**A Árvore-B e uma
Proposta de Implementação**

Andréa Zisman

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Valdemar W. Setzer**

Durante a elaboração deste trabalho, a autora recebeu apoio financeiro do CNPq e da FAPESP

-São Paulo, Junho de 1993-

A Árvore-B e uma Proposta de Implementação

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Andréa Zisman e aprovada pela comissão julgadora.

São Paulo, 10 de junho de 1992.

Banca examinadora:

- Prof. Dr. Valdemar W. Setzer (orientador) - IME-USP
- Prof. Dr. Siang W. Song - IME-USP
- Prof. Dr. Nívio Ziviani - DCC-UFMG

aos meus pais

Meraldo e Sylvia

Resumo

Esta tese é composta por 2 tópicos principais: uma resenha extensa em *árvores-B* e uma proposta de implementação de um gerenciador de arquivos baseado em um tipo especial de *árvore-B*.

A resenha cobre o histórico das *árvores-B*, sua complexidade e variações, algoritmos para o controle de concorrência, compactação, bem como sistemas com versões e segurança. Para ter uma idéia de como funcionam os sistemas comerciais baseados nas *árvores-B*, descrevemos os aspectos mais importantes dos sistemas VSAM da IBM e BTRIEVE da NOVELL.

Nossa proposta apresenta alguns aspectos importantes como: uma estrutura uniforme para índices primários e secundários, a possibilidade de construir arquivos de índices para qualquer arquivo de dados existente, uma nova solução para o controle de concorrência permitindo compactação concorrente, através de um novo algoritmo. Contém também especificações detalhadas de parâmetros que o usuário deve fornecer ao sistema.

Abstract

This thesis has two main topics: an extensive survey on *B-trees* and a proposal for implementation of a file manager based on a special type of *B-trees*.

The survey covers the history of *B-trees*, their complexity and variations, algorithms for concurrency control, compactification, as well as version systems and security. To have an idea of what commercial systems based on *B-trees* consist of, we describe the most relevant features of IBM-VSAM and NOVELL's BTRIEVE.

Our proposal has some original aspects: an uniform structure for primary and secondary indeces, the possibility of constructing index files for any existent data file, a new solution for concurrency control allowing concurrent compactation under a new algorithm. It also contains some detailed specifications about the parameters the user may give to the system.

Agradecimentos

Ao Prof. Dr. **Valdemar W. Setzer** pela constante orientação, incentivo, paciência, dedicação, confiança e por acreditar na realização deste trabalho. A atenção, amizade e carinho recebidos ultrapassaram em muito o que se pode esperar de um orientador.

Ao casal **Vania e Joel Goldbaum** pelo apoio, carinho, compreensão, torcida, afeto, paciência, amizade e amor recebidos em todo o tempo de convivência.

Ao amigo **Alfredo Goldman** pelo carinho, estímulo, companherismo, apoio, mas principalmente pela grande amizade demonstrada durante todos o programa de mestrado (cursos, exames de qualificação, tese, ...).

Ao Prof. Dr. **Carlos Humes** pelas importantes "conversas", nos momentos oportunos, que estimularam e ajudaram na continuidade do curso.

Ao **Dany Heller** por compreender e acreditar na importância deste trabalho.

Aos **amigos** que direta e indiretamente ajudaram e incentivaram na elaboração deste trabalho.

A **Wania, Gislaine, Edna, Adenilza e Joelma** pelo carinho demonstrado.

O meu muito obrigada!

Conteúdo

Prefácio	xi
1 A Árvore-B	1
1.1 Introdução	1
1.2 Definição de Árvore- B	3
1.3 Complexidade	5
1.4 Operações Básicas: Busca, Inserção e Remoção	8
1.4.1 Busca	8
1.4.2 Inserção	11
1.4.3 Remoção	15
1.5 Escolha da Ordem	20
1.6 Fator de Ocupação de Espaço	20
2 VARIAÇÕES DA ÁRVORE-B	26
2.1 Árvore-2-3	26
2.2 Árvore- BB (Binary B-Tree)	28
2.3 Árvore- B^0	30

2.4	Árvore- B^*	31
2.5	Árvore- B^+	36
2.6	Árvore- B Pré-Fixada	39
2.7	Árvore- B com Multitabelas	45
2.8	Árvore- B^L	46
2.9	Árvore- B Histórica	48
2.10	Árvore- BD (Árvore- B Digital)	50
2.11	Organização Bounded Disorder	53
2.12	Árvore- Bit	56
2.13	Outras Variações	59
2.13.1	Árvore- M (Multiway Tree)	59
2.13.2	Árvore- T_B	61
2.13.3	Árvores para Campos Múltiplos	62
3	TÓPICOS ESPECIAIS	67
3.1	Compressão de Índices	67
3.2	Compactação de Árvores- B	71
3.2.1	Método de Rosenberg e Snyder	71
3.2.2	Algoritmo de Permutação	73
3.3	Organização Ótima das Árvores- B com Elementos de Tamanho Variável	75
3.3.1	Algoritmo de Análise Local	75
3.3.2	Algoritmo de Análise Global	76
3.3.3	Algoritmo para Construir Uma Árvore de Altura Mínima	78

3.4	Sistemas de Versões para as Árvores- B^+	80
3.4.1	Método do Nó Gordo Extendido para as Árvores- B^+	80
3.4.2	Método do Campo Gordo	82
3.4.3	Método do Bloco de Versão Pura	84
3.5	Árvores- B com Vários Discos	86
3.6	Segurança	89
4	CONCORRÊNCIA	94
4.1	Introdução	94
4.2	Solução devida a Samadi	99
4.2.1	Definições	99
4.2.2	Descrição de Funcionamento	100
4.3	Soluções devidas a Bayer e Schkolnick	101
4.3.1	Definições Gerais	101
4.3.2	Solução 1	103
4.3.3	Solução 2	105
4.3.4	Solução 3	106
4.3.5	Solução 4 (Generalização)	108
4.4	Solução devida a Miller e Snyder	111
4.4.1	Definições	112
4.4.2	Descrição de Funcionamento	113
4.5	Solução devida a Lehman e Yao	117
4.5.1	Definições	117

4.5.2	Descrição de Funcionamento	118
4.6	Solução devida a Sagiv	121
4.6.1	Definições	121
4.6.2	Descrição de Funcionamento	121
4.7	Solução devida a Kwong e Wood	124
4.7.1	Definições	124
4.7.2	Descrição de Funcionamento	125
4.8	Outras Soluções	131
4.9	Comparação entre as Soluções	135
5	EXEMPLOS DE SISTEMAS COMERCIAIS	137
5.1	Sistema BTRIEVE	137
5.1.1	Introdução	137
5.1.2	Características Gerais do Gerenciador de Arquivos	139
5.1.3	Controle de Integridade, Concorrência e Segurança	142
5.1.4	Comandos e Operações	145
5.2	Sistema VSAM	147
5.2.1	Introdução	147
5.2.2	Características Gerais dos Arquivos Sequenciados pelo Índice	148
5.2.3	Análise de Alguns Aspectos	150
6	PROPOSTA DE IMPLEMENTAÇÃO	152
6.1	Introdução	152
6.2	Características de Índices	153

6.3	Estrutura da Árvore de Índices	154
6.4	Possíveis Estruturas para Os Nós da Árvore	162
6.5	Alocação Física	165
6.6	Concorrência	167
6.7	Operações	172
6.8	Reorganização do Arquivo de Dados	173
6.9	Compactação Total	174
6.10	Parâmetros Fornecidos pelo Usuário	176
6.11	Quadro de Avisos (Informações para o Usuário)	178
 Conclusão		 179
 Bibliografia		 181

Prefácio

A idéia deste trabalho surgiu de solicitações de uma entidade financeira que estava necessitando de um gerenciador de arquivos eficiente para microcomputadores. Propusemos inicialmente a elaborar um projeto de um gerenciador de arquivos com *árvores-B* para implementação dos índices. Com base nesse fato, fizemos uma pesquisa nas principais universidades brasileiras, tendo encontrado uma única dissertação de mestrado, da Universidade Federal de Pernambuco [21] com uma abordagem sucinta. Realizamos um levantamento bibliográfico sobre tudo que envolve as *árvores-B* e encontramos cerca de 100 trabalhos, como pode ser verificado na bibliografia no final do texto.

Face a extensa bibliografia e ao fato da última resenha sobre o assunto, artigo de Comer [19] (1979), ser relativamente antiga, não abordando determinados assuntos como algumas variações da estrutura, algoritmos de compactação, uso das *árvores-B* com várias versões, algumas soluções para o problema de concorrência, entre outros, concluímos que seria importante para o meio científico e profissional a elaboração de uma resenha sobre o tema envolvendo *árvores-B*.

O trabalho a ser apresentado está dividido em 6 capítulos, estruturados de forma original. O capítulo 1 faz um apanhado teórico englobando histórico, definições, análise da complexidade dos nós, número de valores do índice e altura, operações básicas de busca, inserção e remoção e uma análise do fator de ocupação médio das *árvores-B*.

O capítulo 2 aborda todas as variações da *árvore-B* existentes na literatura e estruturas que deram origem à mesma. Gostaríamos de esclarecer que tudo que envolver a estrutura de *árvore-2-3* será apenas citado, em função de sua importância histórica, não sendo retratado de forma detalhada, por fugir do tema.

No capítulo 3 reunimos alguns tópicos especiais como: compressão de índices, algoritmos de compactação para as *árvores-B*, algumas organizações com elementos de tamanho variável, uso da estrutura em sistemas com várias versões e vários discos e um método de segurança.

No capítulo 4 fizemos uma resenha detalhada sobre as soluções para o problema de concorrência seguida de uma comparação entre elas.

No capítulo 5 damos exemplos de 2 gerenciadores de arquivos comerciais que usam *árvores-B* (IBM-VSAM e BTRIEVE).

Finalmente, no capítulo 6, sugerimos uma proposta de implementação para um geren-

ciador de arquivos. Esta proposta é constituída quase que totalmente por nossas idéias, a menos dos casos referenciados. Em virtude da extensão do trabalho realizado não fizemos simulações nem implementações.

Em todo o texto que se segue, tudo que for de nossa autoria e sugestão será referenciado através de verbos na primeira pessoa do plural, por exemplo, "notemos" , "observamos", "sugerimos", "cremos", etc.

Capítulo 1

A Árvore-B

1.1 Introdução

O desenvolvimento de dispositivos de armazenamento secundário facilitou a gravação, a recuperação e a atualização de uma grande quantidade de dados, reunidos logicamente em coleções chamadas de *arquivos*. Os arquivos são compostos por *registros* e estes, por *campos*. Os campos podem ser *simples* ou *compostos*. Cada campo simples contém um dado elementar, isto é, que não pode ser subdividido em outros dados; já os campos compostos são formados pela concatenação de vários campos (eventualmente não contíguos) de um único registro; obviamente, do ponto de vista lógico o registro, se não for um dado elementar, é um campo composto. Os registros são, por sua vez, agrupados em *páginas*, as quais são copiadas dos dispositivos de memória secundária (discos, fitas) para a memória primária, sempre que um processo deseja examinar ou modificar algum dado, em apenas uma instrução de leitura. Na realidade, as páginas são seções de tamanho fixo que dividem a memória secundária.

Operações no processamento de arquivos envolvem 2 tipos importantes de acesso, o *sequencial* e o *indexado*. O primeiro é executado através da busca linear, registro a registro, em todo o arquivo de entrada até atingir o registro com o dado desejado; já o segundo obtém um registro sem precisar varrer todo o arquivo, com o auxílio de um *índice*, o qual é uma estrutura formada conceitualmente por um campo simples ou composto, cujo *valor* é usado na busca dos registros.

Uma organização particular de índices é a *sequencial-indexada*, onde os valores de um índice são ordenados fisicamente. Uma das formas dessa organização consiste na construção de um arquivo denominado de *arquivo de índices*, contendo valores de índices

associados a informações que permitem a localização dos registros de dados. Essas informações podem ser ponteiros para as páginas de dados ou outro tipo de informação que possibilite a localização dos mesmos. Temos assim, uma subdivisão dos arquivos em 2 tipos: os arquivos de índices, compostos por *registros de índice* e os *arquivos de dados* propriamente ditos, compostos por *registros de dados*. Os registros de índice podem ser organizados em um arquivo de índices em forma de árvore, denominada de *árvore de índice*. Cada página de registros é então um nó da árvore.

Um índice é *primário* se os registros de dados são ordenados dentro do arquivo de dados segundo os valores do índice. Havendo um índice primário para um arquivo de dados, qualquer outro índice para esse arquivo é denominado de índice *secundário* que, portanto, não influencia a organização daquele arquivo. Se um índice tem valores únicos, isto é, não ocorrem 2 registros diferentes de dados com o mesmo valor do índice, ele é chamado de *chave*. Em muitos artigos da literatura usa-se a denominação “key” em lugar de “index”, o que não será o nosso caso. Além disso, encontra-se também a confusão entre “índice primário” e “chave”. Consideraremos “índice primário” como uma questão de eficiência e “chave” como restrição de integridade sendo, portanto, noções independentes. Um índice secundário pode ser chave.

Com o objetivo de permitir um processo de recuperação eficiente, houve o desenvolvimento de várias formas de organização de arquivos. Uma das técnicas de organizar os arquivos sequenciais-indexados consiste na *árvore-B*, uma árvore de busca balanceada, projetada para ser usada em unidades de disco magnético ou qualquer outro meio de memória secundária de acesso direto.

De acordo com Comer [19], por volta de 1960 os produtores de computadores e vários grupos de pesquisas desenvolveram, independentemente, os chamados “métodos de acessos” para os arquivos de suas máquinas. Ele afirma que H. Chiat, M. Schwartz e outros da Sperry Univac Corporation, em conjunto com a Universidade de Case Western Reserve, desenvolveram e implementaram um sistema no qual as operações de inserção e busca eram feitas de maneira parecida com as da *árvore-B*. Mencionam também que independentemente, B. Cole, S. Radcliffe, M. Kauffman e outros desenvolveram um sistema similar na Control Data Corporation, em conjunto com a Universidade de Stanford.

Segundo Comer, a denominação *árvore-B* (“B-Tree”) apareceu pela primeira vez em 1970, no Boeing Scientific Research Labs, devida a Bayer e McCreight, que produziram em 1972 o seu famoso artigo [9] ¹.

¹Comer menciona o desconhecimento da origem do nome *árvore-B* (B-tree), afirmando que uns atribuem o ‘B’ ao fato dessa estrutura ser ‘Balanceada’, outros a ‘Boeing’ ou até mesmo a ‘Bayer’.

1.2 Definição de Árvore-B

Para um certo índice, uma *árvore-B* é uma árvore orientada de *ordem* K , onde K é um número natural, se e somente se ela obedece às seguintes propriedades:

1. Cada nó da árvore é armazenado em uma e apenas uma página do disco².
2. Cada nó, exceto a raiz, tem m valores de tamanho fixo do índice, onde $K \leq m \leq 2K$, armazenados no nó em ordem não-decrescentes.
3. A raiz tem entre 1 e $2K$ valores do índice.
4. Todo nó n da árvore ou é uma folha, isto é, não tem descendentes, ou tem $m+1$ filhos; a cada um desses filhos está associado um ponteiro p de tamanho fixo armazenado em n .
5. A cada valor i de índice em um nó está associado um ponteiro d para o registro de dados que contém i .
6. Todos os nós folhas estão no mesmo nível, ou seja, todos os caminhos da raiz até uma folha têm o mesmo comprimento.

Um nó que contém m valores de um índice $i_1, i_2, \dots, i_m$ e $m+1$ ponteiros $p_0, p_1, \dots, p_m$, pode ser representado como na figura 1.1, de acordo com a definição de Bayer e McCreight [9]. Os valores do índice estão ordenados de forma crescente ($i_1 < i_2 < \dots < i_m$). p_j , $1 \leq j < m$, aponta para a sub-árvore que contém nós com valores compreendidos entre i_j e i_{j+1} ; p_0 e p_m apontam para as subárvores com valores menores do que i_1 e maiores do que i_m respectivamente. Cada d_j é o ponteiro para o registro de dados que contém o valor i_j . Nas folhas os ponteiros p_j são indefinidos, isto é, o conteúdo dos mesmos é igual a *nil*. A figura 1.2 ilustra um exemplo numérico de uma *árvore-B* de ordem $K = 2$ e altura $h = 2$, na qual não representamos os ponteiros d_j 's para os registros de dados. O símbolo será usado em todo o texto para representar um ponteiro vazio (*nil*). No texto que se segue, ignoraremos os ponteiros para os registros de dados, devendo-se considerá-los como existentes em cada nó mencionado.

²A partir de agora os termos *página* e *nó* serão utilizados como sinônimos.

Figura 1.1: Representação de um nó da *árvore-B* com m valores de índices.

Figura 1.2: Exemplo de uma *árvore-B* de ordem $K = 2$ e altura $h = 1$.

