o tamanho dessa lista invertida. O vocabulário é construído com base em uma Função Hash Perfeita Mínima, gerada a partir do dicionário. A posição de cada palavra no vetor do vocabulário depende dessa função Hash. A biblioteca CMPH[2] foi usada para gerar a função Hash.

O tamanho do arquivo invertido e o tempo gasto para gerá-lo depende linearmente em função do número de palavras e do número de documentos que cada palavra contém. Esse tamanho também é dependente do método de compressão utilizado. Nesse trabalho foi utilizado o método de compressão Elias γ.

O tempo de construção da função hash é linear no tamanho do dicionário, mas ela permite acesso aos termos do vocabulário em $O(1)$.

3.4. Consultas

O Index_builder também é um processador de consultas simplificado. Ele suporta consultas booleanas do tipo termo1 AND termo2 OR termo3 e retorna a lista de documentos que satisfazem a consulta. Uma consulta com AND retorna a interseção das listas invertidas de cada palavra. Já as consultas com operadores OR retornam a união das listas. Não foi tratado precedência entre os operadores, sendo está, na ordem em que aparece na expressão.

Para fazer uma consulta, gasta-se um tempo linearmente proporcional ao tamanho das listas invertidas de cada palavra. A interseção tem uma complexidade de tempo linear no tamanho da menor lista. O tempo de união é linear ao tamanho da soma das duas listas.

4. Implementação

Este trabalho foi implementado em *C++*. Os arquivos com cabeçalho e código fontes estão organizados em pastas dentro do diretório *src*. Os três arquivos principais de execução dos programas estão na raiz desta pasta.

[2]CMPH: http://cmph.sourceforge.net/

As bibliotecas de terceiros necessárias para a compilação dos programas são fornecidas juntamente com o código fonte. Elas estão na pasta *lib* e seus cabeçalhos na pasta *include*. As bibliotecas são: Gumbo Parser, riCode e CMPH citadas anteriormente, e a biblioteca zlib[3].

4.1. Compilação

Para compilar os programas e fazer o *link* as bibliotecas execute no Linux:

```
make all
```

4.2. Execução

Para executar um teste dos programas utilizando a coleção *toyExample*, execute no Linux:

```
make run
```

Para executar os programas individualmente, os seguintes argumentos devem ser passados:

```
./index_writer [-d -m -n -i -c]
./index_sorter [-d -t -m]
./index_builder [-d -t -q]
```

onde as opções são,

- **-d**: diretório de saída, onde os arquivos gerados pela execução serão criados.
- **-t**: nome do arquivo temporário de triplas.
- **-m**: tamanho da memória em MB, se não especificado, 1 MB será utilizado. O tamanho da memória durante a execução tem que ser o mesmo para todos os programas.
- **-n**: número máximo de documentos a serem indexados, se não especificado, no máximo 1 milhão de documentos serão indexados.
- **-i**: diretório da coleção comprimida de documentos.
- **-c**: índice da coleção comprimida de documentos.
- **-q**: caminho completo do arquivo contendo as consultas. Uma consulta por linha.

4.3. Saída dos programas

Além dos arquivos gerados, todos os programas, ao fim da execução, imprimem estatísticas como tempo de execução, número de documentos indexados, número de escritas e leituras em disco etc.

O index_builder especificamente, imprime os resultados das consultas realizadas, mostrando os códigos dos documentos encontrados na busca.

5. Avaliação Experimental

Uma coleção comprimida com 4,7GB de páginas Web foi utilizada para testar a máquina de busca criada.

Os testes foram realizados no sistema operacional Ubuntu 13.10, rodando um notebook com processador 3rd Generation Intel® Core™ i7-3630QM (2.40GHz 6MB Cache), 8GB RAM, HHD 1 TB (5,400 rpm). Foram feitos 5 execuções para cada teste e retirado a média.

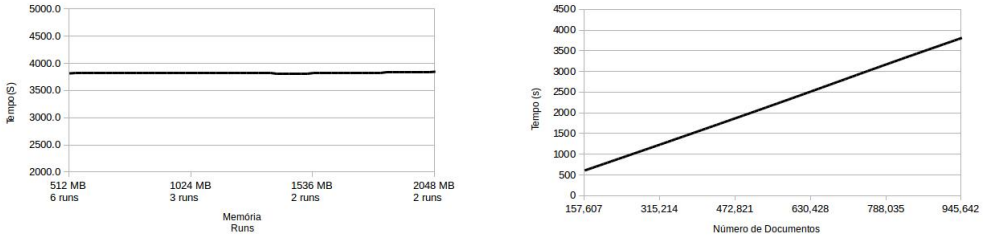

Figura 2. Tempo médio de execução do Index_writer para diferentes valores de memória e de documentos. O gráfico da esquerda mostra apresenta resultados para variação de memória de 512 MB até 2 GB. O gráfico da direita os resultados para valores crescentes do número de documentos.

5.1. Index_writer

O Index_writer é o programa mais caro da máquina de busca simplificada. Sua execução para uma coleção de 945 mil documentos demora cerca de 64 min. A figura 2 apresenta os tempos médios de execução do Index_writer para diferentes valores de memória e de número de documentos.

Com uma memória maior, é possível manter mais triplas no buffer, para depois serem ordenadas e escritas no arquivo de triplas. Dessa forma, quanto maior a memória, menos *runs* são necessárias. Analiticamente, podemos dizer que se um número menor de *runs* é necessário, mais rápido será o processamento, pois menos acessos ao disco serão necessários. No entanto, os resultados mostraram que não houve alteração significativa no tempo de execução. Isso é devido ao fato da operação mais cara do Index_writer ser a de *parser* e *tokenizer*, que representam respectivamente 51% e 46% do tempo de execução.

