Trabalho Prático 4
Indexador
Recuperação de Informação

Daniel Ferreira Abadi[1]
2018088062

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

1. Introdução

O problema abordado consiste em utilizar nosso trabalho prático anterior para uma coleção de um milhão de páginas. Com o detalhe adicional de que o processo todo não necessariamente caiba na memória principal, mais algumas alterações propostas pelo monitor.

2. Implementação

Assim como no trabalho prático anterior, foi necessário utilizar uma biblioteca para remover todas as marcações HTML, CSS e JavaScript , sendo esta a Gumbo Parser. Na atividade anterior foi decido retirar todos os números do texto, porém nesse trabalho foi decidido que eles deveriam voltar, visto que são de grande importância nas pesquisas atuais. Mas todos os detalhes como manter o hífen, converter todos os caracteres para minúsculo, retirar ”emotes”e caracteres que não são Latinos foram mantidos.

2.1. Construção do arquivo invertido

O arquivo invertido proposto para o trabalho é bem diferente do que nos foi indicado para o trabalho anterior, não necessitando exatamente da mesma estrutura. Por esse motivo foi removida a estrutura criada anteriormente. Para a questão principal de armazenar o termo, o número do documento e as posições na qual aparecem, foi utilizado uma estrutura do tipo ”Lista”, contendo uma estrutura do tipo ”Tuple”de três inteiros.

Para cada termo encontrado foi necessário criar um índice, para evitar uso de memória desnecessário e, também, para facilitar comparações. Isso foi feito por meio de uma estrutura ”Hash”, cuja chave era o termo e o valor associado um inteiro, e todos os termos e seus índices são escritos em um arquivo de texto. Há também um índice para os documentos, e, como o arquivo que nos foi disponibilizado estava configurado de forma que cada linha era um ”JSON”com endereço WEB e conteúdo HTML, os índices foram criados de acordo com as linhas do mesmo. E, dada a estrutura do documento, foi utilizada a biblioteca ”RapidJson”para fazer o ”parsing”. Essa biblioteca não necessita de nenhuma instalação, basta apenas baixar os cabeçalhos e incluir no código fonte.

As mais de um milhão de páginas foram divididas em 32 partes, eram lidas pouco mais de 31 mil páginas, adicionando todos os termos, número do documento e posição na ”Lista”citada anteriormente. Quando se chegava nesse limite pré-estabelecido, todos os elementos da ”Lista”são ordenados e passados para um arquivo de texto e a ”Lista”era esvaziada. Este número, 32, foi escolhido de forma a facilitar a ordenação dos arquivos,

visto que o arquivo total não cabe na memória principal. Para que o programa funcione corretamente, é necessário criar uma pasta chamada ”outputs”na raiz do programa, assim como no anterior.

2.2. Ordenação externa

Como dito anteriormente, o número de arquivos ser 32 foi uma escolha estratégica para facilitar a operação de ordenação, por ser uma potência de dois, funcionando como um ”merge sort”. Inicialmente temos 32 arquivos, que são agrupados de dois em dois, até chegar a um único arquivo.

A cada par de documentos escolhidos é lida a primeira posição de ambos e comparados o primeiro número de cada, sendo este referente ao índice do termo. Se os números forem iguais, a linha do primeiro documento é passada para o novo arquivo gerado, o primeiro documento tem sempre a preferência pois foi gerado com as páginas com índices menores, e então a próxima linha do documento é escolhida. Caso o um tenha o número menor do que o outro, sempre o menor índice de termo tem preferência, seguindo o mesmo comportamento citado acima. Isso se repete até que um dos arquivos acabe, logo após isso todos os valores do arquivo restantes são passados para o arquivo resultado.

2.3. Construção do índice

Como nos foi sugerido, deveríamos criar um índice para o arquivo gerado, contendo o índice do termo, a posição inicial e final de seu bloco e seu IDF. Para armazenar tais informações foi criada uma estrutura ”Hash”, que tem como chave o índice do termo e como valor uma estrutura ”Tuple”com três valores, sendo dois do tipo ”long int”e um do tipo ”float”.

Como o arquivo já está ordenado, é lida linha por linha, mantendo-se o número do termo atual, salvando os páginas em que o termo aparece em uma estrutura ”Set”, que apenas contém termos únicos. Quando o número muda, é então salvo a posição final do último número e a posição inicial do atual, com a função ”tellg”, da biblioteca padrão do C++, é calculado o IDF utilizando o tamanho do ”Set”que é então reiniciado, os dados são guardados na estrutura ”Hash”e também salvos em um arquivo de texto.

3. Dados da coleção

Foram utilizados 1.000.068 documentos, com um arquivo no tamanho de aproximadamente 86 GB. Tal coleção teve 1.857.726 de termos distintos, de acordo com o método de limpeza utilizado nesse trabalho. O arquivo invertido tem o tamanho de aproximadamente 13 GB e índice do mesmo tem aproximadamente 72 KB.

4. Conclusão

Foi construído um arquivo invertido, como o que nos foi apresentado em uma das aulas durante o curso. Contendo o índice do termo, índice do documento a posição no mesmo. Os dados ficam dispostos, inicialmente, em 32 arquivos diferentes e, com uma ordenação externa, são todos juntados em um único arquivo. Com este arquivo, é então criado um novo índice em memória, como especificado.

A dificuldade principal na realização do trabalho foi o tempo de execução, tornando o processo de encontro de erros e ”bugs”difícil. E, para resolver tal problema, foi necessário criar vários pequenos exemplos para cobrir a maior parte dos possíveis casos.