Seja n um nó da árvore como na figura 1.1; $n(p_j)$ um nó filho de n apontado por p_j ; $I(p_j)$ o conjunto dos valores de índices contidos nos nós da sub-árvore para a qual $n(p_j)$ funciona como raiz. Dessa forma, uma das condições abaixo sempre ocorre:

1. $\forall i \in I(p_0), i < i_1$
2. $\forall i \in I(p_j), i_j < i < i_{j+1} (j = 1, 2, \dots, m-1)$
3. $\forall i \in I(p_m), i > i_m$

A estrutura da *árvore-B* tem as seguintes vantagens:

1. Desprezando-se a raiz a ocupação de espaço é de no mínimo 50%, uma vez que cada nó, exceto a raiz, tem que ter pelo menos K valores de índices, quando o total permitido é cerca de $2K$. Todavia, no caso médio essa ocupação é de 69,3% como será mostrado na seção 1.6 deste capítulo.
2. O espaço para a árvore é alocado e liberado à medida que o arquivo aumenta e diminui, economizando espaço.
3. Permite uma busca “eficiente” de registros de dados quando forem dados determinados valores do índice.

4. A ordem dos valores do índice é mantida e permite que sejam feitos processamentos baseados nesta ordem, ou seja, encontrar predecessores e sucessores, e fazer buscas dado um intervalo de valores do índice. Este é o armazenamento denominado de “sequencial indexado”.
5. Algoritmos simples exceto em casos especiais.

Definições Gerais

Antes de encerrar essa seção, vejamos algumas definições que serão úteis para o restante do texto.

Considere $n : p_0, i_1, p_1, \dots, p_{m-1}, i_m, p_m$ um nó da *árvore-B* e n' e n'' dois filhos de n apontados por p_j e p_{j+1} respectivamente.

1. n é *nó completo* se $m = 2K$.
2. n torna-se *sobrecarregado* se $m = 2K + 1$.
3. n torna-se *subcarregado* se $m = K - 1$.
4. O *grau* de n é igual a $m + 1$ (número de filhos).
5. O valor de índice i_{j+1} , $1 \leq j < m$, é *separador* de n' e n'' .
6. No texto, quando não houver confusão com outros usos, empregaremos *valor* em lugar de *valor do índice*.
7. A *profundidade* de n é dada pelo comprimento do caminho (número de arestas) deste nó até a raiz da árvore, passando uma única vez por cada nó do caminho. Para as *árvores-B* todos os nós folhas têm a mesma profundidade.
8. A *altura* de n consiste no comprimento de um caminho de n até uma folha da árvore. A altura h de uma árvore é igual à altura de sua raiz. Quando a árvore é vazia $h = -1$; quando a árvore contém um único nó $h = 0$.

1.3 Complexidade

Bayer e McCreight [9], apresentam o cálculo dos números mínimo e máximo, N_{min} e N_{max} , de nós de uma *árvore-B*. Considere $\mathcal{A}$, uma *árvore-B* de ordem K e altura $h \geq 0$.

Claramente, para $\mathcal{A}$ tal que $h = 0$, isto é, formada apenas pela raiz, tem-se que $N_{min} = 1$. Mas, por outro lado, quando $\mathcal{A}$ tem altura $h > 0$, o número de nós é mínimo quando a raiz tem apenas um valor do índice e todos os outros nós têm K valores do índice. Dessa maneira, tem-se 2 nós na profundidade 1, $2(K + 1)$ nós na profundidade 2, $2(K + 1)^2$ nós na profundidade 3 e assim sucessivamente até a profundidade h , onde tem-se $2(K + 1)^{h-1}$ nós.

Assim,

$$N_{min} = 1 + 2((K + 1)^0 + (K + 1)^1 + (K + 1)^2 + \cdots + (K + 1)^{h-1}) = 1 + 2 \sum_{j=0}^{h-1} (K + 1)^j$$

Resolvendo a somatória

$$N_{min} = 1 + \frac{2}{K}((K + 1)^h - 1), h \geq 0$$

Analogamente, é possível calcular o número máximo de nós. Para $\mathcal{A}$ tal que $h = 0$, $N_{max} = 1$. Todavia, quando $h > 0$, este número é maximizado quando todos os nós têm $2K$ valores do índice. Portanto, a árvore contém $(2K + 1)$ nós na profundidade 1, $(2K + 1)^2$ nós na profundidade 2 e $(2K + 1)^h$ nós no nível das folhas. Logo,

$$N_{max} = \sum_{j=0}^h (2K + 1)^j$$

Resolvendo a somatória

$$N_{max} = \frac{(2K + 1)^{h+1} - 1}{2K}, h \geq 0$$

Para uma *árvore-B*, o número de nós N é dado por:

$$\begin{cases} 1 + \frac{2}{K}((K + 1)^h - 1) \leq N \leq \frac{1}{2K}((2K + 1)^{h+1} - 1) & \text{se } h \geq 0 \\ N = 0 & \text{se } h = -1 \text{ (árvore vazia)} \end{cases}$$

Com base nos números mínimo e máximo de nós da *árvore-B* é possível encontrar, facilmente, as quantidades mínima e máxima, I_{min} e I_{max} , dos valores de índices para $\mathcal{A}$. Portanto, como o número mínimo de valores do índice em cada nó da *árvore-B*, exceto para raiz, é K , então o mínimo desses valores para toda a árvore pode ser obtido por (o 1 é devido a raiz):

$$I_{min} = 1 + KN_{min} = 1 + K\left(\frac{2((K + 1)^h - 1)}{K}\right) = 2(K + 1)^h - 1, h \geq 0$$

Por outro lado, o número máximo de valores do índice em cada nó da árvore é $2K$. Logo, para toda a árvore este número é:

$$I_{max} = 2KN_{max} = 2K \frac{((2K+1)^{h+1} - 1)}{2K} = (2K+1)^{h+1} - 1, h \geq 0$$

Assim, o número de valores de índices I , em uma *árvore-B* satisfaz:

$$\begin{cases} 2(K+1)^h - 1 \leq I \leq (2K+1)^{h+1} - 1 & \text{se } h \geq 0 \\ I = 0 & \text{se } h = -1 \end{cases}$$

Através desses resultados, torna-se simples calcular a altura da *árvore-B* em função de um número dado I de valores do índice.

$$\begin{aligned} I &\leq (2K+1)^{h+1} - 1 \\ \log_2(I+1) &\leq \log_2(2K+1)^{h+1} \\ h+1 &\geq \frac{\log_2(I+1)}{\log_2(2K+1)} \\ h &\geq \log_{2K+1}(I+1) - 1 \end{aligned}$$

Por outro lado,

$$\begin{aligned} I &\geq 2(K+1)^h - 1 \\ \log_2(I+1) &\geq \log_2(2) + \log_2(K+1)^h \\ h &\leq \frac{\log_2(I+1) - \log_2(2)}{\log_2(K+1)} \\ h &\leq \log_{K+1} \frac{I+1}{2} \end{aligned}$$

Portanto,

$$\begin{cases} \log_{2K+1}(I+1) - 1 \leq h \leq \log_{K+1} \frac{I+1}{2} & \text{se } I \geq 1 \\ h = -1 & \text{se } I = 0 \end{cases}$$

Assim, a altura de uma *árvore-B* é no pior caso $\log_{K+1} \frac{I+1}{2}$, isto é, $O(\log_K I)$. Como era de se esperar, quanto maior o número de valores do índice em cada nó, menor a altura da árvore. A escolha do valor de K será mostrada adiante.

1.4 Operações Básicas: Busca, Inserção e Remoção

Neste item, vamos supor como [9], uma *árvore-B* para um índice que é uma chave. No capítulo 6 abordaremos o caso geral como extensão deste.

1.4.1 Busca

O procedimento de busca de um valor i do índice é bastante simples, assemelhando-se ao processo utilizado nas árvores binárias, com a diferença de que a escolha do próximo nó a ser seguido não é binária e sim $(m+1)$ -ária, onde $K \leq m \leq 2K$. Dessa forma, quando um nó n com m valores de um índice for visitado, procura-se um valor $i_j = i$, ou $i < i_1$, ou $i > i_m$, ou $i_j < i < i_{j+1}$ ($1 \leq j < m$) determinando-se o ponteiro p_0 , p_m ou p_j apropriado. Se após esta verificação for encontrado o índice sendo procurado, então a busca teve sucesso; do contrário, o nó apontado por esse ponteiro é alcançado e o processo continua até se encontrar o valor procurado ou um ponteiro igual a *nil* (indefinido), isto é, não existe nó descendente; o procedimento atingiu um nó folha que não contém i , mostrando com isso o insucesso da busca.

Para se empregar o algoritmo de busca nos algoritmos de inserção e remoção, é necessário dar como saída deste não só o endereço n do último nó folha visitado ou do nó em que o valor i foi encontrado, como também a posição j . Se i não foi encontrado, j corresponde à posição de inserção de i , devolvida na saída.

Os valores de índice em um nó podem ser examinados através de uma busca sequencial, quando o seu número for pequeno, ou seguindo sugestões de autores como Knuth [44], Maruyama e Smith [63], por uma busca binária quando a quantidade dos mesmos for elevada.

Maruyama e Smith propuseram um outro método para realizar uma busca interna em um nó denominada de *busca em raiz quadrada* ("square root search"), a qual deve ser usada com valores do índice de tamanhos fixos não comprimidos e caracteriza-se por dividir o conjunto dos $I(n)$ valores do índice em um nó n , em $\sqrt{I(n)}$ intervalos. Em seguida, executa-se uma busca sequencial nos valores máximos de cada intervalo, afim de determinar o intervalo correto que é percorrido sequencialmente.

Algoritmo de Busca

O algoritmo Busca, cujo corpo (e dos demais próximos algoritmos) será apresentado por meio de uma tabela de decisão com o formato descrito em Nagayama (1990) [71], é recursivo. A entrada do algoritmo é uma quádrupla ordenada (n, q, j, i) , onde: n contém o endereço do nó corrente sendo visitado; q é o endereço do nó pai de n , contendo um ponteiro que referencia n ; j contém a posição deste ponteiro em q e i é o valor do índice sendo procurado. No início n contém o endereço da raiz da árvore e, para permitir o começo do processamento, atribui-se os valores artificiais $q = n$ e $j = 0$. Se a árvore for vazia $n = \text{nil}$.

O algoritmo pára quando i for encontrado em um nó com endereço n , ou quando n for igual a nil . Neste caso, i não foi localizado e chegou-se a um nó folha da árvore. O endereço desta folha está em q , que juntamente com a posição de j poderá ser usado pelo algoritmo de inserção.

A saída do algoritmo é uma tripla ordenada $(n, \text{verdadeiro}, j)$, se i pertence à árvore, onde n é o endereço do nó que contém i e j a posição de i neste nó; ou (n, falso, j) , se i não pertence a árvore, onde q é o endereço do último nó folha visitado e j a posição em q na qual o valor de i poderia ser inserido. Como na linguagem Pascal, atribuiremos a tripla de saída a Busca para maior clareza.

No algoritmo um nó de endereço n é considerado contendo a forma $(p_0, i_1, p_1, \dots, i_j, p_j, i_{j+1}, p_{j+1}, \dots, i_m, p_m)$, $1 \leq m \leq 2K$. Suponhamos que neste algoritmo o termo $\exists j Q(j)$, onde Q é uma proposição, atribui a j o único valor que torna Q verdadeiro, quando este for o caso.

Busca(n, q, j, i):

$n = \text{nil}?$	Sim	Não	Não	Não	Não
$i < i_1?$	*	Sim	*	*	Não
$\exists j(i = i_j)?$	*	*	Sim	*	Não
$\exists j(i_j < i < i_{j+1})?$	*	*	*	Sim	Não
Busca:=Busca($p_0, n, 0, i$)		1			
Busca:=Busca(p_j, n, j, i)				1	
Busca:=Busca(p_m, n, m, i)					1
Busca:=($n, \text{verdadeiro}, j$)			1		
Busca:=(q, falso, j)	1				
termina Busca	2	2	2	2	2

Custo de Busca

Observa-se sem dificuldades que o custo da busca de um valor i é proporcional à altura h da árvore, pois no pior caso este pode pertencer a uma folha. Por sua vez, conforme demonstrado na seção 1.3, esta altura é, no pior caso, logarítmica no número de valores ($\log_{K+1} \frac{I+1}{2}$). A tabela 1.1, calculada por Comer [19], mostra como o pior caso pode ser razoável mesmo em grandes arquivos.

K—I	10^3	10^4	10^5	10^6	10^7
10	3	4	5	6	7
50	2	3	3	4	4
100	2	2	3	3	4
150	2	2	3	3	4

Tabela 1.1 - Número máximo de nós visitados na busca de um valor do índice, para um arquivo com I valores em uma *árvore-B* de ordem K .

Todavia, segundo Bayer e McCreight, na análise do custo deve-se levar em conta o número de páginas do arquivo de índices que são trazidas da memória secundária (disco) para a primária (central) e das que são gravadas de volta na memória secundária. Considera-se que cada operação elementar (busca, atualização, inserção e remoção), dado o endereço do registro de dados, envolve um único acesso a uma página do arquivo de índices do disco.

Supondo $Tmin$ e $Tmax$ o número mínimo e máximo respectivamente de páginas do arquivo de índices trazidas da memória secundária para primária, $Gmin$ e $Gmax$ o número mínimo e máximo das páginas gravadas de volta na memória secundária, o fato de que a raiz da árvore não está armazenada permanentemente na memória primária e adaptando a análise de Bayer e McCreight, junto com a definição vista de altura da árvore, tem-se que:

$$Tmin = 1 \text{ e } Tmax = h + 1$$

Como na busca não há modificação de nenhuma página

$$Gmin = Gmax = 0$$

1.4.2 Inserção

Em uma *árvore-B* de ordem K a inserção de um valor do índice é sempre realizada em um nó folha, constituindo um processo simples, que envolve dois passos: um para localizar o nó folha apropriado para a inclusão do novo elemento, ou seja, busca da posição de inserção (*fase 1*) e o outro para inserir o elemento e fazer o ajuste da árvore quando necessário (*fase 2*).

Portanto, ao se detectar o nó folha n , no qual o elemento deve ser inserido, examina-se este nó colocando no mesmo o novo elemento na posição apropriada. Se n tem espaço suficiente para receber o novo elemento, o processo de inserção termina; entretanto, no caso deste já conter $2K$ valores do índice, a inclusão de mais um outro valor proporcionará uma *sobrecarga* do nó (“overflow”). Neste caso, utiliza-se a técnica de *desdobrar* (“split”) o nó em torno do valor mediano para manter as características e propriedades da *árvore-B*, ou a técnica denominada de *overflow*³ (Bayer e McCreight), para evitar desdobramentos.

1. Desdobramento

Seja n um nó folha completo, no qual o novo elemento i deva ser incluído, e n_p o seu pai.

n : $p_0, i_1, p_1, i_2, p_2, \dots, i_{2K}, p_{2K}$ ($p_0 = p_1 = \dots = p_{2K} = \text{nil}$).

Após a inserção do novo elemento, renomeando-se os índices, n teria a seguinte configuração:

n : $p_0, i_1, p_1, i_2, p_2, \dots, i_{2K+1}, p_{2K+1}$.

Como n ficou sobrecarregado, o procedimento cria um novo nó n' e divide o conteúdo de n entre n e n' , de tal forma que n contenha os K menores elementos, n' os K maiores elementos e o valor mediano i_{K+1} (pois $\lceil \frac{2K+1}{2} \rceil = K + 1$) é inserido em n_p , juntamente com um apontador para n' , tornando n e n' irmãos. O apontador para n em n_p é trocado pela sequência $(p_n, i_{K+1}, p_{n'})$.

n : $p_0, i_1, p_1, i_2, p_2, \dots, i_K, p_K$ n' : $p_{K+1}, i_{K+2}, p_{K+2}, \dots, i_{2K+1}, p_{2K+1}$

Todavia, a inserção de $(i_{K+1}, p_{n'})$ em n_p pode tornar este nó sobrecarregado, acarretando uma divisão de n_p . No pior caso, esse processo propaga-se até a raiz causando

³O termo dado à técnica será mantido do inglês para não descaracterizá-lo.

o desdobramento da mesma e com isso o aumento da altura da *árvore-B*. A figura 1.3 ilustra o processo de desdobramento de um nó, pela inserção do valor 12 em uma *árvore-B* de ordem $K = 2$.

Figura 1.3: Exemplo do desdobramento de um nó n .

Em 1983, Küspert [47] descreveu uma variação da técnica de desdobramento, aplicada apenas no nível das folhas e denominada de *desdobramento marginal* (“marginal split”). Nesta técnica, quando o valor i a ser inserido no nó folha n sobrecarregado, consiste do menor ou maior valor dentre os demais existentes, isto é, n é o nó folha mais à esquerda ou mais à direita, a divisão é feita de forma especial. Deixa-se em n , $2K - 1$ valores do índice e coloca-se em um novo nó o valor i junto com o restante do conteúdo de n . O uso desse método durante inserções de elementos que estejam ordenados ascendente ou decendentemente, acarreta um aumento na ocupação do espaço no nível das folhas e garante um número constante de inserções até que seja necessário realizar um novo desdobramento (a cada $2K - 1$ operações de inserções).

2. Overflow

Considere n um nó completo, n' um nó com menos do que $2K$ valores do índice, tais que n e n' são irmãos adjacentes sendo n_p o pai dos mesmos. Seja i o valor a ser inserido em n . Como n' contém espaço, o método consiste em unir os elementos de n e n' , juntamente com o separador destes, localizado em n_p , e com o novo valor i . Em seguida, faz-se a distribuição desses valores entre n e n' , colocando em n_p o valor mediano, de tal forma que a quantidade de valores em n e n' difira no máximo de 1.

