TP2: Índice Invertido

Documentação

Disciplina: Algoritmos e Estruturas de Dados III

Professor: Olga Goussevskaia

Aluno: Artur Henrique Marzano Gonzaga

Data: 11/06/2017

1 INTRODUÇÃO

O trabalho consiste em auxiliar Hetelberto na construção do *índice invertido* de uma quantidade arbitrária de conversas que não cabe na pequena memória **M** do seu celular, informada na entrada.

No problema apresentado, o *índice invertido* é uma estrutura que mapeia palavras de conversas em documentos. Essa estrutura facilita a recuperação de informação em grandes volumes de dados. Deve-se produzir um índice com tantas linhas quanto palavras nos arquivos das conversas, cada uma no formato "**w**,**d**,**f**,**p**", onde **w** é a palavra, **d** é o documento onde se encontra, **f** é a frequência da palavra e **p** sua posição no documento.

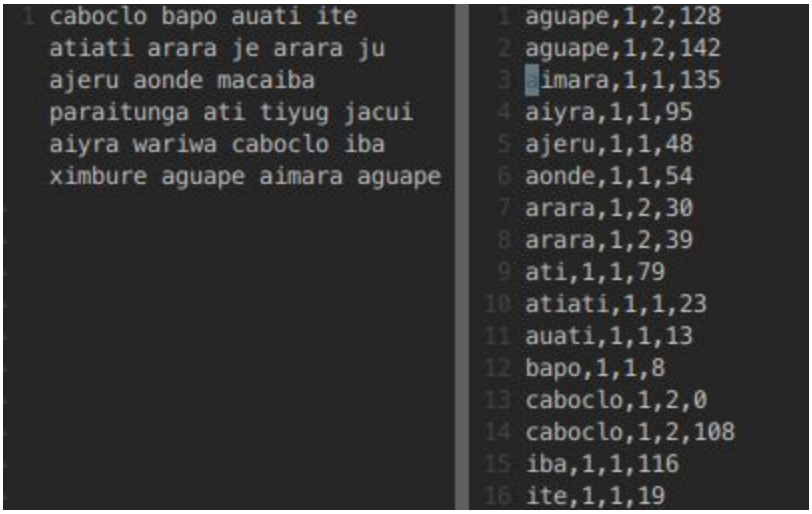

Figura 1. Entrada de exemplo e trecho da saída correspondente.

Para cumprir essa tarefa, é necessário o uso de algoritmos de ordenação em memória secundária. O trabalho é dividido em 6 módulos principais no diretório raíz: *main, list, sorts, invindex, quickext* e *utils*. Um Makefile também foi incluso.

2 SOLUÇÃO DO PROBLEMA

O problema foi solucionado com o uso do *quicksort externo* aplicado em um arquivo binário, para simplificar o acesso indexado aos registros, seguindo o esquema abaixo:

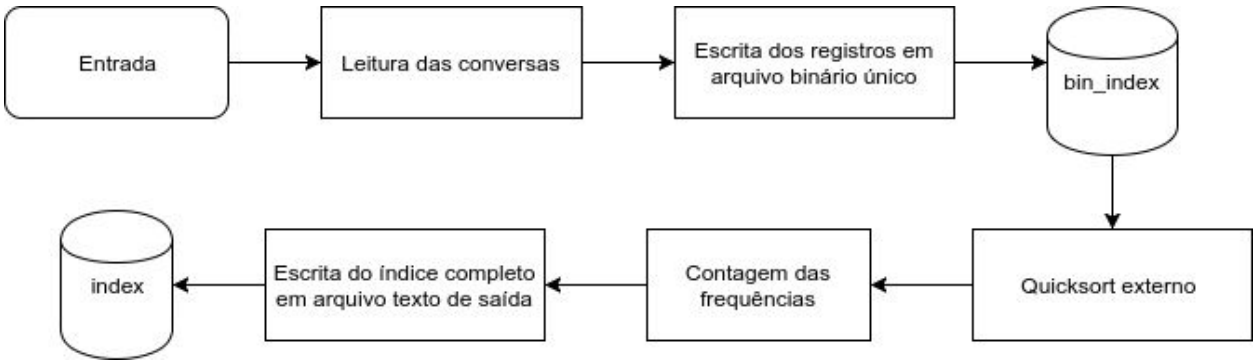

Figura 2. Fluxograma do procedimento empregado na solução do problema.

