TP1: Banco de Dados com Arvores B+

Giovani Amaral Barcelos

5 de maio de 2016

1 Introdução

O objetivo desse trabalho prático é construir um banco de dados simplificado usando uma árvore B+, que conterá operações simples de busca de chave, inserção de registro e impressão por nível. Além disso, temos o detalhe que entre duas operações somente a raíz ficará em memória, ao passo que todos os outros nós ficarão em disco.

A solução apresentada consiste na implementação de uma árvore B+ e, para manter os nós em disco, foi usado serialização e desserialização dos mesmos, sempre que for preciso carrregar algum nó para a memória ou persisti-lo em disco.

2 Solução do problema

2.1 Primeira parte - árvore B+

O primeiro grande desafio do problema é implementar uma árvore B+. Alguns detalhes permitiram fazer isso com mais facilidade: 1)as chaves dos registros sempre são únicas e 2)sempre são valores numéricos. Desta forma, foi adaptado o algoritmo da árvore B do [1] para se tornar uma árvore B+, atendendo nossa necessidade. Caso o leitor não saiba, a diferença é que, na arvore B+, só temos registros nas folhas, e cada folha tem um apontador para seu irmão a direita, facilitando caminhamento sequencial. No código da árvore, modificamos algumas coisas para atender ao critério de somente a raiz em memória.

2.1.1 Fluxo externo - leitura de dados

Para lidar com as operações básicas, temos um loop que lê todas as linhas do arquivo de entradas, e para cada linha é identificada a operação(*add*, *search* ou *dump*) e então é chamada a função correspondente.

Algorithm 1: Pseudo-código do fluxo principal

```
1  for linha ∈ arquivo do
2  |  campos ⟵ processar_linha(linha, numero_campos, operacao)
   |  if operacao == "add" then
3  |  |  registro ⟵ cria_registro(campos)
4  |  |  inserir(raiz, arquivo_arvore, registro)
5  |  end
6  |  if operacao == "search" then
7  |  |  registro ⟵ busca(arquivo_arvore, raiz, chave, numero_campos)
8  |  |  Print(registro)
9  |  end
10 |  if operacao == "dump" then
11 |  |  busca_largura(raiz, arquivo_saida, arquivo_arvore, numeroCampos)
12 |  end
13 end
```

2.1.2 Inserção

O código da inserção pode ser aproximado pelo código abaixo. Pode-se notar que sua base se parece com o código do do [1], mas aqui nós desalocamos um nó, evitando que tenha mais de um nó em memória. Essa alteração é feita na linha 11, que faz com que sempre que precisamos dividir um nó, nós desaloquemos o que não vamos usar mais nessa inserção. Caso formos usar novamente, isso será decidido dentro da função de inserir com espaço.

Algorithm 2: Pseudo-código da função inserir

Data: nodo* raiz, FILE* arquivo, Registro* reg
Result: nodo* raiz

```
1  // Salvamos a ordem em uma variável para facilitar o uso
2  ordem ← no → ordem
3  r ← raiz
4  // Máximo de registros
5  if r → numeroRegitros == ordem then
6  |  nodo * s ← aloca_nodo()
7  |  raiz ← s
8  |  s → numeroRegitros ← 0
9  |  separa_filho(arquivo, s, 1, r, reg → numeroCampos)
10 |  // Diferente do Cormem, precisamos desalocar devido ao nosso critério
   |     especial
11 |  desaloco_nodo(r, reg → numeroCampos)
12 |  // Passamos o nodo no qual vamos tentar inserir e 1 para indicar que é a
   |     raiz
13 |  insere_nodo_com_espaco(s, arquivo, reg, 1)
14 else
15 |  insere_nodo_com_espaco(r, arquivo, reg, 1)
16 end
17 return raiz
```

No algoritmo abaixo, nós estaremos inserindo um registro em um nó que tem espaço. Se esse nó for uma folha, inserimos nele. Se não, vamos para seu filho que contenha o valor correto para inserção. A grande alteração desse algoritmo que faz com que ele seja compatível com nosso requerimento, é que sempre, ao terminar uma inserção, nós desalocamos um nó caso ele não seja uma raiz(linha 18). Caso precisemos caminhar na árvore para inserir mais embaixo, iremos desalocando os nós a medida que formos descendo. Isso é visto na linha 39 e 45.