Dessa forma, há um aproveitamento de todo espaço existente, evitando o desdobramento de n , o qual só ocorrerá quando os irmãos adjacentes à esquerda e à direita deste contiverem $2K$ valores do índice cada um. Uma variação do método consiste em considerar uma vizinhança de nós, ao invés de apenas os irmãos adjacentes, aumentando ainda mais a utilização do espaço e diminuindo o crescimento da altura da árvore. A figura 1.4 ilustra um exemplo do uso da técnica de overflow pela inserção do valor 12.

Bayer e McCreight afirmam que, segundo resultados experimentais, o emprego deste método aumenta a ocupação do espaço, no pior caso, para 66%, quando são feitas apenas inserções, sem remoções. Entretanto, no caso de se fazer inserções e remoções, a ocupação mínima cai novamente para 50%. Notemos que este método tem a desvantagem de exigir a leitura e gravação de um nó adicional (n').

Figura 1.4: Exemplo do método de overflow.

Algoritmo de Inserção

O algoritmo tem como entradas o endereço n da raiz, que contém *nil* se a árvore é vazia, o valor i que se deseja inserir e o endereço d_i no disco, do registro que contém o valor i . O procedimento chama o algoritmo Busca para verificar a existência de i na árvore ou localizar o endereço do nó folha onde i deve ser inserido. Na tabela (B), a ação Desdobre(n) executa o desdobramento de uma página quando necessário. Os valores devolvidos por Busca são colocados na tripla ordenada $(n, existe, j)$ onde n contém o endereço do nó que contém i na posição j , quando *existe* é verdadeiro, ou n é o endereço do nó folha onde i será inserido, na posição entre j e $j + 1$, quando *existe* é falso.

Inserção(n, i, d_i):

A:

$n = \text{nil?}$	Sim	Não
crie um novo nó n	1	
insira $(\text{nil}, i, d_i, \text{nil})$ em n	2	
$(n, existe, j) := \text{Busca}(n, n, 0, i)$		1
vá para tabela B		2
termina Inserção	3	

B:

<i>existe</i> = verdadeiro?	Sim	Não	Não
<i>n</i> contém $2K$ valores?	*	Não	Sim
insira (i, d_i, nil) entre j e $j + 1$ em n		1	1
Desdobre(n)			2
termina Inserção	1	2	3

Custo de Inserção

O custo para incluir um novo elemento na *árvore-B* é medido pelo custo para localizar a posição de inserção (fase 1) acrescido do custo de propagação dos desdobramentos dos nós (fase 2), quando existirem. Como as inserções sempre são realizadas nas folhas, o tempo para a fase 1 é também proporcional à altura da árvore. Da mesma maneira, como as divisões dos nós podem se propagar até a raiz, no pior caso, o tempo gasto na fase 2 é também da ordem da altura da árvore. Portanto, o tempo total para inserção de um valor i é $O(h)$.

Por sua vez, através da análise feita por Bayer e McCreight e seguindo o que foi definido para análise do custo de busca, tem-se que o gasto com a inserção de um novo valor do índice, caso não haja necessidade de se fazer o desdobramento do nó, é dado por:

$$T_{min} = h + 1 \text{ e } G_{min} = 1$$

pois ocorreu alteração em um único nó.

O pior caso acontece quando todas as páginas do caminho da busca, inclusive a raiz, forem desdobradas. Nesse caso tem-se que:

$$T_{max} = h + 1 \text{ e } G_{max} = 2(h + 1) + 1 = 2h + 3$$

uma vez que todos os nós do percurso foram desdobrados e uma nova raiz foi criada.

Analisando o custo com inserções que fazem uso da técnica de overflow tem-se que:

$$T_{min} = h + 1 \text{ e } G_{min} = 1$$

$$T_{max} = 3(h + 1) - 2 = 3h + 1 \text{ e } G_{max} = 2(h + 1) + 1$$

Note-se que o pior caso (T_{max} e G_{max}) ocorre quando se tenta usar a técnica de overflow em todo o caminho de baixo para cima na árvore, examinando-se os irmãos de ambos os lados, exceto na raiz, mas a mesma não pode ser aplicada, sendo necessário executar desdobramentos.

Considerando uma árvore na qual os valores do índice nunca são removidos, Bayer e McCreight analisaram a quantidade de trabalho média para se construir uma *árvore-B* com I valores. Seja $N(I)$ o número de nós de uma *árvore-B* com I valores. Como cada divisão de um nó acarreta a criação de um outro, ou até mesmo de 2 novos quando ocorre o desdobramento da raiz, então a quantidade total de nós divididos na criação da árvore é limitada por $N(I) - 1$. Como todos os nós, exceto a raiz, têm no mínimo K valores e a raiz 1 valor, temos que $N(I) \leq 1 + \frac{I-1}{K}$. Assim, o número total de nós desdobrados na construção da árvore é: $N(I) - 1 \leq \frac{I-1}{K}$.

Note-se que os desdobramentos não requerem a leitura adicional de nenhuma página e dessa forma, o número médio de páginas transferidas na inserção de um único valor na árvore é igual aos números mínimo e máximo de páginas transferidas ($T_{med} = h + 1$). Por outro lado, cada desdobramento de um nó ocasiona a gravação adicional de 2 nós na memória secundária. Portanto, o número médio de nós gravados (G_{med}) em cada inserção de um valor de índice é dado por:

$$G_{med} = 1 + \frac{2}{I}(N(I) - 1) \leq 1 + \frac{2}{I}\left(\frac{I-1}{K}\right) = 1 + \frac{2}{K} - \frac{2}{IK}$$

$$G_{med} < 1 + \frac{2}{K}$$

1.4.3 Remoção

O processo de remoção de um valor i do índice em uma *árvore-B* também envolve 2 passos como na inserção, isto é, a operação de localizar o elemento a ser removido (fase 1) e a de reestruturar a árvore (fase 2), quando o número de valores do índice do nó que sofreu remoção for menor do que K ("underflow"). Todavia, deve-se levar em consideração 2 situações distintas: o valor i está 1) em um nó folha; 2) em um nó não folha.

No caso 1 a remoção é trivial. No caso 2, o procedimento deve procurar o valor i' que é a) o sucessor imediato de i , isto é, o valor menor dentre os maiores do que i ; b) ou simetricamente o predecessor imediato de i , isto é, o maior valor dentre os menores do que i . Suponhamos que i esteja na posição j de seu nó n . Pela construção das *árvores-B*, e no caso (a) i' estará em uma certa folha n_f que deve ser encontrada percorrendo-se a árvore a partir de p_{j+1} , atingindo-se o nó filho de n , e daí seguir pelos ponteiros p_0 desde este filho até alcançar-se n_f . O valor i será o menor valor de n_f . No caso (b), o percurso é simétrico,

partindo-se do filho apontado por p_j , seguindo-se agora pelos ponteiros p_m até atingir-se n_f . Encontrado i' , este deve ser colocado no lugar de i no nó n , e consequentemente retirado de n_f ; os outros nós não mudam. Após a remoção do valor, tanto pelo caso 1 como pelo caso 2, o procedimento deve verificar o número de valores do nó folha para detectar uma possível subcarga do mesmo.

Bayer e McCreight propuseram 2 métodos especificados adiante, para o caso de ocorrer subcarga. Para melhor compreensão, considere n e n' dois nós irmãos adjacentes, ou seja, têm o mesmo pai n_p e são apontados por ponteiros adjacentes em n_p .

1. Redistribuição ou Rotação

Este método é utilizado sempre que, após a remoção de um valor do índice em n ou n' um deles ficou com menos do que K valores do índice e a soma do número de valores em n e n' é maior ou igual a $2K$. O procedimento executa uma redistribuição entre os elementos de n e n' de maneira análoga à técnica de overflow (v. 1.4.2). Observe-se que essa técnica não acarreta propagação na árvore, pois o número de elementos em n_p não é alterado, apenas ocorre uma mudança no valor de algum de seus valores de índices. A figura 1.5 ilustra esse procedimento, após a remoção do valor 25 ($2K = 4$).

Figura 1.5: Exemplo do método de redistribuição em uma *árvore-B* de ordem $K=2$.

2. Concatenação

O método de concatenação consiste no processo inverso de *desdobramento de um nó* sendo utilizado toda vez que, após a remoção de algum valor em n ou n' a soma do número de valores do índice em n e n' se torne menor do que $2K$, impedindo a redistribuição. Seja i' o valor do índice em n_p separador de n e n' . O procedimento consiste em colocar-se em um destes - por exemplo, n - tanto i' como todos os valores

de n e n' . Em seguida, são removidos i' de n_p e o nó n' da árvore, conforme retratado na figura 1.6, após a remoção do valor 25. Entretanto, a remoção de um valor em n_p pode torná-lo subcarregado forçando uma repetição da aplicação do processo em n_p . No pior caso, isso pode propagar-se em toda a árvore, acarretando a remoção do nó raiz e consequentemente uma diminuição da altura da árvore.

Figura 1.6: Exemplo do método de concatenação em uma *árvore-B* de ordem $K=2$.

Algoritmo de Remoção

O algoritmo de remoção tem como entradas o endereço n da raiz, que contém *nil* se a árvore é vazia e o valor i que se deseja excluir. O procedimento chama o algoritmo Busca que localiza o endereço do nó n que contém i e a posição j de i dentro de n , ou verifica a não existência de i na árvore. Esses valores são devolvidos na tripla ordenada $(n, existe, j)$, como no caso do algoritmo Inserção. Na tabela Concatenação/Redistribuição(n) realiza estas rotinas quando se fizer necessário. Procure folha(i', n_f, j') percorre a árvore para localizar o nó folha de endereço n_f que contém o sucessor ou predecessor imediato i' de i , na posição j' de n_f .

Remoção(n, i):

A:

$n = \text{nil}?$	Sim	Não
$(n, existe, j) := \text{Busca}(n, n, j, 0)$		1
vá para tabela B		2
termina Remoção	1	

B:

$existe = verdadeiro?$	Não	Sim	Sim	Sim
n é folha?	*	Sim	Sim	Não
n contém K valores?	*	Sim	Não	*
remove (i, p_j, d_i)		1	1	
Concatenação/ Redistribuição(n)		2		
Procure folha(i', n_f, j')				1
troque i por i' em n				2
remova $(i', p_{j'}, d_{i'})$				3
vá para tabela C				4
termina Remoção	1	3	2	

C:

n_f contém menos do que K valores do índice?	Sim	Não
Concatenação/ Redistribuição(n_f)	1	
termina Remoção	2	1

Custo de Remoção

Da mesma maneira que no processo de inserção, observa-se que o custo para remover um elemento da *árvore-B* é também proporcional à altura h da árvore ($O(h)$), uma vez que há necessidade da localização do elemento a ser excluído e da possível reestruturação da árvore. Baseando-se na análise de Bayer e McCreight e nas definições introduzidas na análise do custo de busca, tem-se que quando a remoção é realizada com sucesso, isto é, o elemento a ser eliminado está na árvore, pertence a um nó folha e não é necessário fazer-se nenhuma reestruturação, o custo é:

$$T_{min} = h + 1 \text{ e } G_{min} = 1$$

Porém, se o elemento a ser removido não está em um nó folha, é necessário localizar seu sucessor ou predecessor imediato. Como nesse caso ocorre a modificação de 2 nós (o que contém o elemento a ser removido e o nó folha que possui o sucessor ou predecessor imediato), então

$$T = h + 1 \text{ e } G = 2$$

Por outro lado, no pior caso a operação de remoção ocasiona a concatenação de pares de nós vizinhos da árvore até o nível dos nós que são “netos” da raiz, e uma redistribuição nos valores de 2 filhos da raiz com uma consequente modificação desta, (isto é, de 3 nós). O custo é dado então por:

$$T_{max} = 2h + 1 \text{ e } G_{max} = (h - 1) + 3 = h + 2$$

Como o caso acima é possível, mas muito difícil de acontecer, vamos analisar a quantidade média de trabalho necessária para se remover um único valor do índice. Considere o processo executado em uma *árvore-B* com $N(I)$ nós, em que todos os valores do índice são removidos, mas nenhum é inserido.

Observa-se que quando a remoção não provoca a subcarga de um nó, então o número de páginas transferidas é dado por $T = h + 1$.

No caso de um nó folha tornar-se subcarregado é necessário trazer uma página a mais da memória secundária para a primária ($T' = 1$). Se esse problema for solucionado com o método de redistribuição, não há transferência de nenhuma outra página; se for aplicada a técnica de concatenação e esta se propagar pela árvore, então serão trazidas para a memória primária uma quantidade extra de páginas. Essa quantidade é igual ao número de concatenações propagadas. Como ocorrem no máximo $N(I) - 1$ concatenações em toda a árvore e o número médio é $\frac{N(I)-1}{I}$, então o número médio de páginas transferidas em cada remoção que ocasiona a propagação da técnica de concatenação na árvore é:

$$T'' = \frac{N(I) - 1}{I} = \frac{1}{I} \left(\frac{I - 1}{K} \right) < \frac{1}{K}$$

Assim, o número médio de páginas transferidas na remoção de um único valor de índice é limitado por:

$$T_{med} < T + T' + T'' < h + 1 + 1 + \frac{1}{K} = h + 2 + \frac{1}{K}$$

Quando a remoção é realizada sem ocasionar subcarga em um nó, o número de páginas gravadas é, no pior caso, $G = 2$. Se ocorrer a subcarga de um nó e for utilizada a técnica de concatenação, então como cada concatenação ocasiona a modificação em 2 páginas, temos que o número médio de páginas gravadas por cada concatenação é $G' < \frac{2}{K}$. Por outro lado, como a aplicação do método de redistribuição requer a gravação de 2 páginas

adicionais na memória do disco ($G'' = 2$), então o número médio de páginas gravadas na remoção de um único valor é limitado por:

$$G_{med} < 2 + 2 + \frac{2}{K} = 4 + \frac{2}{K}$$

1.5 Escolha da Ordem

Como já foi visto, os custos das rotinas de busca, inserção e remoção dependem da altura da *árvore-B* e esta, por sua vez, da ordem K , pois $h = O(\log_K I)$.

Bayer e McCreight [9] analisaram a influência de K no tempo de transferência de uma página entre a memória primária e secundária e na busca binária realizada com os valores do índice de um nó. Nessa análise, eles propuseram uma fórmula que possibilita a determinação de um valor K para minimizar o tempo total.

Para os atuais sistemas não faz sentido utilizar a idéia proposta em [9], uma vez que o cálculo de K é feito pelo sistema de *árvore-B*, de tal maneira a preencher toda a página, cujo tamanho é fixado pelo sistema operacional (eventualmente pelo usuário).

1.6 Fator de Ocupação de Espaço

O fator de ocupação de espaço ($\mathcal{U}$) de uma *árvore-B* foi definido por Quitzow e Klopprogge [79] como sendo o quociente entre o número de valores do índice armazenados na árvore e a capacidade total de armazenamento dos nós existentes na árvore, isto é:

$$\mathcal{U} = \frac{I}{2KN} \quad (1.1)$$

onde:

K = ordem da árvore;

I = número total dos valores do índice na árvore;

N = número total de nós na árvore;

Pelas características da *árvore-B*, observa-se que o fator de ocupação mínima de espaço é de 50%, uma vez que cada nó (exceto a raiz) contém entre K e $2K$ valores do índice.

Vários autores [3, 4, 18, 26, 33, 47, 54, 72, 79, 97, 98] estudaram e desenvolveram

modelos para determinar o fator de ocupação médio das *árvores-B*, construídas através de inserções aleatórias, chegando ao resultado de que esse fator é igual a $\ln 2 \cong 69,31\%$.

Nakamura e Mizoguchi (1978) [72], Küspert (1983) [47], Wright (1985) [97], Chu e Knot (1989) [18], demonstraram o valor médio do fator de ocupação fazendo uso de equações de recorrências, que expressam o número de nós contendo m ($K \leq m \leq 2K$) valores do índice. Vamos mostrar esse valor médio com base na demonstração de Nakamura e Mizoguchi.

Considere um *diagrama de transição probabilística* composto por um conjunto de vértices V , rotulados por v_j , $K \leq j \leq 2K$, e por um conjunto de arestas dirigidas A , com as seguintes características:

1. Cada $v_j \in V$ representa os nós da *árvore-B* com j valores do índice.
2. Cada aresta $a_l = (v_x, v_y) \in A$, $K \leq x, y \leq 2K$, representa a transição do número de nós da árvore com x valores para y valores do índice, onde $|x - y| = 1$.
3. $\forall v_j \in V$, existe um número L_j associado cujo valor é a quantidade de nós contendo j valores do índice.
4. $\forall a_l = (v_x, v_y)$, existe um número Q_l associado, tal que Q_l é a probabilidade de diminuir L_x e aumentar L_y , após uma inserção.
5. Nas inserções aleatórias, a probabilidade de um novo elemento ser incluído em um determinado nó é proporcional ao número de valores existentes nesse nó. Assim, para Q_l associado a $a_l = (v_x, v_y)$, tem-se que $Q_l = \frac{x}{I} L_x$.
6. A inserção realizada em um nó com r valores ($K \leq r < 2K$), acarreta um aumento em L_{r+1} de uma unidade e uma diminuição em L_r de uma unidade.
7. A inserção realizada em um nó com $2K$ valores, acarreta uma diminuição em L_{2K} de uma unidade e um aumento de 1 unidade tanto em L_K como em L_{K+1} , em virtude do desdobramento do nó.