Os registros utilizados ocupam 32 bytes cada (20 palavra, 4 documento, 4 frequência e 4 posição).

O quicksort externo funciona da seguinte maneira:

1. São alocados **M** bytes para o buffer, que será utilizado como pivô;
2. Leia os **M/2** elementos do início e **M/2** do final do arquivo para o buffer;
3. Ordene o buffer com ordenação interna;
4. Leia o próximo elemento do início ou do final de forma a balancear a escrita;
    a. Se o elemento for menor que o mínimo do buffer, escreva-o no espaço disponível no início;
    b. Se o elemento for maior que o máximo do buffer, escreva-o no espaço disponível no final;
    c. Caso o elemento esteja no intervalo do buffer, escreva o máximo ou o mínimo do buffer, ponha o elemento no buffer e reordene-o;
5. Quando o ponteiro da leitura esquerda ultrapassar o da leitura direita, escreva o buffer;
6. Ordene recursivamente as demais partições.

No procedimento de particionamento, após o preenchimento inicial, o buffer é ordenado com o quicksort interno. Após modificações no buffer, ele é reordenado por inserção, uma vez que é uma escolha suficientemente eficiente para vetores semiordenados.

3 ANÁLISE DE COMPLEXIDADE

Variável	Significado
m	Memória disponível.
f	Frequência da palavra no documento.
p	Quantidade de palavras no arquivo de entrada.
n	Parâmetro principal da função correspondente.
b	Tamanho do registro.

Tabela 1. Significado dos parâmetros da análise.

As funções com ambos os custos constantes e funções não utilizadas na versão final foram omitidas por brevidade.

Arquivo	Função	Complex. Tempo	Complex. Espaço
sorts	partition	$\mathcal{O}(nlogn)$ (melhor e médio) $\mathcal{O}(n^2)$ (pior caso)	$\mathcal{O}(1)$
	quick		$\mathcal{O}(1)$
	insertion	$\mathcal{O}(n)$ (melhor caso) $\mathcal{O}(n^2)$ (pior e médio)	$\mathcal{O}(1)$
invindex	write_pseudo_index	$\mathcal{O}(p)$	$\mathcal{O}(1)$
	write_back	$\mathcal{O}(f)$	$\mathcal{O}(1)$
	build_index	$\mathcal{O}(p)$	$\mathcal{O}(1)$
quickext	fill_mem	$\mathcal{O}(m)$	$\mathcal{O}(m)$
	ext_partition	$\mathcal{O}(n/b)$ (melhor caso) $\mathcal{O}(n/b * log_2(n/m))$ (caso médio) $\mathcal{O}(n^2/m)$ (pior caso)	$\mathcal{O}(m)$
	ext_quicksort_rec		$\mathcal{O}(m)$
	ext_quicksort		$\mathcal{O}(m)$

Tabela 2. Análise de complexidade de tempo e espaço.

4 AVALIAÇÃO EXPERIMENTAL

Além da avaliação de memória com o *valgrind* e da confirmação do trabalho com os *testes toys*, que não fornecem entradas suficientemente exaustivas, entradas maiores também foram geradas. Para esses testes foram variados tanto o *tamanho total das conversas* quanto a *memória disponível*. As conversas foram geradas utilizando palavras aleatórias de um dicionário não muito grande (320 palavras).

Variável	**Mínimo**	**Step**	**Máximo**
Tamanho das conversas (kB)	10	10	500
Memória (32B)	1	10	491

Tabela 3. Parâmetros dos testes realizados.

Todos os testes foram realizados em um *Intel Core i5 3317U 1.7GHz* com *6GiB de RAM* e o trabalho foi desenvolvido na distribuição *Arch Linux*.