Algorithm 3: Pseudo-codigo da funcao inserir_nodo_com_espaco

Data: nodo* raiz, FILE* arquivo, Registro* reg, int is_raiz
Result: void

```
// Salvamos a ordem em uma variável para facilitar o uso
ordem ← no → ordem
i ← r → numeroRegitros
if r → folha then
    // Movemos os registros para o lado
    while i >= 1 and reg →< x → chaves[i] do
        x → chaves[i + 1] ← x → chaves[i]
        x → registros[i + 1] ← x → registros[i]
        i ← i − 1
    end
    // Insere registro na posição correta
    x → chaves[i + 1] ← reg → chave
    x → registros[i + 1] ← reg
    x → numeroRegistros ← x → numeroRegistros + 1
    serializar(fp, x, x → registros[i + 1] → numeroCampos, x → offset)
    // Desalocamos o nó, caso ele não seja uma raiz
    if !is_raiz) then
        desaloca_nodo(x, reg → numeroCampos)
    end
end
else
    // Decidimos para qual filho vai este registro
    while i >= 1 and reg →< x → chaves[i] do
        i ← i − 1
    end
    i++
    nodo* no
    aloca_memoria(no)
    // Recuperamos o filho correto do arquivo
    desserializar(arquivo, x → offsetFilhos[i], no)
    // Se estiver cheio, vamos dividí-lo
    if no → numeroRegistros == ordem then
        separa_filho(arquivo, x, i, no, reg → numeroCampos)
        // Checamos se vamos inserir no da esquerda ou direita
        if reg → chave > x → chaves[i] then
            i++
        end
    end
    desaloca_nodo(no, reg → numeroCampos)
    // Recuperamos novamente o nó, caso o nó destino tenha mudado após a
       divisão
    aloca_memoria(no)
    desserializar(arquivo, x → offsetFilhos[i], no)
    // Se não é raiz, desalocamos
    if !is_raiz) then
        desaloca_nodo(x, reg → numeroCampos)
    end
    // Agora passamos 0, pois o no sendo passado não é a raiz
    insere_nodo_com_espaco(no, arquivo, reg, 0)
end
```

Como último algoritmo importante da parte de inserção, temos o separa_filho, que recebe um nó e um pai, e separa o filho em dois.

Algorithm 4: Pseudo-codigo da funcao separa_filho

Data: nodo* x, FILE* arquivo, nodo* y, int i, int numero_campos
Result: void

```
1  Seja ordem a ordem da árvore
2  aloca_memoria(z)
3  if y → folha then
4  |  separa_folha(y, z, ordem)
5  end
6  else
7  |  separa_interno(y, z, ordem)
8  end
9  // Movemos os valores dos offsets do filhos para a direita, para inserir o
       novo valor do novo nó gerado na divisão
10 for j ← x → numeroRegistros + 1to j >= 1 + 1 do
11 |  x → offsetFilhos[j + 1] ← x → offsetFilhos[j]
12 end
13 x → offsetFilhos[i + 1] ← z → offset
14 // Movemos agora as chaves
15 for j ← x → numeroRegistros to j >= 1 do
16 |  x → chaves[j + 1] ← x → chaves[j]
17 end
18 x → chaves[i] ← y → chaves[(ordem/2) + 1]
19 x → numeroRegistros + +
20 serializar(arquivo, x, numeroCampos, x → offset)
21 serializar(arquivo, y, numeroCampos, y → offset)
22 serializar(arquivo, z, numeroCampos, z → offset)
23 desaloca_nodo(z, numeroCampos)
```

2.1.3 Busca em largura - Impressão

Os algoritmos acima nos ajudarão a analisar a complexidade geral do nosso algoritmo e especifica em funções. Vamos ver aqui apenas mais 4 funções importantes: 2 da árvore (busca em largura e pesquisa) e 2 das operações em disco (serialização e desserialização).

Algorithm 5: Pseudo-codigo da funcao busca em largura

Data: nodo* raiz, FILE* arquivo_saida, FILE* arquivo_arvore, int numero_campos
Result: void

```
1  Seja f uma fila encadeada
2  Enfileira(raiz)
3  while !f− > vazia do
4  |  n ← Desenfileira()
5  |  for filho ∈ n do
6  |  |  if !n → folha then
7  |  |  |  Enfileira(filho)
8  |  |  end
9  |  end
10 |  Print(n → registros)
11 |  desaloca_nodo(n)
12 end
```

2.1.4 Busca de chave

Nossa próxima e última operação da árvore sera a busca:

Algorithm 6: Pseudo-codigo da funcao busca de chave

```
Data: nodo* raiz, int k, FILE* arquivo_arvore, int numero_campos
Result: Registro*
Seja registro_encontrado um ponteiro para registro
while i <= x → numeroRegistrosANDk => x → chaves[i] do
    i++
end
if x → folha then
    if k == x → chaves[i − 1] then
        registro_encontrado ← x → registros[i − 1]
        desaloca_nodo(x, numeroCampos)
        returnregistro_encontrado
    end
    else
        desaloca_nodo(x, numeroCampos)
        return NULL
    end
end
aloca(no)
desserializar(arquivo_arvore, x → offsetFilhos[i], no)
desaloca_nodo(x, numeroCampos)
returnbusca(arquivo_arvore, no, k, numeroCampos);
```