Considere uma *árvore-B* de ordem K com $(I - 1)$ valores do índice. Após a I -ésima operação de inserção, temos que o número de nós com j elementos (L_j^I), $K + 1 < j \leq 2K$ é igual ao número de nós com j elementos antes da inserção (L_j^{I-1}), acrescido da probabilidade da inserção ocorrer em um nó com $(j - 1)$ elementos e diminuído da probabilidade da inserção se realizar em um nó com j elementos. Assim,

$$L_j^I = L_j^{I-1} + \frac{(j-1)}{(I-1)} L_{j-1}^{I-1} - \frac{j}{(I-1)} L_j^{I-1}$$

Para $j = K$ e $j = K + 1$ tem-se que:

$$L_K^I = L_K^{I-1} + \frac{(2K)}{(I-1)} L_{2K}^{I-1} - \frac{K}{(I-1)} L_K^{I-1}$$

$$L_{K+1}^I = L_{K+1}^{I-1} + \frac{(2K)}{(I-1)} L_{2K}^{I-1} + \frac{K}{(I-1)} L_K^{I-1} - \frac{K+1}{(I-1)} L_K^{I-1}$$

Para montar as equações de recorrência em função dos L_j 's, basta observar o diagrama e adicionar os valores Q associados às arestas que entram em um vértice v_j e subtrair os valores associados às arestas que saem de v_j .

Portanto,

$$\begin{aligned} L_K^I &= L_K^{I-1} + \frac{(2K)}{(I-1)} L_{2K}^{I-1} - \frac{K}{I-1} L_K^{I-1} \\ L_{K+1}^I &= L_{K+1}^{I-1} + \frac{K}{(I-1)} L_K^{I-1} - \frac{(K+1)}{(I-1)} L_{K+1}^{I-1} + \frac{(2K)}{(I-1)} L_{2K}^{I-1} \\ L_{K+2}^I &= L_{K+2}^{I-1} + \frac{(K+1)}{(I-1)} L_{K+1}^{I-1} - \frac{(K+2)}{(I-1)} L_{K+2}^{I-1} \\ &\vdots \\ L_{2K}^I &= L_{2K}^{I-1} + \frac{(2K-1)}{(I-1)} L_{2K-1}^{I-1} - \frac{(2K)}{(I-1)} L_{2K}^{I-1} \end{aligned} \quad (1.2)$$

Note-se que:

$$N = \sum_{j=0}^K L_{K+j}^I \quad (1.3)$$

Fazendo-se a somatória das equações (1.2) tem-se que:

$$\sum_{j=0}^K L_{K+j}^I = \sum_{j=0}^K L_{K+j}^{I-1} + \frac{2K}{(I-1)} L_{2K}^{I-1} \quad (1.4)$$

Por outro lado,

$$I = \sum_{j=0}^K (K+j) L_{K+j}^I \quad (1.5)$$

Fazendo-se $L_j^I = Il_j$ e aplicando em (1.3), (1.4), (1.5) e nas de equações (1.2), tem-se:

$$N = I \sum_{j=0}^K l_{K+j} = (I-1) \sum_{j=0}^K l_{K+j} + 2K l_{2K} \Rightarrow \sum_{j=0}^K l_{K+j} = 2K l_{2K} \quad (1.6)$$

$$\sum_{j=0}^K (K+j) l_{K+j} = 1 \quad (1.7)$$

$$\begin{aligned} (2K+1)l_{2K} &= (2K-1)l_{2K-1} \\ (2K)l_{2K-1} &= (2K-2)l_{2K-2} \\ &\vdots \\ (K+2)l_{K+1} &= (K)l_K + 2K l_{2K} \end{aligned} \quad (1.8)$$

Como $\mathcal{U} = \frac{I}{2KN}$ e, por (1.6) tem-se que $\frac{I}{N} = \frac{1}{2K l_{2K}}$, obtem-se:

$$\mathcal{U} = \frac{1}{(2K)^2 l_{2K}} \quad (1.9)$$

Somando-se as equações (1.8) entre si:

$$(2K+1)l_{2K} + (2K)l_{2K-1} + (2K-1)l_{2K-2} + \cdots + (K+2)l_{K+1} = \sum_{j=0}^K (K+j)l_{K+j} = 1$$

Colocando-se todos os termos do lado esquerdo da equação acima em função de l_{2K} deriva-se:

$$(2K+1)l_{2K} + 2K \frac{(2K+1)}{(2K-1)} l_{2K} + 2K \frac{(2K+1)}{(2K-2)} l_{2K} + \cdots + 2K \frac{(2K+1)}{(K+1)} l_{2K} = 1$$

$$2K(2K+1)l_{2K} \left(\frac{1}{2K} + \frac{1}{(2K-1)} + \frac{1}{(2K-2)} + \cdots + \frac{1}{(K+1)} \right) = 1$$

Portanto,
$$2K l_{2K} = \frac{1}{(2K+1) \sum_{j=K+1}^{2K} \frac{1}{j}}$$

Colocando-se este resultado em (1.9):

$$\mathcal{U} = \frac{1}{(2K) \frac{1}{(2K+1) \sum_{j=K+1}^{2K} \frac{1}{j}}} = \frac{1}{2K} (2K+1) \sum_{j=K+1}^{2K} \frac{1}{j}$$

Para grandes valores de $2K$,

$$\lim_{2K \rightarrow \infty} \mathcal{U} = \lim_{2K \rightarrow \infty} \frac{2K+1}{2K} \int_{K+1}^{2K} \frac{d_j}{j} = \ln(2) \cong 69,31\%$$

Em 1984, Leung [54] publicou um método bastante simples para determinar o fator de ocupação médio das *árvores-B*. Ele afirma que para um número fixo I de valores do índice, o fator de ocupação esperado é:

$$E(\mathcal{U}) = \frac{I}{2K} E\left(\frac{1}{N}\right)$$

Considerando f ($0 < f < 1$) um fator arbitrário onde todo nó não raiz está pelo menos $100\%f$ preenchido (a *árvore-B* é um caso especial em que $f = 50\%$), temos que os números mínimo e máximo de nós na árvore são: $N_{min} = \frac{I}{2K}$ e $N_{max} = \frac{I}{2Kf}$, respectivamente.

Leung aproximou N pela distribuição retangular contínua sobre o intervalo $[\frac{I}{2K}, \frac{I}{2Kf}]$ e obteve:

$$E\left(\frac{1}{N}\right) = \frac{1}{N_{max} - N_{min}} \int_{N_{min}}^{N_{max}} \frac{d_N}{N} = \frac{2Kf}{I(1-f)} \ln \frac{1}{f}$$

Assim,

$$E(\mathcal{U}) = \frac{I}{2K} \frac{2Kf}{I(1-f)} \ln \frac{1}{f} = \frac{f}{(1-f)} \ln \frac{1}{f}$$

Para $f = 50\%$, $E(\mathcal{U}) = \ln 2$. Chu e Knott encontraram a mesma fórmula fazendo uso de equações de recorrência.

Gupta e Srinivasan (1986) [33] apontam duas falhas na demonstração de Leung: 1) o número de nós não é uma variável contínua conforme assumido e 2) não se justifica a

utilização de uma distribuição retangular para N . A falha (1) é solucionada fazendo-se uma modificação na derivação:

$$E\left(\frac{1}{N}\right) = \frac{1}{N_{max} - N_{min} + 1} \sum_{N_j=N_{min}}^{N_{max}} \frac{1}{N_j}$$

Já a (2) é resolvida utilizando-se uma expressão para expressar o número de diferentes configurações possíveis em uma *árvore-B* de ordem K , com N nós, construída pela inserção de I elementos.

Em [3, 18, 26, 79], foram feitas algumas derivações do fator de ocupação médio quando se usa a técnica de overflow. Küspert encontrou a seguinte fórmula:

$$\mathcal{U}(d') = d' \ln\left(\frac{d' + 1}{d'}\right)$$

onde $(d' - 1)$ corresponde ao número de nós adjacentes visitados durante o uso do método. Nos casos em que a técnica de overflow não é empregada $d' = 1$ e $\mathcal{U}(1) = 1 \ln(2) = \ln(2)$.

Yao (1978) [98], Eisenbarth et al. (1982) [26] e Baeza-Yates [3, 4] (1989) analisaram o fator de ocupação de espaço médio das *árvores-B* e algumas de suas variações, que utilizam ou não a técnica de overflow, através de um método denominado de *fringe analysis*⁴. Este método baseia-se em uma forma de descrever a composição de um *fringe*.

Seja C uma coleção finita de árvores de vários tipos. Denomina-se *fringe de uma árvore* ao conjunto disjunto de uma ou mais sub-árvores isomórficas aos elementos de C . Dessa forma, a composição de um *fringe* pode ser descrita de várias maneiras. Yao a descreve em função do número esperado dos vários tipos de árvore contidas no *fringe*. Eisenbarth et al. definem a composição como a probabilidade de encontrar uma folha pertencente a um determinado tipo de árvore, isto é, a probabilidade da inserção do I -ésimo elemento ocorrer em um nó com m valores do índice ($K \leq m \leq 2K$). Esta probabilidade é escrita através de uma matriz de recorrência, a qual converge para a solução de um sistema linear.

O método caracteriza-se por analisar os nós do nível das folhas, seguido de uma análise para o nível dos pais das folhas e daí fazendo-se estimativas para os demais níveis, uma vez que os níveis mais baixos de uma árvore contêm uma grande quantidade de valores. O método foi também utilizado para determinar outras estimativas como: número de desdobramentos por inserção, número de nós e altura da árvore, entre outras.

⁴A denominação do método foi dada por Eisenbarth et al. O método foi também utilizado para análise das *árvores-2-3* (v. 2.1).

Capítulo 2

VARIAÇÕES DA ÁRVORE-B

Conforme visto no capítulo 1, o custo das operações na *árvore-B* depende da altura da mesma, a qual pode ser reduzida aumentando-se o número de elementos armazenados em um nó. Com o intuito de aumentar a ocupação da memória e/ou diminuir o custo das operações na *árvore-B*, foram propostas inúmeras variações da mesma, quebrando-se um pouco as regras pré-estabelecidas, que serão apresentadas obedecendo a ordem cronológica de aparecimento na literatura.

2.1 Árvore-2-3

Introduzida por John Hopcroft em 1970 (não publicado, referido por [19, 44]), a *árvore-2-3* consiste em uma classe das árvores balanceadas, a qual pode ser considerada como precursora das *árvores-B*, ou seja, a *árvore-B* é um caso generalizado da *árvore-2-3*. A estrutura é definida como uma árvore onde cada nó não folha tem 2 ou 3 filhos e todas as folhas têm a mesma profundidade, isto é, a *árvore-2-3* é uma *árvore-B* de ordem $K = 1$, como ilustrado na figura 2.1. Em função do pequeno tamanho dos nós Comer [19] afirma que a *árvore-2-3* é impraticável para armazenamento na memória secundária, mas bastante apropriada quando utilizada como uma estrutura de dados interna.

Considere $\mathcal{A}^{2-3}$ uma *árvore-2-3* de altura $h^{2-3} \geq 0$. Seja N^{2-3} o número total de nós de $\mathcal{A}^{2-3}$ e N_f^{2-3} o número de nós folhas. Como cada nó n de $\mathcal{A}^{2-3}$ tem no mínimo 2 filhos, então existe pelo menos 1 nó no nível 0, 2 nós no nível 1, 4 nós no nível 2, e assim

sucessivamente até o nível das folhas, o qual contém no mínimo $2^{h^{2-3}}$ nós. Portanto,

$$N^{2-3} \geq \sum_{j=0}^{h^{2-3}} 2^j = 2^{h^{2-3}+1} - 1$$

Por outro lado, cada nó n de $\mathcal{A}^{2-3}$ tem no máximo 3 filhos e dessa forma existe, no caso extremo, 1 nó no nível 0, 3 nós no nível 1, 9 nós no nível 2 e $3^{h^{2-3}}$ nós folhas.

$$N^{2-3} \leq \sum_{j=0}^{h^{2-3}} 3^j = \frac{3^{h^{2-3}+1} - 1}{2}$$

Logo, para $h^{2-3} \geq 0$

$$2^{h^{2-3}+1} - 1 \leq N^{2-3} \leq \frac{3^{h^{2-3}+1} - 1}{2}$$

$$2^{h^{2-3}} \leq N_f^{2-3} \leq 3^{h^{2-3}}$$

Figura 2.1: Exemplo de uma *árvore-2-3* de altura $h^{2-3} = 2$.

2.2 Árvore-BB (Binary B-Tree)

Em 1971, Bayer [7] propôs uma representação binária para a *árvore-2-3* de Hopcroft, isto é, uma *binarização* desta estrutura, a qual pode ser utilizada em armazenamentos na memória primária. Estas árvores surgiram com o intuito de evitar gastos de espaços para os nós que contêm apenas metade de sua capacidade (um único valor de índice). Para tal, utiliza-se uma *alocação ligada*, na qual os nós são representados como mostrado na figura 2.2 representando (a) um nó ternário e (b) um nó binário.

Figura 2.2: Representação dos nós em uma *árvore-2-3* e a correspondência em uma *árvore-BB*.

Observa-se que os nós da *árvore-2-3* perdem sua identidade, transformando a mesma em uma árvore binária, com todos os nós folhas no mesmo nível. Entretanto, é necessário fazer distinção entre os ponteiros que referenciam os nós descendentes (verticais) e os que apontam para os nós irmãos (horizontais), ou seja, para os elementos que estariam em uma mesma página na *árvore-2-3*. Esta diferenciação é facilmente realizada através do uso de um bit extra no ponteiro da direita. A figura 2.3 ilustra o exemplo da *árvore-BB* para o exemplo da *árvore-2-3* da figura 2.1.

Os algoritmos de inserção e remoção para as *árvores-BB* são mais complicados do que os da *árvore-B*, exigindo cuidados especiais para evitar a situação de existência de 2 ponteiros sucessivos à direita, como mostrado abaixo.

Note-se que a *árvore-BB* apresenta os custos para as operações de busca, inserção e remoção de ordem logarítmica no número de nós ($O(\log_2 N^{BB})$, onde N^{BB} representa o número de nós da *árvore-BB*). O uso do ponteiro da direita para o nó irmão não pode ser ignorado. No pior caso, há uma duplicação do número de nós visitados em uma consulta à árvore, porém continua sendo de ordem logarítmica.

Figura 2.3: Representação de uma *árvore-BB*.

Em 1972, Bayer [8] apresentou uma extensão da *árvore-BB*, a qual permite ponteiros de ligação para direita e para esquerda, denominada de *árvore-BBS* ("Symetric Binary B-Tree"), como na figura 2.4. As *árvores-BBS* podem ser vistas como uma representação binária das *árvores-2-3-4* (todos os nós internos têm entre 2 e 4 filhos). Essa estrutura apresenta 2 bits extras para distinguir a natureza de cada ponteiro e requer algoritmos de inserção e remoção mais complicados do que as *árvores-BB*. Os algoritmos são executados de tal maneira a garantir que em qualquer caminho percorrido por alguma operação de busca, não existam 2 ponteiros horizontais sucessivos e todos os caminhos tenham o mesmo número de ponteiros verticais. Com isso, o custo das operações de busca, inserção e remoção continua sendo igual à *árvore-BB*.

Em 1978, Guibas e Sedgewick [32] propuseram uma técnica de coloração para manter as *árvores-BB* e *árvores-BBS* balanceadas. Nesta técnica os ponteiros verticais e horizontais recebem marcações correspondentes a duas cores distintas. O balanceamento é conseguido através de rotações em torno dos nós das árvores, afim de preservar determinadas propriedades, como nas árvores AVL [35].

Figura 2.4: Exemplo de uma *árvore-BBS*.

2.3 Árvore- B^O

Definida em 1973 por Knuth [44]¹, esta variação da *árvore-B* tem como principal característica o fato de todos os nós, exceto a raiz, possuírem pelo menos $\frac{2}{3}$ (66%) de seu espaço preenchido ao invés de $\frac{1}{2}$ (50%) como as *árvores-B*. Para conseguir este efeito, evita-se o desdobramento dos nós, através da utilização da técnica de overflow entre os nós vizinhos. Quando 2 nós adjacentes estiverem um sobrecarregado e o outro completo, a união do conteúdo destes é dividida em 3 nós, cada um contendo aproximadamente $\frac{2}{3}$ dos elementos, isto é, os 3 nós terão respectivamente $\lfloor \frac{4K}{3} \rfloor$, $\lfloor \frac{4K+1}{3} \rfloor$, $\lfloor \frac{4K+2}{3} \rfloor$ valores cada um, e dois valores serão inseridos no nó pai. Denominaremos essa técnica de *desdobramento de 2 para 3*; este termo será usado no capítulo 6.

