Trabalho Prático 2: Índice Invertido

Marcos Paulo Quintao Fernandes

Maio 2017

1 Introdução

Neste trabalho iremos construir um Índice invertido de uma coleção. Índice invertido é uma estrutura de índice que armazena registros que mapeam um conteúdo em documentos de uma coleção. Tais índices servem principalmente como forma de se realizar buscas rápidas em uma base de texto muito grande. A implementação a ser utilizada consiste em construir registros do tipo $< w, d, f, p >$, onde w é a palavra, d é o documento onde encontramos aquela palavra, f é a frequência da palavra naquele documento,p é a posição em bytes daquela palavra no documento. Esses registros foram gerados para todas as palavras no texto e ordenados em um grande arquivo. No entanto, ordenar esses registros não é uma tarefa trivial, uma vez que seu tamanho pode exceder a memória principal disponível. Tendo como base esse fato, utilizaremos a ordenação externa para resolver esse problema, uma vez que ela reduz o custo de usarmos a memória principal.

2 Solução do Problema

Tendo o conhecimento de quanto de memória principal pode ser utilizado e quais são os documentos a serem processados para a construção do Índice Invertido, dividiremos a construção em 4 etapas:

1. Apartir de cada documento, iremos criar arquivos onde cada linha destes são do tipo $< w, d, f, p >$. (Divisao dos Documentos)
2. Ordenar cada arquivo criado. (Ordenaçao interna)
3. Realizar o merge de cada arquivo gerado. (Merge)
4. Atualizaçao das frequências.

2.1 Divisão dos Documentos

A idéia dessa etapa é construir registros pequenos de tal forma que podemos ordená-los usando a memória principal posteriormente. Desta forma, cada arquivo deverá ter um tamanho inferior ao da memória principal. Como cada

documento pode ser maior que a memória principal, iremos dividí-lo em vários arquivos. Nesta etapa, além de todos os registros estarem desordenados, temos que cada termo terá a frequência 1.

2.2 Ordenação Interna

A idéia dessa etapa é ordenar internamente os arquivos. Observamos que cada arquivo a ser ordenado nessa etapa tem seu tamanho menor que a memória principal disponível (garantido pela primeira etapa). O critério da ordenação será do tipo:

1. ordenar as palavras lexicograficamente w
2. em caso de empate em 1, ordenar pelos indices de documento do termo d
3. em caso de empate em 1,2 ordenar pela frequência do termo no documento f
4. em caso de empate em 1,2,3 ordenar pela posicao do termo no documento p

Observamos que ainda nessa etapa a frequências de todos os termos ainda é igual a 1.

2.3 Merge

A partir do item acima, dado 2 registros nós conseguimos inferir qual deles deve ser priorizado em uma ordenação. Com isso, dado dois arquivos ordenados, conseguimos construir um outro arquivo ordenado a partir desses dois intercalando os menores registros (mesma técnica do mergesort).

Suponhamos que precisamos de ordenar os arquivos enumarados de $[1, N]$. Essa ordenação é equivalente de dar um merge dos arquivos enumerados de $[1, N/2]$ e $[N/2 + 1, N]$. No entanto , as ordenações dos arquivos $[1, N/2]$ e $[N/2+1, N]$ também podem ser divididas até que temos o caso base: $[X, X]$ que no caso é o próprio arquivo ordenado da etapa 2. Esta forma de rearranjarmos o merge foi feita por questões de eficiencia e será discutida mais abaixo.

2.4 Atualização das frequências

Nesta etapa, já possuimos um arquivo com os índices invertidos (ordenados) de cada termo. No entanto, a frequência de cada termo está desatualizada (igual a 1). Para atualizá-la, iremos criar um arquivo auxiliar que manterá registrado na ordem a palavra, qual documento ela pertence, e a sua frequência correta. Dessa forma, a partir desse arquivo auxiliar, conseguimos atualizar a frequência de todos os termos comparando-os nos dois arquivos: se eles forem iguais, modificamos a frequência.

3 Análise de Complexidade

3.1 Complexidade de Tempo

Primeiramente podemos perceber que as 4 etapas sâo sequenciais, logo a complexidade que rege a solução é a soma da complexidade das etapas. Iremos adotar N como o número de palavras a ser considerado, M o limite de memória principal utilizada e K o número máximo de arquivos obtidos na etapa 1, ou seja: $K = \lceil \frac{N}{M} \rceil$. Dessa forma, a complexidade de cada etapa é:

1. Nesta etapa, a divisâo dos documentos é feita em $O(N)$, uma vez que cada termo será processado apenas 1 vez.

2. Nesta etapa, basicamente faremos um quick sort interno para cada arquivo. Desta forma, iremos realizar um quick sort para cada arquivo. Como temos K arquivos e a complexidade de cada quicksort é $O(Mlog(M))$ temos que a complexidade desta etapa é $O(KMLog(M))$.