2.2 Segunda parte - Serialização e Desserialização

Nosso grande desafio foi manter apenas a raiz em memória entre uma operação e outra. Só é permitido ter a raiz em memória, e para isso, não poderiamos ter apontadores apontando para os filhos, mas sim guardamos offsets que indicariam a posição do filho no arquivo. Desta forma, podemos recuperar um nó sabendo a sua posição no arquivo.

2.2.1 Serialização

Na hora de serializar, era necessário guardar o maior espaço que um nodo poderia ocupar. Como exemplo, vamos imaginar uma folha com 3 registro em uma árvore ordem 5. Se salvamos apenas 3 registros no disco e esta arvore recebe mais um, precisariamos escrever além do espaço reservado para ela no arquivo, ou seja, sobreescreveríamos informações importantes para outros nodos.
Desta forma, a melhor solução foi preencher com 0's onde poderia vir a existir um registro.

O código da serialização pode ser tangenciado pelo seguinte pseudo código:

Algorithm 7: *Serializacao*

Data: nodo* raiz, long long unsigned offset, FILE* arquivo_arvore, int numero_campos
Result: void

Seja metadata um vetor de *longlongunsigned* com 5 posições
$metadata[0] \leftarrow no \rightarrow numeroRegistros$
$metadata[1] \leftarrow no \rightarrow folha$
$metadata[2] \leftarrow no \rightarrow ordem$
$metadata[3] \leftarrow no \rightarrow offset$
$metadata[4] \leftarrow no \rightarrow prox$
$EscreveArquivo(arquivo\_arvore, metadata)$
`// Serializamos agora o vetor de chaves`
$serializa\_vetor\_chaves(arquivo\_arvore, no)$
`// Serializamos agora o vetor de offsets dos filhos`
$serializa\_vetor\_offsetFilhos(arquivo\_arvore, no)$
`// Se o nó é uma folha, serializamos também seus registros`
if $no \rightarrow folha$ **then**
    $serializa\_matriz\_registros(arquivo\_arvore, no \rightarrow registros, no \rightarrow ordem, no \rightarrow folha, numeroCampos, no \rightarrow numeroRegistros)$
end

Você pode estar se perguntando o motivo de não colocarmos, por exemplo, o vetor de chaves no metadata. Isso se dá pois ele é um ponteiro, e não queremos serializar um endereço, mas sim uma lista de valores. Por isso precisamos serializar um a um.

2.3 Desserializacao

Por fim, dado um offset, essa função deve ser capaz de ir até essa posição no arquivo e carregar o nó que está armazenado lá, através da leitura e conversão de seus bytes.

Algorithm 8: *Desserializacao*

Data: nodo* no, long long unsigned offset, FILE* arquivo_arvore
Result: void

Colocamos o ponteiro do arquivo na posição indicada por $offset$
Seja no um no com mempria previamente alocada
$no \rightarrow numeroRegistros \leftarrow le\_arquivo(longlongunsigned, 1, arquivo\_arvore)$
$no \rightarrow folha \leftarrow le\_arquivo(longlongunsigned, 1, arquivo\_arvore)$
$no \rightarrow ordem \leftarrow le\_arquivo(longlongunsigned, 1, arquivo\_arvore)$
$no \rightarrow offset \leftarrow le\_arquivo(longlongunsigned, 1, arquivo\_arvore)$
$no \rightarrow prox \leftarrow le\_arquivo(longlongunsigned, 1, arquivo\_arvore)$
$aloca\_memoria(no \rightarrow chaves)$
$aloca\_memoria(no \rightarrow offsetFilhos)$
$desserializa\_vetor\_chaves(arquivo\_arvore, no \rightarrow chaves, no \rightarrow ordem)$
$desserializa\_vetor\_offsetFilhos(arquivo\_arvore, no \rightarrow offsetFilhos, no \rightarrow ordem)$
`// Se for uma folha, lemos os registros`
if $no \rightarrow folha$ **then**
    $desserializa\_registros(arquivo\_arvore, no, no- > ordem)$
end

3 Análise teórica do custo assintótico de tempo

Nas analises seguintes, consideraremos M a ordem da árvore. Além disso, chamaremos de Z o tamanho do novo nó criado na separação e Y o tamanho do nó que foi dividido. Desta forma, X é o nó pai dos dois citadas anteriormente. Consideramos também C sendo o número de campos que um registro possui e N sendo o número de registros da árvore.

É importante ressaltar que Y é sempre M/2 e Z é sempre M/2 ou (M/2)-1, mas podemos desconsiderar esse -1 e utilizaremos Y e Z para facilitar o entendimento.

desaloca_nodo - Decidi analisar esta função pois ela é utilizado ao longo de todo o código. Entretanto, sua complexidade é simples, pois ela possui um for que roda o numero de registro, e para cada vez que roda chama desaloca_registro, que roda o número de campos C. Logo, nossa complexidade é O(MC).

aloca_nodo - A justificativa desta análise é igual a da anterior. Sua complexidade é O(MC), pois ela chama a funçao de serializar.

serializar - *Algorithm*7 - Nossa função tem algumas operações O(1). Na linha 9, chama uma função que serializa as chaves. Como ela serializa o máximo de chaves que for possível(M), sua complexidade é O(M). Na linha 11 temos a mesma ideia da serialização de chaves, mas temos um offset a mais que chaves, logo roda M+1 vezes. No pior caso, se for um registro, ele serializa o número máximo de registros(M)(linha 14) e cada registro possui C campos, logo temos um for que roda C vezes.

No final, a complexidade de nossa função é : 1 + M + M+1 + (M*C) = 2M * MC = M(2+C) = MC. Complexidade O(MC)

Desserialização *Algorithm*8 - Essa complexidade se assemelha a compexidade de serialização. Serializamos as chaves (O(M)) e offsets (O(M+1)). No pior caso de ser um registro, serializamos no máximo M registro e cada um com C campos. Logo, nossa complexidade é de M + M + 1 + MC = 2M + MC = O(MC)

separa_interno - temos um for que roda Z+1 vezes e o outro que roda Z vezes, totalizando 2Z+1. Logo a complexidade da nossa função é O(Z);

separa_folha - temos um for que roda Z vezes, logo a complexidade é O(Z)

separa_filho - *Algorithm*4 - Essa função chama um das duas funções citadas acima(linha 4 ou 7), com complexidade 0(Z). Na linha 10, temos um for que desloca os offsets do nó pai que roda o número de registros do nó pai - i. Esse número é, no pior caso, M - 1, pois o máximo que o pai pode ter de registro é M e o mínimo que i pode ser é 1. O(M-1)

Já na linha 15, temos outro for que roda o mesmo tanto que o for anterior. O(M-1).

Nas linhas 20, 21 e 22 chamamos a função de serialização, com complexidade O(MC).

Não podemos esquecer que é feita uma chamada para aloca_nodo e desaloca_nodo, e suas complexidades são, respectivamente, O(MC) e O(MC). Logo, a complexidade final é (M-1)*2 + 3*(MC) + Z + MC + MC, ou seja, M - 2 + 5MC + Z = M + MC + Z e como Z é M/2 e podemos ignorar a constante,temos 2M + MC que é igual da ordem **O(M + MC) ou O(MC)**

Insere_nodo_com_espaco *Algorithm*5 -

Caso o nó que iremos inserir é uma folha, temos na linha 6 um while que roda, nó maximo, o número de registros em um nó, ou seja, M.

Na linha 15 fazemos uma chamada a serializar, que é O(MC). Na linha 18, desalocamos um nó, que tem complexidade O(MC).

Logo, a complexidade desta parte é M + MC + MC = M(1 + 2C) = O(2MC) = O(MC).

Já na segunda parte, caso o nó que estamos olhando não seja uma folha, possuimos na linha 23 um while que roda no máximo M vezes, que é o número máximo de registros.

Na linha 30 e 42, fazemos uma chamada para desserialização, que tem o custo de MC cada, totalizando 2MC.

Podemos ter, na linha 33, uma chamada para separa filho, com complexidade O(MC). Na linha 39 e 45 desalocamos um nodo, então temos 2MC de custo.

Desta forma, a complexidade da segunda parte é M + 2MC + 2MC = M(1 + 4C) = 4MC ou O(MC).

No pior caso, a altura da nossa arvore e $log_M(N)$. Entao a segunda parte vai rodar $log_M(N)$ vezes ate

chegar na folha e entao rodar a primeira parte uma vez. Logo nossa complexidade e $(log_M(n) - 1 * MC) + MC$, ou seja, ordem de complexidade: $O(log_M(n) * MC)$

Inserir - *Algorithm*2- No pior caso, alocamos um nodo (linha 6), desalocamos um nodo(linha 11), chamamos separa_filhos(linha 9) e chamamos insere nodo com espaco(linha 13), $MC + MC + MC + log_M(n) * MC$, que equivale a dizer $O(log_M(n) * MC)$

Busca por chave - *Algorithm*6 Temos um while na linha 2 que roda no máximo M vezes. Caso seja uma folha, temos apenas a complexidade de desalocar um nodo, que é O(MC). Sendo uma folha, nossa complexidade é M + MC = MC.