Comparando com a *árvore-B*, essa versão possibilita um aumento na ocupação de espaço, sendo de pelo menos 66% em lugar de 50%, e uma diminuição na altura da árvore. Com base nos cálculos realizados no capítulo 1 tem-se que o número mínimo de valores (I^O_{min}) da *árvore-B^O* é dado por:

$$I^O_{min} = 2\left(\frac{2K}{3} + 1\right)^{h^O} - 1$$

onde h^O é a altura da *árvore-B^O* e facilmente calculada. Seja I^O o número de valores da *árvore-B^O*. Claramente,

$$I^O \geq 2\left(\frac{2K}{3} + 1\right)^{h^O} - 1$$

¹Esta estrutura foi denominada por Knuth de *árvore-B**.

$$\log_2 2 + \log_2 \left(\frac{2K}{3} + 1 \right)^{h^O} \leq \log_2 (I^O + 1)$$

$$h^O \leq \log_{(\frac{2K}{3}+1)} \left(\frac{I^O + 1}{2} \right)$$

Quanto ao fator de ocupação de espaço, Leung [54] afirma que em média este atinge 81%, o qual pode ser verificado com base na fórmula determinada por ele e por Chu e Knott [18] (v. 1.6), usando o fator $f = \frac{2}{3}$.

$$E(\mathcal{U}) \cong \frac{f}{1-f} \ln\left(\frac{1}{f}\right) = \frac{\frac{2}{3}}{1-\frac{2}{3}} \ln\left(\frac{1}{\frac{2}{3}}\right) = 2 \ln\left(\frac{3}{2}\right) \cong 81\%$$

A figura 2.5 mostra um gráfico comparativo da ocupação média de memória da *árvore-B* e *árvore- B^O* .

Figura 2.5: Gráfico comparativo do fator de ocupação médio de espaço.

2.4 Árvore- B^*

Objetivando diminuir a altura da *árvore-B* através da maximização da quantidade de elementos armazenados nos nós, Knuth [44] sugeriu em 1973 uma variação da *árvore-B* que denominaremos de *árvore- B^{*2}* . Esta estrutura caracteriza-se por conter todos

²Na realidade, Knuth não atribuiu nenhum nome a esta nova versão da *árvore-B* e chamou de *árvore- B^** a variação que denominamos de *árvore- B^O* . Por outro lado, Comer [19], Korth [45], Boswell e Tharp [15] classificaram a *árvore- B^** de Knuth de *árvore- B^+* .

os registros de dados nas folhas da árvore, e somente nestas. Elas são conectadas por ponteiros formando uma lista ligada, a qual denominaremos de *região de dados*. Os nós não folhas contêm apenas valores do índice e os ponteiros para os nós filhos (registros de índice), formando o que denominaremos de *região de índice*. A figura 2.6 mostra a separação lógica das regiões que compõem a estrutura.

Figura 2.6: Estrutura da *árvore-B**.

Os nós localizados na região de índice e os nós folhas podem ter formatos e tamanhos diferentes. A capacidade mínima de um nó folha é de K registros de dados e a de um nó não folha é de K^* pares de ponteiros e valores do índice. Wedekind [94] define estes nós conforme a figura 2.7, onde (a) representa um nó não folha da *árvore-B** com m^* valores do índice $i_1, i_2, \dots, i_{m^*}$ e ponteiros para os nós filhos $p_0, p_1, \dots, p_{m^*}$ ($K^* \leq m^* \leq 2K^*$), e (b) um nó folha da *árvore-B** com m registros de dados $d_1, d_2, \dots, d_m$, associados a m valores do índice $i_1, i_2, \dots, i_m$, ($K \leq m \leq 2K$) e uma marca M para distinguir os nós folhas dos nós não folhas. Na realidade, os valores do índice fazem parte do registro de dados, no entanto foram representados separadamente para maior clareza. cremos que esta marca M é totalmente dispensável, pois a altura h da árvore pode ser utilizada para descobrir se foi atingido um nó folha. Na figura 2.8 há um exemplo numérico da *árvore-B**.

Figura 2.7: Representação dos nós da *árvore-B**.

Figura 2.8: Exemplo numérico de uma *árvore- B^** ($h = 1$, $K = 1$ e $K^* = 2$).

Seja n um nó não folha da *árvore- B^** , $n(p_j)$ o nó descendente de n referenciado por p_j e $I^*(p_j)$ o conjunto dos valores do índice contidos nos nós da sub-árvore para a qual $n(p_j)$ funciona como raiz. Dessa forma, uma das condições abaixo sempre ocorre:

1. $\forall i \in I^*(p_0), i \leq i_1$
2. $\forall i \in I^*(p_j), i_j < i \leq i_{j+1} \quad j = 1, 2, \dots, m^* - 1$
3. $\forall i \in I^*(p_{m^*}), i > i_{m^*}$

De acordo com Wedekind, os 2 tipos de nós têm a mesma capacidade em bytes (tamanho da página do disco) que deve satisfazer:

$$L = (\lambda_i + \lambda_d)2K + \lambda_M = \lambda_p + 2K^*(\lambda_i + \lambda_p)$$

onde

λ_i = tamanho do campo dos valores do índice;

λ_d = tamanho do registro de dados, excluindo-se o campo do valor do índice;

λ_M = tamanho da marca M ;

λ_p = tamanho do campo dos ponteiros;

Em função de sua estrutura, a *árvore- B^** permite processamento aleatório e sequencial. Este último é realizado através da lista ligada formada pelos nós folhas, eventualmente após a localização de um registro com um determinado valor do índice.

A operação de busca é análoga às *árvores- B* , com a diferença de que ao se encontrar o valor i sendo procurado, em um nó da região de índice, o procedimento não pára, segue pelo ponteiro imediatamente à esquerda desse valor até atingir o nó folha apropriado. Desse modo, qualquer operação de busca aleatória de um valor do índice necessita percorrer

todo o caminho que vai da raiz até uma folha. Nesse sentido, essa estrutura apresenta uma pequena desvantagem quando comparada com a *árvore-B*.

Já o procedimento de inserção de um novo valor i é similar ao da *árvore-B*, diferindo apenas no caso em que o nó folha n , onde deve ser feita a inserção, encontra-se sobre-carregado. Nesta situação n também é desdobrado e o valor do índice mediano de n é mantido na metade esquerda, além de ser incluído no nó pai.

Por sua vez, o fato de todos os registros localizarem-se nas folhas torna simples o processo de remoção. Comer [19] propõe que a remoção de um registro de dados não provoque a eliminação do valor do índice correspondente, se este ocorrer na região de índice. Quando essa remoção provocar a subcarga de um nó folha, os procedimentos de redistribuição e concatenação podem propiciar modificações na região de índice.

Do ponto de vista do desempenho, a *árvore-B** é considerada melhor do que a *árvore-B*, fato que pode ser verificado através da determinação da altura da árvore. A seguir deduzimos a altura de uma *árvore-B**, cuja fórmula foi apresentada por Wedekind. Para tal, considere $\mathcal{A}^*$ uma *árvore-B** de altura $h^* \geq 1$, de ordem K^* para os nós não folhas e K para os nós folhas. Seja N^*min e N^*max , os números mínimo e máximo, respectivamente, de nós da árvore $\mathcal{A}^*$, onde:

N^*min_{int} = número mínimo de nós internos sem contar a raiz;

N^*max_{int} = número máximo de nós internos sem contar a raiz;

N^*min_f = número mínimo de nós folhas;

N^*max_f = número máximo de nós folhas;

Seguindo as ideias apresentadas no capítulo 1, seção 1.3 tem-se que

$$N^*min = 1 + N^*min_{int} + N^*min_f$$

Como todo nó interno, exceto a raiz, contém no mínimo $(K^* + 1)$ filhos,

$$N^*min_f = 2(K^* + 1)^{h^*-1}$$

Analogamente,

$$N^*max = 1 + N^*max_{int} + N^*max_f$$

Como todo nó interno contém no máximo $(2K^* + 1)$ descendentes,

$$N^*max_f = (2K^* + 1)^{h^*}$$

Em função da *árvore-B** conter os valores do índice dos registros de dados localizados nas páginas folhas, o número mínimo e máximo dos valores, I^*min e I^*max , é dado por:

$$I^*min = KN^*min_f = K2(K^* + 1)^{h^*-1}$$

$$I^*max = 2K N^*max_f = 2K(2K^* + 1)^{h^*}$$

Assim,

$$\begin{cases} 2K(K^* + 1)^{h^*-1} \leq I^* \leq 2K(2K^* + 1)^{h^*} & \text{se } h^* \geq 1 \\ I = 0 & \text{se } h^* = -1 \text{ (árvore vazia)} \end{cases}$$

Através desse resultado e utilizando um pouco de álgebra, torna-se simples calcular a altura h^* da árvore $\mathcal{A}^*$.

$$\begin{aligned} I^* &\leq 2K(2K^* + 1)^{h^*} \\ \log_2 I^* &\leq \log_2(2K) + \log_2(2K^* + 1)^{h^*} \\ h^* &\geq \frac{\log_2 \frac{I^*}{2K}}{\log_2(2K^* + 1)} \\ h^* &\geq \log_{(2K^*+1)} \frac{I^*}{2K} \end{aligned}$$

Por outro lado,

$$\begin{aligned} I^* &\geq 2K(K^* + 1)^{h^*-1} \\ \log_2 I^* &\geq \log_2 2K + \log_2(K^* + 1)^{h^*-1} \\ h^* - 1 &\leq \frac{\log_2 \frac{I^*}{2K}}{\log_2(K^* + 1)} \\ h^* &\leq \log_{(K^*+1)} \frac{I^*}{2K} + 1 \end{aligned}$$

Portanto,

$$\begin{cases} \log_{(2K^*+1)} \frac{I^*}{2K} \leq h^* \leq \log_{(K^*+1)} \frac{I^*}{2K} + 1 & \text{se } I^* \geq 1 \\ h^* = -1 & \text{se } I^* = 0 \end{cases}$$

Conforme demonstrado no capítulo 1, a altura de uma *árvore-B* de ordem K' é

$$\begin{cases} \log_{(2K'+1)}(I + 1) - 1 \leq h \leq \log_{(K'+1)} \frac{I+1}{2} & \text{se } I \geq 1 \\ h = -1 & \text{se } I = 0 \end{cases}$$

Considerando o fato de que na prática K^* é maior do que K' , pois não há necessidade de se armazenar os ponteiros para os registros de dados na região de índice, tem-se que o desempenho da *árvore- B^** é muito melhor do que da *árvore-B*. O gráfico adiante (figura 2.9) dado por Wedekind [94], ilustra uma comparação qualitativa para os valores máximos de h e h^* , supondo que $I = I^*$; $R = \frac{K^*}{K'}$.

Figura 2.9: Gráfico comparativo das alturas da *árvore-B** e *árvore-B*.

Nakamura e Mizoguchi [72], Küspert [47], Chu e Knott [18] e Baeza-Yates [3] fizeram uma análise do fator médio de ocupação das *árvores-B**, com ou sem o uso das técnicas de overflow. Nesta análise eles chegaram aos mesmos resultados e fórmulas encontradas para as *árvores-B* na seção 1.6.

2.5 Árvore- B^+

Consideraremos a *árvore- B^+* uma variação da *árvore- B^** onde os nós folhas contêm ponteiros para os registros de dados e não os próprios registros. Dessa forma, a recuperação de um registro de dados com valor i do índice, requer mais um acesso à memória secundária, ao atingir um nó folha. Os nós folhas formam uma lista ligada, permitindo a recuperação dos dados de acordo com valores crescentes dos índices, com necessidade de percurso na árvore somente para o primeiro valor.

Um nó folha da *árvore- B^+* com m registros de dados está representado na figura 2.10, onde $i_1, i_2, \dots, i_m$ são os valores do índice associados aos registros, $p_1, p_2, \dots, p_m$ os ponteiros para estes registros respectivamente e p_{m+1} um ponteiro para o nó folha vizinho à direita.

p_1	i_1	p_2	i_2	$\cdots$	p_m	i_m	p_{m+1}
-------	-------	-------	-------	----------	-------	-------	-----------

Figura 2.10: Representação de um nó folha na *árvore- B^+* .

Analogamente às *árvores- B^** , podemos determinar a altura h^+ de uma *árvore- B^+* , de ordem K^+ para os nós não folhas e K' para os nós folhas (não usamos aqui a letra K , pois serão feitas comparações com a *árvore- B^** , na qual ela foi usada, cf. seção 2.4). Observemos que $K' < K^+$, pois o tamanho dos ponteiros para os registros de dados é, em geral, maior do que o tamanho dos ponteiros para nós na árvore, já que há menos páginas para estes, e contando-se com a ocupação de espaço físico contíguo para esses nós.

Portanto (v. 2.4),

$$\begin{cases} \log_{(2K^++1)} \frac{I^+}{2K'} \leq h^+ \leq \log_{(K^++1)} \frac{I^+}{2K'} + 1 & \text{se } I^+ \geq 1 \\ h^+ = -1 & \text{se } I^+ = 0 \end{cases}$$

onde I^+ é o número de valores do índice localizados nas páginas folhas.

Vamos comparar a altura de uma *árvore- B^** , com ordem K^* nos nós internos e K nas folhas (v. 2.4), com a de uma *árvore- B^+* , que armazenam o mesmo número I de valores do índice. Notemos que $K^+ = K^*$, pois os nós internos das 2 árvores contêm o mesmo tipo de informações. Por outro lado, $K' \gg K$ uma vez que os nós folhas das *árvores- B^** contêm registros de dados e não ponteiros. Como estamos interessados em diferenças de alturas, introduzimos $d_{\min}$ e $d_{\max}$, tal que $h_{\min}^* = h_{\min}^+ + d_{\min}$ e $h_{\max}^* = h_{\max}^+ + d_{\max}$. Considerando $d_{\min} = \log_{(2K^++1)} d'$ e $d_{\max} = \log_{(K^++1)} d''$, temos que:

$$\begin{aligned} \log_{(2K^++1)} \frac{I}{2K} &= \log_{(2K^++1)} \frac{I}{2K'} + \log_{(2K^++1)} d' \\ \log_{(2K^++1)} \frac{I}{2K} &= \log_{(2K^++1)} \left(\frac{I}{2K'} d' \right) \\ \frac{I}{2K} &= \frac{I}{2K'} d' \end{aligned}$$

Dessa forma,

$K' = d'K$, quando realizam-se comparações entre $h_{\min}^*$ e $h_{\min}^+$;

$K' = d''K$, quando realizam-se comparações entre $h_{\max}^*$ e $h_{\max}^+$;

Assim,

Para $d_{min} = 0 \Rightarrow h_{min}^* = h_{min}^+$ e $K' = K$;
 Para $d_{min} \in (0, 1) \Rightarrow 1 < \frac{K'}{K} < 2K^* + 1$;
 Para $d_{min} = 1 \Rightarrow \frac{K'}{K} = 2K^* + 1$;
 Para $d_{min} \in (1, 2) \Rightarrow 2K^* + 1 < \frac{K'}{K} < (2K^* + 1)^2$;
 Para $d_{min} = 2 \Rightarrow \frac{K'}{K} = (2K^* + 1)^2$;

⋮

Para $d_{max} = 0 \Rightarrow h_{max}^* = h_{max}^+$ e $K' = K$;
 Para $d_{max} \in (0, 1) \Rightarrow 1 < \frac{K'}{K} < K^* + 1$;
 Para $d_{max} = 1 \Rightarrow \frac{K'}{K} = K^* + 1$;
 Para $d_{max} \in (1, 2) \Rightarrow K^* + 1 < \frac{K'}{K} < (K^* + 1)^2$;
 Para $d_{max} = 2 \Rightarrow \frac{K'}{K} = (K^* + 1)^2$;

⋮

Notemos que na prática essas relações nem sempre se aplicam, pois as alturas são sempre valores inteiros e conseqüentemente as diferenças entre as mesmas também. No entanto, não há garantia de que as fórmulas para h^* e h^+ dadas dão resultados inteiros.

Poder-se-ia pensar numa solução em que se determinem valores fracionários (isto é, não inteiros) para d_{min} e d_{max} , através dos valores encontrados para h_{min}^* , h_{min}^+ , h_{max}^* e h_{max}^+ respectivamente, e depois sejam feitos ajustes para $\lceil d_{min} \rceil$ e $\lceil d_{max} \rceil$. Entretanto, isso não é correto, pois $\lceil t \rceil - \lceil u \rceil \leq \lceil t - u \rceil$ ³.

Por outro lado, como $t \leq u \leq v \Rightarrow \lceil t \rceil \leq \lceil u \rceil \leq \lceil v \rceil$ podemos partir dos valores fracionários das fórmulas de h^* e h^+ e utilizar $\lceil h_{min}^* \rceil$, $\lceil h_{min}^+ \rceil$, $\lceil h_{max}^* \rceil$ e $\lceil h_{max}^+ \rceil$.

³ $\lceil t \rceil - \lceil u \rceil = t + \Delta_t - u - \Delta_u = t - u + (\Delta_t - \Delta_u)$, onde $0 \leq \Delta_t, \Delta_u < 1$;

$\lceil t - u \rceil = t - u + \Delta_{(t-u)}$, onde $0 \leq \Delta_{(t-u)} < 1$;

Temos que $\max(\Delta_t - \Delta_u) < 1$ e $\max(\Delta_{(t-u)}) < 1$.

Mas, $\min(\Delta_t - \Delta_u) > -1$ e $\min(\Delta_{(t-u)}) = 0$.

Assim, para $(\Delta_t - \Delta_u) \in (-1, 0)$ e $\Delta_{(t-u)} = 0 \Rightarrow \lceil t \rceil - \lceil u \rceil < \lceil t - u \rceil$.

Considere $h_{min}^* = \log_{(2K^*+1)} \frac{I}{2K} = \log_a b$ e $h_{min}^+ = \log_{(2K^++1)} \frac{I}{2K'} = \log_a c$.

$$\lceil \log_a b \rceil - \lceil \log_a c \rceil = d_{min}$$

Para $0 < b \leq a \rightarrow h_{min}^* = 1$
 Para $a < b \leq a^2 \rightarrow h_{min}^* = 2$

Para $0 < c \leq a \rightarrow h_{min}^+ = 1$
 Para $a < c \leq a^2 \rightarrow h_{min}^+ = 2$

$\vdots$

$\vdots$

Para $a^{l-1} < b \leq a^l \rightarrow h_{min}^* = l$

Para $a^{j-1} < c \leq a^j \rightarrow h_{min}^+ = j$

Portanto, determinados l e j obtem-se $d_{min} = l - j$.

Analogamente considere $h_{max}^* = \log_{(K^*+1)} \frac{I}{2K} + 1 = \log_{a'} b' + 1$ e $h_{max}^+ = \log_{(K^++1)} \frac{I}{2K'} + 1 = \log_{a'} c' + 1$.

$$\lceil \log_{a'} b' \rceil - \lceil \log_{a'} c' \rceil = d_{max}$$

Para $0 < b' \leq a' \rightarrow h_{max}^* = 1$

Para $0 < c' \leq a' \rightarrow h_{max}^+ = 1$

$\vdots$

$\vdots$

Para $a^{l'-1} < b' \leq a^{l'} \rightarrow h_{max}^* = l'$

Para $a^{j'-1} < c' \leq a^{j'} \rightarrow h_{max}^+ = j'$

Portanto, determinados l' e j' obtem-se $d_{max} = l' - j'$.

Assim, determinam-se facilmente as diferenças de alturas dependendo das relações existentes entre (a, b) , (a, c) , (a', b') , (a', c') , conforme será mostrado para alguns exemplos práticos no capítulo 6.

Notemos que a comparação entre os casos mínimos e máximos fornece uma avaliação nos casos extremos. Pode-se aplicar o mesmo método para os casos de ocupação média ($69,3\% \cong 0,7$) tomando-se $h_{med}^* = \log_{(1,4K^*+1)} \frac{I}{1,4K}$ e $h_{med}^+ = \log_{(1,4K^++1)} \frac{I}{1,4K'}$.

2.6 Árvore-B Pré-Fixada

Conforme visto anteriormente na *árvore-B** os valores do índice armazenados na região de índice servem de guia para direcionar a busca de um registro de dados com um determinado

valor i para o índice que necessariamente está numa folha. Pode-se observar que não há necessidade de utilizar exatamente os mesmos valores existentes nas páginas folhas para construir a região de índice. Com base nessa idéia, esses valores podem ser escolhidos de forma tal a aumentar o grau dos nós, bastando para isso serem de tamanho pequeno, gerando assim uma diminuição na altura da árvore. Esses valores não necessitam ser exatamente os valores do índice, podendo ser *prefixos*, isto é, sub-cadeias mais à esquerda, que distingam de forma correta o caminho a ser seguido. Essa estratégia só pode ser empregada no caso em que os valores são cadeias de caracteres ou números com um comprimento fixo de dígitos.

Em 1977, pensando em valores na forma de cadeias de caracteres e na perspectiva dos mesmos serem de tamanho variável, Bayer e Unterauer [12] desenvolveram uma variação da *árvore-B** denominada de *árvore-B pré-fixada simples*. Esta estrutura tem a característica de conter prefixos dos valores do índice na região de índice e de não desdobrar um determinado nó exatamente no valor mediano, quando se encontra sobrecarregado, mas sim em um certo *intervalo de desdobramento* pré-determinado em torno desse valor. O valor a ser incluído no nó pai é escolhido entre aqueles que possuem o menor tamanho em relação aos outros valores localizados naquele intervalo (podendo ser eventualmente um prefixo). O intervalo de desdobramento é um determinado espaço do nó. A figura 2.11 ilustra um exemplo de uma *árvore-B pré-fixada simples*.

Figura 2.11: Exemplo de uma *árvore-B pré-fixada simples*.

Sejam θ_f e θ_i os intervalos de desdobramento dos nós localizados nas folhas e dos nós da região de índice, respectivamente. Um aumento de θ_f deve provocar a diminuição do tamanho médio dos valores do índice na região de índice e com isso reduz o número de nós nesta área. Por outro lado, um aumento de θ_i deve favorecer o fato dos valores com tamanhos pequenos localizarem-se próximos à raiz e assim aumentar o grau dos nós. A influência dos intervalos de desdobramento em relação ao desempenho da estrutura é considerado um problema em aberto, em função da dificuldade de análise da mesma. É

necessário considerar, além da observação acima sobre o tamanho do intervalo, o fato de que intervalos grandes produzem gastos adicionais na busca do valor de menor tamanho.

O algoritmo de inserção para a *árvore-B pré-fixada simples* é eficiente e simples. O procedimento consiste em se fazer uma busca para determinar a folha na qual o elemento deve ser inserido. Se esta inclusão proporcionar a sobrecarga de um nó, este deve ser desdobrado, como visto, escolhendo-se o valor (reduzido eventualmente a um prefixo) de menor tamanho dentre aqueles pertencentes ao intervalo. Após a escolha, o valor é inserido na região de índice, propagando novos desdobramentos quando necessário. O procedimento de desdobramento de um nó interno é idêntico.

A operação de remoção de um valor é realizada da mesma maneira que na *árvore-B**. Quando o valor do índice retirado de um nó folha n_f possuir o menor ou maior tamanho dentre os demais valores de n_f , então o valor do separador de n_f localizado na página pai deve ser trocado por algum de menor tamanho, que sirva de localizador para os valores de n_f .

No caso de utilização da técnica de overflow, pode acontecer que um certo valor de um nó seja trocado por outro de tamanho maior ou menor. Com isso, o método de overflow pode ocasionar novos desdobramentos, concatenações ou até mesmo reaplicação da técnica de overflow.

Com o intuito de diminuir a altura das *árvores-B pré-fixadas simples*, Bayer e Unterauer descreveram uma modificação das mesmas dando origem a uma outra denominada de *árvore-B pré-fixada*, que utiliza uma técnica de compressão dos prefixos dos valores do índice localizados na região de índices, ordenados lexicograficamente⁴.

Uma *árvore-B pré-fixada* é definida como uma *árvore-B**, isto é, contendo todos os registros de dados nas páginas folhas, onde a região de índice tem as seguintes características.

1. Estrutura da Árvore

A árvore é completamente balanceada. A ordem K da árvore não é mais caracterizada, pois varia de nó para nó. Não há mais restrição quanto ao limite inferior do número de valores do índice em cada nó, que depende da organização interna do nó.

2. Organização de Um Nó

Um nó n' contém uma sequência de ponteiros para os filhos de n' intercalados por valores parciais do índice ordenados lexicograficamente. Estes últimos constituem-se

⁴Ferguson [29] critica o termo adotado por Bayer e Unterauer, *árvore-B pré-fixada*, alegando que o método de compressão utilizado nesta estrutura não é restrito às árvores, podendo ser empregado em qualquer organização que execute busca de algum elemento.

de sufixos dos valores do índice que contêm prefixos comuns. A figura 2.12 mostra um desses nós, onde $p_0, p_1, \dots, p_m$ são ponteiros para os nós filhos de n e $i'_1, i'_2, \dots, i'_m$ são os valores parciais do índice que servem como separadores.

p_0	i'_1	p_1	i'_2	p_3	$\dots$	p_{m-1}	i'_m	p_m
-------	--------	-------	--------	-------	---------	-----------	--------	-------

Figura 2.12: Estrutura de um nó interno n em uma *árvore-B pré-fixada*.

Considere n um nó de uma árvore cuja região de índice é organizada como uma *árvore-B pré-fixada simples* e $\mathcal{A}(n)$ a sub-árvore maximal enraizada por n . Para todo separador i_s ou valor do índice associado a um registro de dados i_d de $\mathcal{A}(n)$, existe um limitante inferior $Inf(n)$ e superior $Sup(n)$ tal que, $Inf(n) \leq i_s < Sup(n)$ e $Inf(n) \leq i_d < Sup(n)$. Para o nó raiz r tem-se que $Inf(r) = a$ e $Sup(r) = \infty$, onde a é a primeira letra do alfabeto. Para um nó $n : p_0, i_1, \dots, i_m, p_m$, onde $n(p_j)$ é o nó filho de n apontado por p_j , $0 \leq j \leq m$, tem-se que:

$$Inf(n(p_j)) = \begin{cases} i_j, & j = 1, 2, \dots, m \\ Inf(n), & j = 0 \end{cases}$$

$$Sup(n(p_j)) = \begin{cases} i_{j+1}, & j = 0, 1, \dots, m-1 \\ Sup(n), & j = m \end{cases}$$

Assim, pode-se definir um prefixo comum $\mathcal{X}(n(p_j))$ a todos os separadores ou valores de $\mathcal{A}(n(p_j))$. Seja $\overline{X}_j$ o prefixo (eventualmente vazio) de maior tamanho, comum a $Inf(n(p_j))$ e $Sup(n(p_j))$. Então,

$$\mathcal{X}(n(p_j)) = \begin{cases} \overline{X}_j C_l, & \text{se } Inf(n(p_j)) = \overline{X}_j C_l z \text{ e } Sup(n(p_j)) = \overline{X}_j C_{l+1} \\ \overline{X}_j, & \text{caso contrário} \end{cases}$$

onde z é uma cadeia de caracteres arbitrária e o caractere C_{l+1} é sucessor lexicográfico imediato de C_l no alfabeto.

Note-se que $Inf(n(p_j))$, $Sup(n(p_j))$ e $\mathcal{X}(n(p_j))$ podem ser facilmente determinados percorrendo a árvore da raiz até $n(p_j)$.

Em um nó n' , um valor i' parcial do índice é facilmente determinado retirando-se do valor original i , o prefixo $\mathcal{X}(n')$. Portanto,

$$\forall i_j \in n', i_j = \mathcal{X}(n') i'_j, 0 \leq j \leq m$$

O prefixo $\mathcal{X}(n(p_j))$ só precisa ser armazenado uma vez em $\mathcal{A}(n(p_j))$. No caso dos nós folhas, quando se deseja realizar a busca sequencial, então os valores do índice devem ser armazenados em seu formato total (não parcial).

Considerando $I(n(p_j))$ o conjunto de todos os separadores e valores do índice pertencentes a $\mathcal{A}(n(p_j))$, tem-se que:

- (a) $\forall i \in I(n(p_0)), i < i_1$
- (b) $\forall i \in I(n(p_j)), i_j \leq i < i_{j+1} \quad j = 1, 2, \dots, m-1$
- (c) $\forall i \in I(n(p_m)), i \geq i_m$

A figura 2.13 ilustra a *árvore-B pré-fixada* baseada na árvore da figura 2.11. Observe que:

Figura 2.13: Exemplo de uma *árvore-B pré-fixada*.

A seguir serão dados os procedimentos de busca, inserção e remoção para as *árvores-B pré-fixadas*. O trecho do algoritmo abaixo constitui a operação de busca de um valor i do índice em um nó n não folha, tal que, $n : p_0, *, i'_1, p_1, *, i'_2, \dots, *, i'_m, p_m, *$, onde $*$ consiste de um símbolo especial (delimitador) que não pode aparecer em i'_j e p_j ($0 \leq j \leq m$). Este delimitador é utilizado para distinguir os ponteiros dos valores do índice, uma vez que estes últimos são de tamanho variável. Notemos que os autores dessa proposta não especificam

como o delimitador deve ser implementado; no capítulo 6 propomos uma implementação para o mesmo.

Algoritmo:

1. Determine $\mathcal{X}(n)$;
2. Remova $\mathcal{X}(n)$ de i dando origem ao valor i' ;
3. Se $i' < i'_1$, então $n := n(p_0)$;
 Se $i'_j \leq i' < i'_{j+1}$, então $n := n(p_j)$, $1 \leq j < m$;
 Se $i'_m \leq i'$, então $n := n(p_m)$;
4. Se n é um nó folha, então recupere o registro com valor i (caso exista)
 Senão, retorne ao passo 1;

Apesar dos valores do índice dentro de um nó serem de tamanho variável o uso de um delimitador na frente de cada valor possibilita a realização da busca binária. A idéia consiste em localizar, a cada iteração, uma certa posição s da metade da parte do nó sendo usado e em seguida caminhar à esquerda buscando a posição s' do início do nó ou do próximo delimitador. Como os ponteiros têm tamanhos fixos, torna-se fácil encontrar o valor i'_j que será usado para comparação. Notemos que os delimitadores poderiam ser colocados na frente de cada ponteiro, excluindo-se p_0 e p_m . Observemos que essa idéia também pode ser utilizada no método de busca em raiz quadrada (v. 1.4.1) dividindo-se o nó em $\sqrt{I(n)}$ intervalos, onde $I(n)$ é o conjunto dos valores de tamanho variável de um nó n . Se a divisão ocorrer no meio de um valor, caminha-se para a esquerda até encontrar um delimitador, limitando um intervalo nesta posição.

O algoritmo de inserção de um registro consiste da operação inicial de busca do nó folha que receberá a nova informação. Normalmente, após a detecção da posição correta de inclusão, o dado é inserido e o procedimento concluído. Quando ocorre o desdobramento de um nó n , o mesmo é dividido em n e n' e selecionado o valor i , cujo valor parcial i' será colocado em n_p (pai de n). Como $\mathcal{X}(n_p)$ satisfaz a propriedade de ser um prefixo válido para i e consequentemente, $i = \mathcal{X}(n_p)i'$, então a inclusão de i' em n_p não afeta nenhum outro valor de n_p . Por sua vez, no caso em que $\mathcal{X}(n)$ ou $\mathcal{X}(n')$ forem de tamanho maior do que o antigo valor de $\mathcal{X}(n)$, então os valores parciais desses nós devem ser ajustados.

Quando for utilizada a técnica de overflow podem acontecer os mesmos problemas verificados para as *árvores-B pré-fixadas simples*, isto é, ocorrência de novos desdobramentos, concatenações ou reaplicações da técnica de overflow, em virtude das mudanças nos valores do índice.

A remoção de um registro também realiza a busca do nó folha contendo o elemento a ser retirado. No caso de concatenação entre dois nós n e n' , retira-se o separador (valor parcial) localizado no pai dos mesmos e calcula-se novos valores parciais para os separadores do novo nó. A condição de união de dois nós está diretamente ligada com à determinação desses novos valores, ou seja, depende do fato dos novos valores parciais caberem ou não no nó.

Em função da dificuldade de análise da quantidade de espaço que é preservada quando se utiliza a *árvore-B pré-fixada simples* ou a *pré-fixada*, Bayer e Unterauer limitaram-se a resultados experimentais para comparar o desempenho destas estruturas com a *árvore-B* e algumas de suas variações, chegando às seguintes conclusões:

1. Tempo de Processamento

O tempo para executar os algoritmos nas *árvores-B pré-fixadas simples* é praticamente igual ao das *árvores-B**; todavia, as *árvores-B pré-fixadas* necessitam de 50-100% a mais de tempo, em função dos cálculos dos valores parciais do índice. Notemos que hoje os processadores são muito mais rápidos, de modo que o tempo de acesso aos discos deve predominar, diminuindo a diferença apontada por aqueles autores.

2. Acessos ao Disco

Para as árvores com cerca de 200 nós o tempo de acesso ao disco é praticamente o mesmo, entretanto aquelas que possuem de 400 a 800 nós o acesso ao disco é reduzido em 20-25% nas *árvores-B pré-fixadas simples* quando comparado com as *árvore-B**. Já as *árvores-B pré-fixadas* utilizam 2% a menos que as *simples*.

2.7 Árvore-B com Multitabelas

Visando utilizar valores do índice de tamanho variável, Lomet [57] desenvolveu, em 1979, um método para diminuir o espaço ocupado pelos valores do índice na região de índice, e consequentemente aumentar o grau dos nós, diminuindo os acessos ao disco.

A principal diferença entre o método proposto por Lomet e os demais consiste na maneira em que os valores do índice são armazenados nos nós não folhas. Nas outras variações eles são organizados pela ordem dos valores, executando-se buscas sequenciais, binárias ou até mesmo o método da busca em raiz quadrada (v. 1.4.1) para localizar um elemento. Nesta, a idéia consiste na separação dos elementos pelo tamanho dos seus valores do índice. Cada conjunto de valores de mesmo tamanho é armazenado em uma tabela específica, de forma ordenada. Cada entrada em uma tabela contém um valor e

seu ponteiro para o nó filho. Como existem valores de diferentes tamanhos, existem várias tabelas. Esse tipo de organização permite uma economia de espaço através da diminuição das informações de controle. Cada tabela deve ser precedida pelo seu tamanho e pelo número de suas entradas, diferindo das outras técnicas que utilizam valores comprimidos, onde cada entrada precisa ter um campo extra para guardar o seu respectivo tamanho ou o seu delimitador.