3. Para vislumbrarmos a complexidade desta etapa, primeiramente vamos relembrar que construímos uma árvore binária de altura $log(K)$ (vide merge sort). Em cada altura desta árvore, iremos intercalar todos os termos em $O(N)$. Dessa forma, a complexidade desta etapa é: $O(Nlog(K))$.

4. Esta etapa tem complexidade $O(N)$ uma vez que cada termo será visitado apenas 1 vez.

Portanto a complexidade final é: $O(KMlog(M)+Nlog(K)) = O(Nlog(M)+Nlog(K)) = O(Nlog(M*K)) = O(Nlog(N))$.

3.2 Complexidade de Memória

Para calcular a complexidade de memória, iremos adotar a mesma notação utilizada na complexidade de tempo. Pelo mesmo motivo que no tempo, a complexidade de memória que rege a solução é a soma da complexidade de cada etapa. Dessa forma:

1. Nesta etapa, cada palavra de todo o texto é duplicada, ou seja gastaremos $O(N)$ de memória.

2. Nesta etapa, basicamente faremos um quick sort interno para cada arquivo. O quick sort utilizado consome $log(M)$ de espaço extra, ou seja, no caso será consumido $O(Mlog(M))$. Como cada quicksort é feito concorrentemente, temos que a complexidade de memória desse processo é $O(Mlog(M))$.

3. Para vislumbrarmos a complexidade desta etapa, primeiramente vamos relembrar que construiremos uma árvore binária de altura $log(K)$ (vide merge sort). Em cada altura desta árvore, iremos escrever todos os registros dos documentos uma única vez, ou seja, será consumido $O(Nlog(K))$

4. Esta etapa tem complexidade $O(N)$ uma vez que cada termo será visitado apenas 1 vez.

Portanto a complexidade de memória é: $O(Nlog(K)) + Mlog(M))$

4 Análise Experimental

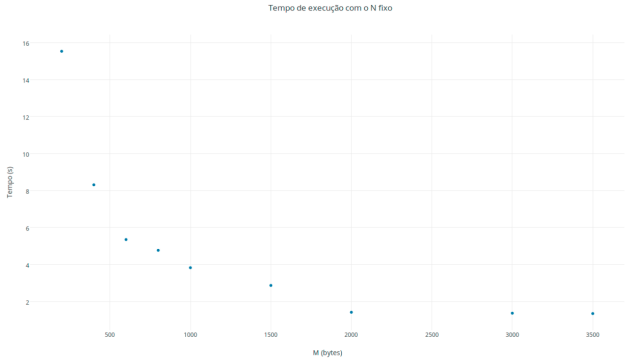
Para avaliarmos o desempenho desse programa utilizaremos uma máquina com as seguintes configurações:

Configurações da Máquina

Para testarmos a eficiência do algoritmo implementado, faremos 2 tipos de teste. O primeiro se consiste em: fixado o número M (quanto de memória principal pode ser utilizada), iremos variar o número de bytes N do documento para analisarmos o comportamento assintótico da solução. O segunto teste se consiste em: fixado o número de bytes N do documento iremos variar o número M (quanto de memória principal pode ser utilizado).

4.1 Teste 1

Para a realização deste teste, foi gerado um Documento com 10^5 palavras aleatórias para observarmos o quanto o desempenho do programa cai quando utilizamos menos a memória principal.

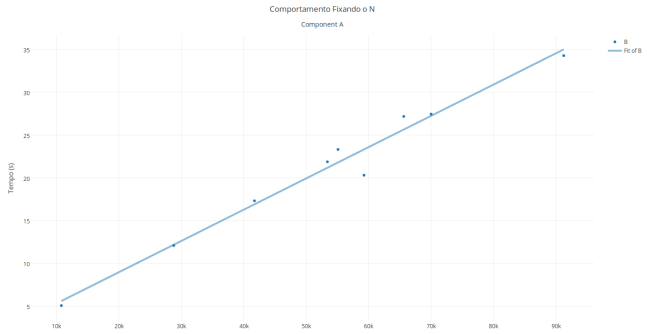


Desempenho Teste 1

Neste experimento podemos evidenciar o quão mais lento é o acesso ao disco em relação ao acesso da memória RAM.

4.2 Teste 2

Para a realização deste teste, foi gerado um documento com N aleatório arbitrariamente grande para um M fixo de 800 bytes. O desempenho esperado era de de uma função do tipo NlogN. O encontrado foi:



Desempenho Teste 2

Neste experimento não podemos evidenciar o LogN da função com clareza uma vez que os valores estão bem próximos.