Não sendo uma folha, vamos desserializar(MC) na linha 17 e desalocar(MC) na linha 18, com complexidade 2MC ou O(MC).

Vamos rodar a segunda parte $log_m(n) - 1$ vezes, pois é o numero de nós que passaremos até chegar em uma folha. Então nossa complexidade será $(log_m(n) - 1 * MC) + MC$. Conclusão: $O(log_m(n) * MC)$

Busca em largura - *Algorithm*5 - Temos um while na linha 3 que visita todos os nós da arvore, então sua complexidade é O(N), sendo N o número de nós da árvore.. Na linha 11, deslocamos um registro, complexidade O(M). Todas as operações de fila são O(1). Então a complexidade é O(NM).

3.1 Análise teórica do custo assintótico de espaço

Para fazer essa análise, vamos considerar as principais estruturas NODO e REGISTRO. A estrutura registro possui um long(4 bytes), um unsigned long long(8 bytes) e uma matriz de char de tamanho 30*C. Logo, sua complexidade de espaço é

$$4 + 8 + 30C = 30C = O(C)$$

Já a estrutura nodo possui 3 inteiros (12 bytes), um long long unsigned int(8 bytes), um apontador de long long unsigned com tamanho M (8*M), um apontador de long long unsigned com tamanho M+1(8*(M+1)) e uma matriz de registros contendo M registros, onde cada um tem complexidade O(C). Se for uma folha, nossa complexidade deverá considerar os registros. Se for um nó interno, ele será menor.

Desta forma, nossa complexidade de espaço total de um nó no pior caso é é

$$12 + 8 + 8M + 8M + 8 + MC = 16M + MC = O(MC)$$

Para obter a complexidade total do programa, basta a complexidade obtida acima pelo número de nós ou páginas existentes.

4 Análise de experimentos

Para realizar a análise experimental, foi implementado um gerador de testes que gera instruções baseadas em um número de campos e um campo índice pre determinado. Para medir o tempo de execução do código, foi utilizada o comando time do Ubuntu. Cada teste foi rodado 5 vezes e foi feita uma média com os tempos. Para realizar os testes foi utilizada uma máquina com Linux Ubuntu 15, processador I3 e 6GB de RAM.

A segunda coisa a ser analisada foi a quantidade de bytes resultantes no arquivo que guarda a árvore com suas informações.

A seguinte tabela é fruto de testes com uma árvore com ordem 4 e 4 campos:

Número de registros	Tempo execução	Tamanho do arquivo da árvore	Bytes Alocados
644	0.037	533.224	1,632,554
2015	0.103	1.374.304	5,661,010
2016	0.106	1.374.408	5,664,290
2017	0.149	1.374.408	5,667,066
2018	0.150	1.376.408	5,670,210
2019	0.152	1.376.496	5,673,530
10080	0.594	6.858.600	32,474,738
20160	1.229	13.704.680	69,308,042
17348	1.176	11.796.096	59,037,386

Dobrando a ordem, ficamos com 4 campos e ordem 8:

Número de registros	Tempo execução	Tamanho do arquivo da árvore	Bytes Alocados
644	0.055	517.736	2,301,738
2015	0.098	1.612.120	7,803,650
2016	0.101	1.612.120	7,807,562
2017	0.136	1.612.120	7,811,842
10080	0.605	8.029.528	32,474,738
20160	1.193	16.067.832	89,368,570

Dobrando os campos, ficamos com ordem 4 e 8 campos:

Número de registros	Tempo execução	Tamanho do arquivo da árvore	Bytes Alocados
644	0.041	835.112	2,359,770
2015	0.136	2.599.184	7,958,018
2016	0.167	2.599.288	7,962,514
10080	0.615	1.302.3664	43,713,938
20160	1.366	2.605.1856	91,179,530

Através das analises experimentais, foi possível concluir que, quando precisamos dividir um nó e gravamos somente mais um nó no arquivo, seu tamanho aumenta em 2088 bytes. Além disso, vemos que a complexidade é praticamente linear, pois dobrando o número de registros, os outros valores superam um pouco o dobro dos valores do original

Entretanto, ao dobrar a ordem ou os campos, nossos resultados não dobram. Aumentam, mas não dobram.

5 Bibliografia

Referências

[1] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.