O método tenta evitar a existência de tabelas com uma baixa quantidade de informações através de 2 maneiras distintas. Uma das estratégias consiste em unir todas as entradas com tamanhos distintos, que apareçam em pequenas porções, em uma única tabela; a outra caracteriza-se por colocar estas entradas em tabelas contendo uma boa quantidade de dados, mesmo que sejam de tamanhos diferentes dessas entradas.

Durante a busca de um certo valor i do índice na região de índice, nem sempre é possível determinar o elemento igual a i . O objetivo é localizar um valor i_j em uma tabela, tal que $i \leq i_j$ e não existe i_l onde $i \leq i_l < i_j$. O procedimento visita todas as tabelas, determinando em cada uma o valor i_j que satisfaça a condição acima. O menor i_j encontrado corresponde ao valor desejado. Por outro lado, as operações de inserção e remoção realizadas na região de índice só ocorrem após o desdobramento ou concatenação de algum nó folha. Nesse caso, como o procedimento conhece o tamanho do valor a ser incluído ou removido, não há necessidade de se percorrer todas as tabelas, apenas aquela associada ao tamanho deste valor.

Lomet garante que o método se multitabelas é competitivo com outros métodos tais como: busca binária empregada por Bayer e Unterauer [12], buscas binárias com entradas de tamanho fixo, entre outras.

2.8 Árvore- B^L

Para permitir algoritmos especiais de manipulação de *árvore-B* com processos concorrentes, Lehman e Yao [53] desenvolveram em 1981 uma variação da *árvore-B**. Esta estrutura foi também empregada posteriormente na proposta de concorrência apresentada por Sagiv [84]⁵, sendo conhecida na literatura por *árvore-B^{Ligada}*. Por simplicidade chamaremos de *árvore-B^L*.

A *árvore-B^L* caracteriza-se por conter nós folhas com ponteiros para os registros de dados. Cada nó da árvore contém um par (valor, ponteiro) e mais, isto é, para todo nó

⁵O capítulo 4 contém a descrição detalhada das soluções para o problema de concorrência propostas por Lehman e Yao e Sagiv.

n , existe um par (i_{m+1}, p_{m+1}) , $K \leq m < 2K$, onde o conteúdo de i_{m+1} é o maior valor existente na sub-árvore enraizada por p_m em n , e p_{m+1} , chamado de *ponteiro de ligação*, aponta para o nó vizinho à direita de n . Assim, os nós pertencentes a um mesmo nível da árvore formam uma lista ligada, possibilitando uma maneira alternativa para percorrer os nós. A figura 2.14 mostra a estrutura de um nó da *árvore- B^L* e a figura 2.15 ilustra um exemplo da mesma.

Figura 2.14: Estrutura de um nó n na *árvore- B^L* .

Figura 2.15: Exemplo numérico de uma *árvore- B^L* de ordem $K = 2$.

Observa-se claramente que o nó mais à direita de cada nível contém o valor i_{m+1} igual ao maior valor do índice armazenado e o ponteiro p_{m+1} indefinido (nil). Por outro lado, durante a criação da estrutura, os valores mais elevados em cada nó folha são colocados 2 vezes nestes nós; todavia, as operações de remoções de elementos poderão acarretar mudanças desta situação de duplicação em determinados valores.

A existência do ponteiro de ligação ocasiona a realização do desdobramento de um nó n em 2 fases: n é dividido em 2 e em seguida incluído um ponteiro de ligação do irmão à esquerda para o nó à direita; só após esta operação executa-se a inclusão do par (valor, ponteiro) no pai apropriado. Com isso, a estrutura permite nós sem pai, mas com vizinhos (irmãos), os quais são facilmente alcançados pelos ponteiros de ligação.

Analogamente, a concatenação de 2 nós proporcionada pela remoção de algum elemento, também é realizada em 2 passos: executa-se primeiro a união dos nós e depois elimina-se as devidas informações do nó pai.

2.9 Árvore-B Histórica

Em uma *árvore-B* de ordem K o número de desdobramentos e concatenações acarretados pela inserção e remoção de algum elemento é, no pior caso, igual a altura da árvore ($O(\log_K I)$). Assim, para uma árvore onde ocorrem u operações de inserções e remoções, o número de divisões e uniões de nós é dado por ($O(u \log_K I)$).

Com o intuito de diminuir essa quantidade, Maier e Salveter [62] desenvolveram em 1981 um novo método para as *árvores-B*, denominada de *árvore-B histórica*⁶. Esta nova versão consiste de uma *árvore-B* de ordem K_p , ou seja, permite nós com no mínimo $(K-p)$ valores do índice, $p \leq \frac{K-1}{2}$, ao invés de K como originalmente. Assim, a quantidade de concatenações executadas na árvore diminui, uma vez que agora são permitidos nós com $K-p \leq m \leq 2K$. Será mostrado adiante que nessa estrutura o número de concatenações e desdobramentos que ocorrem na árvore para u operações de inserções e remoções é reduzido para $O(u)$.

Nas *árvores-B históricas* a operação de inserção é idêntica a *árvore-B* original. Um nó é desdobrado quando o número de valores do mesmo for $2K+1$. A remoção de um valor em um nó n com $(K-p)$ valores faz com que se verifique se algum outro nó vizinho a n contém pelo menos $(K-p+1)$ valores. Em caso afirmativo, esse vizinho empresta um de seus elementos para n ficando cada um deles com $(K-p)$; entretanto, caso os nós vizinhos de n possuírem cada um $(K-p)$ valores, então realiza-se a concatenação de n com algum destes, gerando um nó com $(2(K-p))$ valores.

O número de operações de inserções e remoções, que podem ser realizadas em um nó n da árvore recentemente criado através de uma operação de concatenação ou desdobramento, antes que ocorra outra concatenação ou divisão envolvendo n , é facilmente calculado conforme especificado nos casos adiante.

Caso 1: Número de operações entre 2 desdobramentos.

Para que n seja desdobrado o mesmo precisa conter $2K+1$ valores; como após a primeira divisão ele possuía K valores, então são necessárias $(K+1)$ inserções para ocorrer a segunda divisão.

⁶O artigo afirma que independentemente, em 1979, Kurt Melhorn (não publicado) propuseram uma estrutura denominada de '*weak B-tree*', a qual tem as mesmas características da *árvore-B histórica*.

$$K + x = 2K + 1 \Rightarrow x = K + 1 \text{ inserções}$$

Caso 2: Número de operações entre 1 desdobramento e 1 concatenação.

Após a primeira divisão, n contém K valores. Para haver uma união envolvendo-o é necessário que ocorram $(p + 1)$ remoções em n , afim de que o mesmo possua $(K - p - 1)$ elementos.

$$K - x = (K - p - 1) \Rightarrow x = p + 1 \text{ remoções}$$

Caso 3: Número de operações entre 1 concatenação e 1 desdobramento.

Depois de uma concatenação realizada em n , este possui $(2(K - p))$ valores. Para que ocorra um desdobramento em n , ele precisará conter $(2K + 1)$ valores; assim, são necessárias $(2p + 1)$ inserções.

$$2(K - p) + x = 2K + 1 \Rightarrow x = 2p + 1 \text{ inserções}$$

Caso 4: Número de operações entre 2 concatenações.

Logo em seguida a uma concatenação n tem $(2(K - p))$ valores. Para ocorrer uma segunda concatenação envolvendo-o é preciso que o mesmo fique com $(K - p - 1)$ elementos, fato que acontece após serem realizadas $(K + 1 - p)$ remoções em n .

$$2(K - p) - x = K - p - 1 \Rightarrow x = K + 1 - p \text{ remoções}$$

Como $p \leq \frac{K-1}{2}$, então o menor número de operações ocorre entre um desdobramento e uma concatenação, quando são realizadas $(p + 1)$ operações de remoções (caso 2). Diante deste fato, considere o seguinte teorema provado por Maier e Salveter.

Teorema

Em uma *árvore-B histórica* de ordem K_p , inicialmente vazia, na qual são executadas u operações de inserções e remoções, realizam-se no máximo $\frac{u}{p}$ desdobramentos e concatenações.

Prova

Observa-se que em cada inserção ou remoção realizada na árvore, apenas um único nó dentre aqueles que não sofreram desdobramentos ou concatenações tem seu número de elementos modificados. Ele pode ser o nó no qual a propagação das concatenações e desdobramentos terminam, o nó vizinho a este, que por ventura tenha emprestado algum elemento ou o próprio nó que ganhou ou perdeu um elemento.

Considere um nó n cujo número de elementos é modificado a cada inserção ou remoção realizada na árvore, sem haver nenhuma união ou divisão envolvendo-o.

Conforme visto anteriormente, no caso 2, este nó deve ter o número de elementos modificado pelo menos p vezes, antes que possa ocorrer uma concatenação ou desdobramento em torno do mesmo. n sofrerá pelo menos p inserções ou remoções antes de ser desdobrado ou concatenado com algum outro nó. Portanto, pode-se afirmar que o número total de desdobramentos e concatenações realizados em u inserções e remoções é no máximo $\frac{u}{p}$.

Com base no teorema é possível afirmar que o número total de desdobramentos e concatenações após u operações de inserções ou remoções é reduzido para $O(u)$.

2.10 Árvore-BD (Árvore-B Digital)

As *árvores-B* tendem a exigir mais acessos à memória secundária do que os métodos de hashing⁷. Entretanto, estes últimos têm um péssimo desempenho em buscas sequenciais por um índice ou consultas compreendidas dentro de um certo intervalo de valores desse índice (buscas por intervalo). Visando diminuir o tempo de acesso nas *árvores-B*, Lomet [58] propôs em 1981 uma variação que possibilita um aumento substancial do tamanho do nó. Essa variação é denominada de *árvore-B digital* e permite nós com várias páginas garantindo que em cada acesso aleatório apenas uma página de cada nó é examinada. Para que isso seja possível, a operação de busca de um valor i do índice é realizada de forma similar à técnica empregada nos métodos de *hashing extensível* [28], fazendo uso dos bits dos valores do índice para determinar a página correta a ser visitada, mas sem perder a sequencialidade das informações.

A *árvore-BD* mantém todos os dados do arquivo nas folhas, com os nós internos (região de índice) servindo apenas para direcionar as consultas. A região de índice é organizada como nas *árvores-B pré-fixadas simples*, isto é, os valores dentro de cada nó são geralmente prefixos. Baseando-se na idéia de hashing extensível, o número de páginas em cada nó é uma potência de 2.

Para facilitar as operações executadas nas *árvores-BD* armazenam-se informações especiais como tamanho dos valores do índice e tamanho de cada nó, pois a estrutura permite nós com uma ou várias páginas.

A operação de busca de um determinado valor i é realizada começando na raiz, percorrendo toda a árvore, mas examinando apenas uma página por cada nó visitado. Como nas *árvores-B pré-fixadas*, o procedimento localiza uma entrada de um nó n visitado que corresponda a algum prefixo p do valor i sendo procurado. Se não existir nenhum valor

⁷Esta técnica não será explicada por fugir do escopo do texto.

em n igual a p , então a busca não teve sucesso e o processo pára. Caso contrário, o procedimento examina a página apropriada de um nó filho de n . Para determinar essa página, remove-se de i o prefixo p encontrado, isto é, considerando inicialmente $i = x_1x_2 \cdots x_m$ e $p = x_1x_2 \cdots x_j$ ($j < m$), o novo valor gerado pela eliminação de p em i é $i' = x_{j+1} \cdots x_m$ ⁸. Em seguida, utiliza-se o tamanho do nó n para calcular o deslocamento no respectivo filho. Seja t este tamanho. O processo seleciona os bits $x_{j+1} \cdots x_{j+t}$ de i , os quais são tratados como um valor inteiro positivo na forma binária. Esse valor é somado ao endereço do nó filho a ser examinado. O resultado dessa adição é o endereço da página do nó filho a ser examinada (aquela que contém todas as entradas começando com $x_1 \cdots x_j x_{j+1} \cdots x_{j+t}$). O procedimento continua dessa maneira, até atingir um nó folha que é examinado para se verificar a existência do valor sendo procurado.

O procedimento de inclusão de um registro com valor i em uma *árvore-BD* é semelhante ao da *árvore-B*, diferindo na situação em que uma página de algum nó se torna sobrecarregada. Os desdobramentos das páginas são realizados seguindo-se a técnica de hashing extensível. Seja $\mathcal{P}$ uma página sobrecarregada em um nó n , onde todos os valores de $\mathcal{P}$ têm o prefixo x em comum. $\mathcal{P}$ é desdobrada em duas páginas $\mathcal{P}'$ e $\mathcal{P}''$, colocando-se em $\mathcal{P}'$ todos os valores que começam com $x||0$ em $\mathcal{P}''$ os que começam com $x||1$ ⁹.

Quando n não contém espaço para permitir o desdobramento de $\mathcal{P}$, pode ocorrer uma duplicação do seu tamanho ou um desdobramento do nó.

1. Duplicação do Tamanho do Nó

Esta estratégia consiste em alocar um novo bloco com o dobro do tamanho do nó n sobrecarregado. Se n possui 2^N páginas, o novo bloco terá 2^{N+1} . O conteúdo de cada página deste nó é distribuído entre as várias páginas do novo bloco de forma digital. O nó n é liberado e o novo bloco torna-se o novo nó, atualizando-se o ponteiro do nó pai de n e seu tamanho.

2. Desdobramento do Nó

Neste método o nó n é desdobrado em 2 onde cada metade recebe um tratamento diferente. A parte do nó contendo a página $\mathcal{P}$ sobrecarregada é transferida para um novo bloco que contém o dobro do número de páginas dessa metade, isto é, o mesmo número de páginas do nó n original. As informações dessa parte são distribuídas da mesma maneira que no caso da duplicação do tamanho do nó e no final, a parte antiga é liberada. A outra metade pode ter 2 tratamentos dependendo do seu fator de ocupação. Se esse fator for bom, então não há mudanças. Entretanto, no caso em que essa região é pouco utilizada, alocam-se suas informações em um menor número

⁸Quando o valor de busca i tem tamanho menor do que os valores da *árvore-BD*, este é aumentado acrescentando-se bits 0 no final do mesmo.

⁹Esse tipo de desdobramento foi denominado por Lomet de *digital*, dando origem ao nome da estrutura (*árvore-B digital*). O sinal $||$ significa nesse contexto, *concatenado com*.

de páginas. Os dados de 2 ou 4 páginas adjacentes são colocadas em uma única página, liberando-se aquelas que não forem mais utilizadas. Lomet denominou essa técnica de *consolidação*. O separador do nó pai que referenciava o nó desdobrado é trocado por 2 novos separadores apontando para os novos nós.

A duplicação do tamanho de um nó evita o aumento da altura da árvore, mas acarreta um baixo fator de ocupação de memória, em função do aumento do número de páginas no nó. Por outro lado, o desdobramento de um nó permite uma ocupação adequada, porém gera um aumento na altura da árvore. Dessa forma, é necessário decidir quando um nó deve ser desdobrado ou duplicado de tamanho. Para isso, observam-se os seguintes aspectos: fator de ocupação do nó pai e dos irmãos, tamanho médio dos nós irmãos e informações locais do nó, fornecidas através do perfil de suas páginas. Quando a ocupação global do nó é alta, então o nó deve ser duplicado; quando a ocupação é elevada apenas na página sobrecarregada, o nó deve ser desdobrado. Como é extremamente trabalhoso manter o perfil de ocupação de um nó, Lomet sugere que se guarde um perfil aproximado, o qual deve estar localizado na primeira ou última página do nó.

A remoção de um registro com valor i consiste em simplesmente retirá-lo de um nó folha. Quando isso acarretar um nó com baixa ocupação, o mesmo pode ser consolidado como explicado anteriormente, gerando um nó com um número menor de páginas. A remoção de um nó só ocorre quando o mesmo não contém nenhuma informação.

Para a criação de uma *árvore-BD*, dado um conjunto de valores iniciais, Lomet sugere que a mesma seja realizada em 2 fases. Na primeira fase os nós são sempre desdobrados e a árvore atinge a altura que for necessária para armazenar as informações. Se essa altura ultrapassar 3 níveis (raiz, região de índice e folhas), então executa-se a fase 2. A fase 2 consiste na união dos nós adjacentes na região de índice, até que os níveis mais altos sejam aglutinados em um único nó raiz. Esse método apresenta a vantagem da fase 2 ser apenas utilizada quando o número de dados for elevado, gerando uma árvore balanceada em termos de ocupação/ordem do nó.