O aumento do número de documentos, por outro lado, altera notavelmente o tempo de execução, que cresce linearmente com o número de documentos, conforme mostrado no gráfico da direita da Figura 2.

No total foram extraídas 3.908.119 palavras da coleção.

5.2. Index_sorter

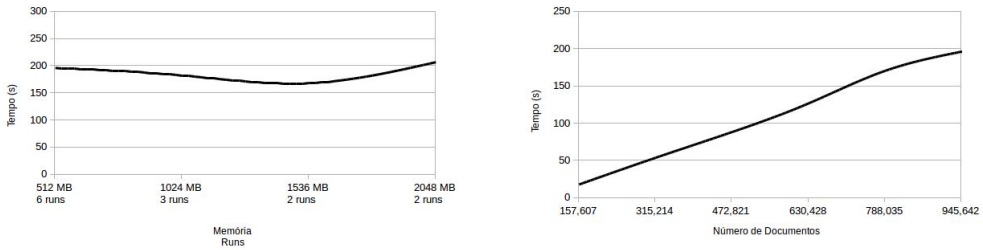

Figura 3. Tempo médio de execução do Index_sorter para diferentes valores de memória e de documentos. O gráfico da esquerda mostra apresenta resultados para variação de memória de 512 MB até 2 GB. O gráfico da direita os resultados para valores crescentes do número de documentos.

[3]zlib: http://www.zlib.net/

A execução do sorter depende do número de *runs* que dependem do tamanho da memória. Se a memória é maior que a coleção, apenas uma *run* é criada, e ela é inteira ordenada já pelo Index_writer. Se são duas *runs*, a ordenação funcionará como um *Merge Sort*. Para 3 ou mais *runs*, o Heap fará o papel de fila de prioridades, o que evita fazer outras mesclagens das triplas. O problema de *runs* grandes, é que elas aumentam o número de bytes de *padding* quando a coleção não consegue preenchê-la totalmente. Esse foi o motivo do ligeiro aumento do tempo de execução mostrado na Figura 3 para 2048 MB. Essa figura também mostra os resultados obtidos, quando o número de documentos são variados, e assim como o Index_writer, foi proporcional.

A Figura 4 mostra o tempo de execução gasto individualmente por cada etapa dos programas. Nessa figura (a direita) é possível notar que o algoritmo de permutação gastou apenas 20% do tempo de execução total da ordenação.

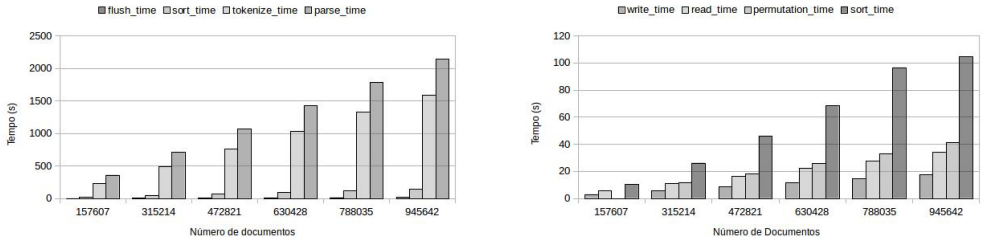

Figura 4. Tempo gasto individualmente por cada etapa do Index_writer (A) e do Index_sorter (B), ao variar o número de documentos.

5.3. Index_builder

O Index_builder foi avaliado em relação ao tempo de construção, do vocabulário e índice invertido, e também ao tempo de consulta.

O tempo médio de construção foi de 18,4 segundos, para o arquivo contendo triplas de toda a coleção. Com compressão esse tempo subiu para 3 minutos, devido a uma implementação não eficiente. O tamanho do índice invertido sem compressão é de 1,8 GB, com compressão o tamanho do arquivo reduz para 400 MB.

Para avaliar o tempo de consulta ao índice, foram pesquisados um conjunto de 88 mil palavras da língua portuguesa e inglesa. O tempo médio total de consulta para esse conjunto foi de 4.2 segundos, dando aproximadamente 48 microssegundos por palavra.

O tamanho do vocabulário criado para a coleção foi de 78,3 MB, gastando aproximadamente 21 bytes por palavra.

6. Conclusão

Neste trabalho foi desenvolvido uma máquina de busca simplificada para avaliar os algoritmos de indexação de documentos Web. Como o volume de dados é muito grande, as operações necessárias para criar o arquivo de índice invertido foram todas realizadas com os dados em memória secundária. Foram implementados os algoritmos: *Sort-based inversion*, para ordenar as triplas na medida que eram produzidas; *Multiway Merge Sort*, para ordenar as triplas em blocos; e Permutação *In-place*, para ordenar os blocos de triplas em

memoria secundária. Para construir o vocabulário da máquina de busca, foi utilizado uma função hash perfeita mínima. O arquivo com as listas invertidas, foi comprimido usando a codificação Elias γ.

Como possíveis melhorias ao trabalho, podemos citar: paralelizar o parser, operação mais cara do processo; compressão do arquivo de triplas; reimplementação do algoritmo de compressão para o torná-lo mais eficiente; e separar a parte de consulta do Index_builder para um novo programa específico;

Referências

Botelho, F. C. and Ziviani, N. (2008). Near-optimal space perfect hashing algorithms. *The thesis of PhD. in Computer Science of the Federal University of Minas Gerais*.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann.