TP1: Banco de Dados com Arvores B+

Vitor Cláudio Chaves de Aguiar

6 de maio de 2016

1 Introdução

O objetivo desse trabalho prático é implementar uma arvore B+ com operações de busca, inserção e impressão das chaves em largura porém utilizando memória secundária. Árvores B+ são usadas muitas vezes em sistemas de banco de dados e elas apresentam pontos positivos tais como ser balanceada e que um nó pode ter vários filhos. Os dados da árvore B+ são armazenados somente nos filhos permitindo uma maior inserção de chaves nos nós internos e uma menor altura.

Os dados são recebidos em forma de um arquivo contento as operações a serem realizadas, a palavra-chave `add` foi reservada para a operação de inserção juntamente com os seus parâmetros separados por tabs. A palavra-chave `search` foi reservada para operação de busca que por sua vez possui o parâmetro da chave separado por um tab. A palavra-chave `dump` foi reservada para fazer o caminhamento em largura na arvore e imprimir suas chaves.

Devido ao número de operações poder ser muito grande, muitas vezes o espaço em memória primária não é o suficiente, devido a isso, é necessário nesse trabalho armazenar todos os dados da árvore em um arquivo em disco e só usar memória primária com a finalidade de fazer a operação e logo depois liberar essa memória.

A solução apresentada consiste em ler o arquivo de entrada, realizar todas as operações e salvar as páginas em disco. Para gravar/ler do disco foi necessário a serialização/deserialização das páginas.

2 Solução do problema

O problema consiste inicialmente em ler o arquivo entrada e realizar as operações na arvore B+ implementada, parâmetros como ordem da arvore, numero de campos que um registro possui e qual desses campos é a chave são passsados na execução do programa. Foram utilizadas referências tais como slides da aula e o livro do Cormen para implementação da árvore.

A árvore implementada possui duas structs diferentes que estão dentro de uma union. Uma para uma página externa e outra para página interna. Um enum permite dizer qual tipo a página é. Ambas estruturas possui um vetor de registros onde cada registro é formado por uma matriz de char. A página interna possui os apontadores para as páginas filho porém foi utilizada de outra estratégia para armazenar esse dado.

Devido ao fato da árvore ser gravada em disco, ao invés da estrutura utilizar ponteiros do tipo página para apontar para seus filhos, foi utilizado a estratégia das páginas internas possuirem um vetor de offsets. O offset vai dizer aonde no arquivo a página está.

Ao realizar as operações é necessário ou gravar uma página no disco ou ler esse dado no disco. Devido a isso, a serialização e a deserialização das páginas é um processo que deve ser feito para que os dados possam ser representados na forma correta dentro do arquivo. Toda págiana ao ser criada recebe um valor de offset referente ao final do arquivo.

Como reflexo disso, toda inserção no disco de uma nova página é feita no final do arquivo, porém se a página já estiver em disco e vai ser feita uma alteração, a serialização vai ser feita no offset que aquela página recebeu na sua primeira inserção. O mesmo vale para deserialização, que recebe o offset da página para buscar

seus dados em disco. A serialização é feita com o tamanho máximo que uma página possa ter.

Algorithm 1: Serialização

```
Data: Páginas P, arquivo Arq, número de registros NR, ordem da árvore O, offset S
aponta pro offset do parâmetro
// Serializa os tipos primitivos
escreve no arquivo o tipo da página
escreve no arquivo o numero de registros
escreve no arquivo o offset
// Serializa os vetores
SerializarRegistros(registros, Arq, NR, O)
if P == Interna then
    SerializarPonteiros(ponteiros, Arq, O)
end
aponta pro final do arquivo
```

O pseudocódigo acima retrata o cenário da serialização onde os tipos primitivos são diretamente serializados e os vetores devem passar por um processo de serializar cada elemento dele. O ponteiro do arquivo fica no offset que foi recebido como parâmetro e logo depois da alteração no arquivo, assegura-se que o ponteiro fique no final do arquivo.

Algorithm 2: Inserção Árvore B+

```
if Raiz = cheia then
    cria nova raíz
    divide raíz antiga entre duas páginas filho
    insere registro
end
else
    insere registro
end
```

O pseudocódigo acima retrata uma particularidade da implementação que sempre verifica na hora de inserir um registro se a raíz está cheia. Se isso ocorrer ele já realiza o processo de divisão dela e prosegue com o fluxo de inserir um registro.

Algorithm 3: Impressão em largura Árvore B+

```
cria-se um fila de páginas
adiciona a raíz na fila
define a raiz pra ser Pagina
for Filho ∈ Pagina do
    adiciona Filho na fila
    if Fila == MAX then
        aumenta fila
    end
end
retira a primeira página da fila
imprime a página com todas suas chaves
define o próximo da fila pra ser Pagina
```

O pseudocódigo acima retrata uma particularidade da implementação onde foi criado uma fila para imprimir corretamente os dados. Porém a fila implementada é definida com um tamanho inicial e sempre que esse numero for atingido, uma nova fila é criada com um tamanho maior, os dados da antiga são copiados pra nova e a nova passa ser utilizada.

3 Análise de complexidade

3.1 Tempo

- As funções: `desalocarNoInterno`, `desalocarNoExterno`, `criarNoExterno`, `criarNoInterno` do arquivo BTree.c nas linhas 24, 38, 59, 82 respectivamente possuem um loop externo que percorre o tamanho máximo de registros e um loop interno que percorre o número de campos de um registro. Devido a isso, a complexidade dessas funções é **O(ordemArvore*camposRegistro)**.

- A função `buscarEmLargura` no arquivo BTree.c na linha 127 possui um loop mais externo na linha 138 junto com um loop interno na linha 143 que garante acessar cada nó da árvore uma vez, ou seja, O(n) sendo n = numero de registros. Deste modo, todos os nós da árvore são armazenados em uma fila e posteriomente são retirados e todos os seus registros impressos. Portanto a complexidade dessa função é **O(n)**.

- A função `pesquisar` no arquivo BTree.c na linha 214 compara a chave do registro a ser procurado com a chave dos primeiros registros de uma página interna. Ao chegar na folha a função verifica se a chave se encotra lá. A busca nas páginas internas é feita por uma chamada recursiva que vai percorrer toda a altura da árvore, como reflexo disso, a complexidade será de **O(long(n))**.

- As funções `inserir`, `dividirFilho`, `inserirNaArvore` no arquivo BTree.c nas linhas 316, 389, 442 respectivamente juntas formam a inserção da árvore. A complexidade da operação de inserção na árvore B+ é long(n) sendo n = numero de registros da árvore. É possível observar no pior caso da inserção, aonde a altura da ávore cresce, a função `inserirNaArvore` chama a `dividirFilho` para fazer a divisão da raíz. Porém essa função só possui loops com valores constantes passadas na execução do programa, como por exemplo na linha 417 que são feitas (ordemArvore/2)+1 iterações. Diante disso, a função que nos interessa é a `inserir`. A função percorre a árvore em níveis com chamadas recursivas, procurando a folha correta para inserir o registro, deste modo ela percorre toda a altura da árvore ocasionando na complexidade **O(long(n))**.

- As funções `serializarPagina` e `deserializarPagina` no arquivo Serializacao.c nas linhas 25 e 67 respectivamente chamam as funções de serializar/deserializar registro e serializar/deserializar ponteiro se for uma página interna. No pior caso então, está função pros registros executa um loop de ordem da árvore vezes o numero de resgistros iterações e para os ponteiros um loop de ordem da árvore iterações. Devido a isso a complexidade dessas funções são **O((ordemArvore*camposRegistro) + ordemArvore)**.

É possível concluir então que em um cenário aonde todas as operações são de impressão;dump (pior caso), a complexidade total do programa será **O(n)**.

3.2 Espaço

Para a análise do espaço desse trabalho é necessário olhar para a estrutura da árvore e as operações em disco de serialização e deserialização. Na perspectiva da memória primária, a complexidade é **O(1)** devido ao fato de poucos registros são deserializados para memória principal entre as operações. Agora, na perspectiva da memória secundária, a complexidade será **O(n)** pois toda página da árvore é guardada em disco, ou seja, é guardado n registros na memória secundária.

4 Análise Experimental

Para realizar a análise experimental foi relizado testes variando a ordem da árvore em um arquivo de entrada com 10000 operações. Cada instância foi testada 3 vezes em uma máquina Intel Core i5 2,67GHz de 4GB de memória RAM. Para medição do tempo de execução foi utilizado o comando `time` no linux e o cálculo final do tempo foi obtido pela média dos 3 testes.

Ordem da Árvore	Tempo
5	82s
10	134s
50	191s
100	72s

5 Conclusão

Neste trabalho foi implementado uma arvore B+ capaz de fazer operações de inserção, busca e impressão em largura. Além disso, a árvore utiliza da memória secundária para realizar suas operações. O problema do uso da memória secundária foi resolvido com serialização/deserialização das páginas com a utilização de offsets no arquivo. A análise de complexidade de tempo foi comprovada através de diversas entradas de teste geradas.