Os resultados estão disponíveis em *tests/data e* podem ser visualizados abaixo:

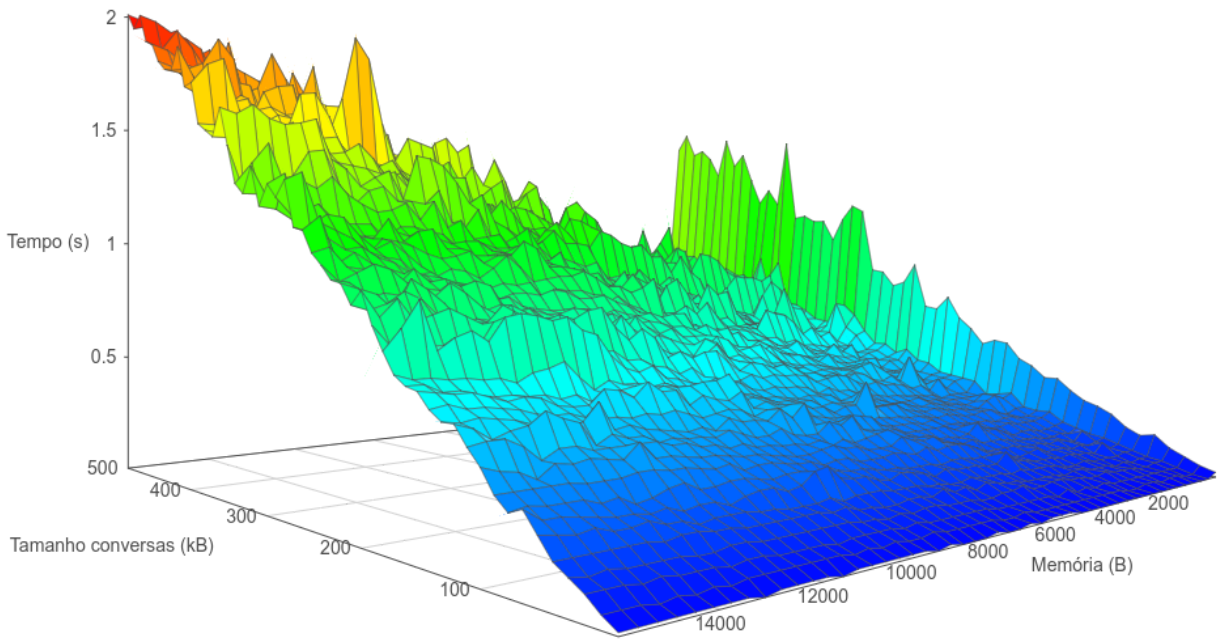

Figura 3. Tempo de execução em função da quantidade de registros e do tamanho total das conversas.

Conclui-se que a memória e o tamanho total das conversas são diretamente proporcionais ao tempo de execução, o que é consistente com o esperado, uma vez que mais memória implica em mais elementos a serem mantidos em ordem pelo quicksort interno.

As execuções do *valgrind*, dos testes toys e dos testes gerados não evidenciaram problemas na implementação do trabalho. Todos os testes podem ser refeitos com as seguintes facilidades inclusas no *Makefile*:

$ make toys
$ make valgrind
$ make massif

5 DETALHES IMPORTANTES

1. O heap é utilizado diretamente uma única vez para alocar o espaço disponível para o *buffer* do quicksort externo.
2. Se uma função utiliza algum espaço na *stack*, ele é auxiliar, pequeno e em geral $\mathcal{O}(1)$.
3. As funções *fopen*, *fread*, *fwrite* e *fseek* são responsáveis por algum overhead indireto de I/O no heap.
4. No cabeçalho do *main.c* podem ser configurados o nível de output (*VERBOSE*, 0/1) da execução, a ordem dos registros a serem ordenados (*ORDER, 0/1*) e o caminho do arquivo temporário utilizado (por padrão, *tmp/bin_index*).
5. No cabeçalho da *quickext.c* pode ser escolhido o método a ser utilizado na reordenação do buffer (dentre *quicksort*, *shellsort* e *insertion*).
6. List.c e list.h são uma simples implementação de um tipo lista de elementos acompanhado de métodos úteis.
7. O diretório **tmp** deve existir no diretório atual para a execução do programa e o arquivo *bin_index* escrito nela será apagado ao fim do processo.

6 CONCLUSÃO

O trabalho foi essencial para fixar o aprendizado de conceitos e algoritmos de ordenação externa, bem como manipulação avançada de arquivos em modo binário, habilidades de primordial importância para aplicação em sistemas com pouca memória, tais como dispositivos móveis e embarcados.