A análise realizada por Lomet, no que tange à ocupação do espaço e o número de informações contidas em cada nó da *árvore-BD*, foi feita através de uma função de distribuição uniforme dos valores do índice dentro do espaço reservado para os mesmos. Essa função foi baseada em um fator de desdobramento $\mathcal{D}$, que é uma fração dos valores do índice, variando entre 0.5 (distribuição uniforme) e 1.0 (quando todos os valores são alocados em metade do espaço original). O fator $\mathcal{D}$ é uma constante e sempre que uma página for desdobrada, uma das partes conterá $\mathcal{D}$ valores e a outra $(1 - \mathcal{D})$. $\mathcal{D}$ pode também ser usado para prever futuras distribuições, isto é, quando I novos valores forem adicionados nas 2 páginas, então $\mathcal{D} * I$ serão acrescidos àquela que contém $\mathcal{D}$ e $(1 - \mathcal{D}) * I$ à outra.

Considere $\mathcal{U}_N$ a ocupação de um nó n com 2^N páginas, onde uma delas encontra-se

sobrecarregada. Sabendo-se que a ocupação máxima para os nós com uma única página (2^0) é 100% ($\mathcal{U}_0 = 1$) e que após uma duplicação na mesma esta ocupação é reduzida para $\mathcal{D}$, podendo receber $\frac{1}{\mathcal{D}}$ novos valores até ocorrer uma nova duplicação envolvendo-a, então tem-se que a ocupação máxima para o nó n é dada por:

$$\mathcal{U}_N = \frac{(\frac{1}{\mathcal{D}})^N}{2^N} = \frac{1}{(2\mathcal{D})^N}$$

Finalmente, pode-se afirmar que a *árvore-BD* é uma estrutura que combina as vantagens da *árvore-B* e do método de hashing extensível, porém apresenta uma implementação complexa, além de poder haver um aumento do espaço do disco ocupado pelo arquivo.

2.11 Organização Bounded Disorder

Em 1987, Litwin e Lomet [56] e em 1988 Lomet [60], propuseram uma forma indexada de organizar os arquivos de dados, usando uma árvore de índice, como por exemplo *árvore-B*, em que cada nó de dados é formado por uma sequência contígua de várias páginas. Essa variação foi denominada de *organização Bounded Disorder (BD)* (desordem limitada) e reúne as vantagens das árvores indexadas e do método de hashing, garantindo um bom desempenho e uma boa ocupação de memória.

O método baseia-se na premissa de que o arquivo de índice para um arquivo de dados esteja residente na memória primária. Cada acesso aleatório exige ainda uma aplicação do método de hashing, para recuperar a página de dados correta. Cada nó é dividido em um número variável b de *buckets*¹⁰ e cada bucket é constituído de $\mathcal{P}$ páginas. Em geral b é uma potência de 2 e $\mathcal{P}$ é 2 ou 3. O arquivo de índice é estruturado de maneira independente da organização *BD*, devendo ser pequeno e permitindo consultas rápidas. Para selecionar o bucket correto em um nó de dados contendo o registro sendo procurado, utiliza-se a técnica de *hashing linear* [55].

A limitação no método consiste da perda da sequencialidade dentro de um nó de dados, porém não entre os nós: todos os valores de um determinado nó de dados são maiores do que aqueles do nó precedente e menores do que o seu sucessor. Todavia, dentro de um nó eles são armazenados de forma aleatória ao longo dos *buckets*¹¹. Os arquivos suportam acessos aleatórios, de intervalo e sequencial, como explicado adiante.

1. Aleatório

¹⁰A idéia de *buckets* é muito utilizada nos métodos de hashing que permitem registros de tamanho variável.

¹¹Essa é a origem do termo “desordem limitada”.

Para buscar ou inserir um registro de dados com o valor i do índice, percorre-se o arquivo de índice, até atingir a entrada que referencia o nó de dados n apropriado. Em seguida, aplica-se a função de hashing ao valor i , cujo resultado é um valor entre 0 e 1, e multiplica-se pelo número de buckets de n . O resultado é o número relativo do bucket. Esse número é multiplicado pelo número de páginas por bucket, gerando o deslocamento da mesma que é adicionado ao endereço do nó n , obtendo-se o endereço da página em questão. O dado sendo procurado deverá estar nesta posição do bucket ou, no pior caso, em um outro bucket contendo os registros que não coberam no original, situação de overflow de hashing que será abordada de forma mais detalhada adiante. Observa-se que apenas o bucket contendo a informação desejada é trazido para a memória primária, com exceção do caso de overflow.

2. De Intervalo

Este tipo de consulta não é executado em arquivos organizados em tabelas de hashing, pois é necessário percorrer todo o arquivo para localizar os dados desejados. Todavia, como nos arquivos *BD* os nós internos contêm os valores ordenados, então limitam-se os nós de dados que podem conter os valores sendo procurados, evitando a visita de todo o arquivo.

Nas *árvores-B* esse tipo de consulta é realizada localizando-se o nó contendo o valor do índice inicial do intervalo seguindo-se da leitura sucessiva dos nós, até determinar aquele em que aparece um valor não pertencente ao intervalo. No método de *BD* o processo é similar com a diferença de que registros fora do intervalo são eventualmente examinados. Litwin e Lomet afirmam que os acessos de intervalos nos arquivos *BD* executam menos operações de entrada/saída do que nas *árvores-B*, pois cada leitura ao disco recupera uma quantidade maior de informações, isto é, um nó possui várias páginas.

3. Sequencial

A busca sequencial requer que a recuperação dos registros seja realizada obedecendo uma determinada ordem dos seus valores do índice, ou seja, é uma busca de intervalo onde os dados devem estar ordenados. Como os registros dentro de um nó de dados não estão necessariamente classificados, então é necessário ordená-los. Litwin e Lomet sugerem que a ordenação seja feita através da técnica de *heap sort*, o que requer tempo logaritmo no número de buckets de um nó. O tempo economizado nas operações de entrada/saída dos arquivos *BD* é desperdiçado com a ordenação das informações; entretanto, o tempo das operações de entrada/saída diminui linearmente com o tamanho do nó, enquanto que a ordenação aumenta logaritmicamente com este tamanho.

Além dos desdobramentos e concatenações típicos das *árvores-B* nas operações de inserção e remoção respectivamente, a estrutura de hashing linear permite que os nós e

dados sejam *expandidos* e *encolhidos*. A expansão de um nó de dados ocorre quando o mesmo, que contém várias páginas, se torna tão cheio que produz um elevado número de colisões no método de hashing. A solução é aumentar o número de buckets do nó sobrecarregado distribuindo os dados em um novo nó. Esse novo nó contém o número de buckets igual a um fator de expansão multiplicado pela quantidade de buckets do nó original. Quando o fator de expansão é igual a 2, diz-se que a expansão foi total; quando o fator é compreendido entre 1 e 2, trata-se de uma expansão parcial. De maneira inversa, o encolhimento de um nó de dados acontece quando o mesmo sofreu mais remoções do que inserções. A técnica consiste da redistribuição dos dados em um novo nó com um número menor de buckets. Essa técnica de expansão dos buckets retardando o desdobramento foi proposta por Lomet [59] como um método de tratamento de overflow, visando um aumento do fator médio de ocupação de espaço. De acordo com Baeza-Yates [5], este fator é em média 81%.

Quando um nó de dados contém um número excessivo de buckets é preferível desdobrá-lo ao invés de expandi-lo. Para isso, criam-se 2 novos nós de dados de mesmo tamanho com a soma dos números de buckets maior do que o nó original. A divisão ocorre em torno de um valor mediano do índice, de tal maneira que cada nó contenha aproximadamente a metade das informações do nó original. Já a concatenação de 2 ou mais nós acontece com nós adjacentes, reduzindo a quantidade de espaço do disco consumido, inserindo-se os registros de um nó em outro, percorrendo-se um bucket por vez.

Algumas vezes, o número de informações dentro de um bucket ultrapassa o limite do mesmo (overflow de hashing). Nessas situações a solução dada por Litwin e Lomet consiste em alocar as informações adicionais em outro bucket referenciado pelo original. Para garantir bons resultados nos acessos sequenciais e aleatórios, além de uma boa ocupação de memória, os autores propõem a existência de apenas um bucket extra por cada nó de dados, o qual continua a fazer parte do nó. Todos os registros deste bucket seriam alcançados com o custo de um acesso a mais ao disco. Assim, quando um registro não for localizado no bucket original que contenha outro bucket adicional, então o registro é procurado neste último.

Boswell e Tharp [15] fizeram um estudo comparativo entre as *árvores-B** e os arquivos *BD*, chegando as seguintes conclusões:

1. Quanto maior o tamanho dos nós de dados nos arquivos organizados em *BD*, melhor o acesso aleatório e o desempenho.
2. A recuperação sequencial nas estruturas de *BD* é mais lenta do que nas *árvores-B**, em função da não ordenação dos elementos dentro dos nós.
3. Nos arquivos *BD*, a distribuição dos valores do índice pode diminuir a ocupação da

memória, pois dependendo da função de hashing utilizada, a mesma pode gerar uma sobrecarga de registros alocados no mesmo nó.

Visando completar o assunto sobre as técnicas de overflow empregadas em estruturas de *árvores-B* que contêm buckets nas folhas, abordaremos a *técnica ajustável de overflow* sugerida por Baeza-Yates [6].

Considere 2 possíveis tamanhos para os buckets, $\mathcal{P}$ e $2\mathcal{P}$, e as seguintes situações:

1. Se um bucket de tamanho $\mathcal{P}$ encontra-se sobrecarregado, ele é expandido para um bucket de tamanho $2\mathcal{P}$.
2. Se um bucket de tamanho $2\mathcal{P}$ tornar-se sobrecarregado, faz-se um desdobramento acarretando a inserção de um novo valor do índice na *árvore-B*. Cria-se um novo bucket de tamanho $\mathcal{P}$, deixando-se $\alpha(2\mathcal{P} + 1)$ valores no antigo e colocando-se $(1 - \alpha)(2\mathcal{P} + 1)$ valores no novo bucket, onde $\frac{1}{2} \leq \alpha \leq \frac{3}{4}$. Como o desdobramento é assimétrico, o novo bucket é colocado à direita (esquerda) do antigo se o valor do índice que ocasionou a sobrecarga é maior (menor ou igual) do que o $(\mathcal{P} + 1)$ -ésimo valor no bucket.

Uma consequência desse desdobramento ajustável é ter o fator de ocupação de espaço igual a α , quando os valores forem inseridos de forma ordenada. A tabela 2.1 dada por Baeza-Yates [6] ilustra uma comparação do fator de ocupação das diversas técnicas de overflow existentes.

Técnica	Pior caso	Caso médio	Val. ordenados
Simples	50%	69%	50%
<i>árvore-B</i> ^o	67%	81%	67%
Expansão (Lomet)	67%	81%	67%
Ajustável ($\alpha = 0,68$)	50%	76%	68%
Ajustável ($\alpha = 0,75$)	50%	75%	75%

2.12 Árvore-Bit

A mais recente variação da *árvore-B* consiste da *árvore-Bit*, proposta por Ferguson [29] em 1992, implementada para o sistema/36 da IBM. A principal característica dessa estrutura é a utilização de *bits de distinção* nos nós folhas, para localizar um determinado registro de dados, ao invés dos valores do índice, proporcionando uma compactação nas

folhas. Os nós não folhas da estrutura podem ser representados por qualquer técnica convencional; Fergunson recomenda que os mesmos sejam representados como uma *árvore-B pré-fixada* (v. 2.6).

Na *árvore-Bit*, a técnica do bit de distinção só detecta a posição correta quando o registro sendo procurado existe no arquivo. Por essa razão, os bits de distinção são usados apenas nos nós folhas, uma vez que a busca de um certo valor i não existente na região de índice acarretaria uma catástrofe.

O bit de distinção é definido como sendo o número ordinal do bit mais significativo diferente dentre os bits de 2 valores do índice consecutivos em um nó folha, acrescido de um valor pré-determinado como “bias”. O objetivo deste “bias” é evitar que o bit de distinção assuma o valor zero.

Considere i_j e i_{j+1} 2 valores consecutivos em um nó folha. Seja $D(i_j, i_{j+1})$ a representação do bit de distinção entre i_j e i_{j+1} , a qual é definida formalmente como sendo:

$$D(i_j, i_{j+1}) = b + m - 1 - \lfloor \log_2(i_j \oplus i_{j+1}) \rfloor$$

onde:

b = bias;

m = tamanho do valor do índice em bits;

$\oplus$ - ou exclusivo aplicado aos bits dos valores do índice;

Exemplo

Seja $i_j = K$ e $i_{j+1} = P$ cujas representações em EBCDIC são dadas abaixo, $b=8$ e $m=8$. Assim,

$i_j = K$	1	1	0	1	0	0	1	0
$i_{j+1} = P$	1	1	0	1	0	1	1	1
Num. dos bits	0	1	2	3	4	5	6	7
Num. com bias	8	9	10	11	12	13	14	15

$$i_j \oplus i_{j+1} = 0000101 = 5$$

$$\lfloor \log_2(i_j \oplus i_{j+1}) \rfloor = \lfloor \log_2 5 \rfloor = 2$$

$$D(i_j, i_{j+1}) = 8 + 8 - 1 - 2 = 13$$

Portanto, o bit mais significativo diferente em i_j e i_{j+1} é o bit de número $7 - 2 = 5$ (com bias, igual a 13). Notemos que o resultado seria o mesmo independentemente dos 2 últimos bits de i_j e i_{j+1} .

Considere o nó folha $n : p_0, i_0, p_1, i_1, \dots, p_m, i_m$, que aponta para $m + 1$ registros de dados. Nas *árvores-Bit* este nó seria representado como na figura 2.16, onde $D_j = D(i_{j-1}, i_j), 1 \leq j \leq m$ (bit de distinção). Como os valores do índice dentro de um nó são organizados em ordem ascendente e D_j representa o bit de distinção mais significativo entre i_{j-1} e i_j , então o bit D_j em i_{j-1} é igual a 0 e em i_j igual a 1.

Figura 2.16: Estrutura de um nó folha n da *árvore-Bit*.

A busca de um registro com valor i em um nó folha n é executado da seguinte maneira. Considere $\mathcal{P}$ uma variável auxiliar, a qual é atribuída inicialmente o valor de p_0 (ponteiro para o primeiro registro referenciado em n). O procedimento verifica para todo D_j , $j = 1, \dots, m$, o bit da posição D_j em i . Se este bit for igual a 1, então $\mathcal{P}$ recebe o valor de p_j e o procedimento continua comparando os outros D_j 's. Se o bit da posição de algum D_j em i for igual a zero, o processo procura no intervalo de $[D_{j+1}, D_m]$ um bit de distinção que contém valor menor do que D_j , pois os que possuem valores maiores ou iguais a D_j também contêm o bit 1 na posição D_j . Na existência desse menor valor, o procedimento continua a comparação a partir do mesmo, isto é, se for igual a 1 atualiza-se $\mathcal{P}$, senão procura-se outro elemento com menor valor. Na realidade o procedimento objetiva encontrar o maior valor de j para o qual a posição do bit D_j em i seja igual a 1. Quando o processo atinge o final do nó, o valor i é comparado com o valor do índice do registro apontado por $\mathcal{P}$. Se forem iguais, então o registro foi localizado; caso contrário, pode-se afirmar que o registro não pertence ao arquivo.

Quando o registro foi localizado e a operação a ser executada é de recuperação ou atualização, então a mesma é simplesmente realizada. Quando a operação for de remoção, o nó folha n precisa ser modificado. Neste caso, elimina-se p_j de n e atualizam-se os valores de D_j e D_{j+1} . Estes são substituídos pelo bit de distinção entre os valores i_{j-1} e i_{j+1} , o qual consiste do menor valor entre D_j e D_{j+1} .

Considere o caso em que se deseje encontrar todos os registros de dados com valores do índice maiores ou iguais a um certo valor i e não há registro com esse valor no arquivo. Seja i_j o valor encontrado na busca e $D' = D(i, i_j)$. Note-se que não existe l tal que $j < l \leq m$ e a posição D_l em i seja igual a 1 pois neste caso o algoritmo escolheria i_l ao invés de i_j . Um dos casos adiante pode ocorrer.

Caso 1: Se $i > i_j$

Como i está à direita de i_j , então o bit da posição D' em i_j é igual a 0. O procedimento verifica as outras entradas a partir de D_j , ignorando aquelas cujos bits de

distinção são maiores do que D' . Para estas o bit da posição D' também é igual a 0 e conseqüentemente menor do que i . O registro desejado é o primeiro que contém o bit de distinção menor do que D' . Note-se que não existe bit de distinção no intervalo de $(D_j, D_m]$ igual a D' pois, se existisse, i_j não seria selecionado.

Caso 2: Se $i < i_j$

Como i está à esquerda de i_j , então o bit da posição D' em i é igual a 0. O procedimento verifica as entradas começando de D_j , no sentido decrescente, isto é, $D_j, D_{j-1}, \dots, D_0$, até atingir o primeiro l , $0 \leq l < j$ tal que $D_l < D'$. Porém, como o bit da posição D' em i_j é igual a 1 e D_l é o primeiro valor encontrado menor do que D' , então todos os valores no intervalo $[D_l, D_j]$ têm o bit 1 na posição D' ($i < i_l < i_j$). Por sua vez, o bit D_l é igual a 0 em i_{l-1} e, pela escolha de i_j , igual a 1 em i , logo $i_{l-1} < i$. O registro de dados que contém o valor i_l é o desejado.

Quando a operação for de inclusão de um novo registro no arquivo, o algoritmo é igual àquele executado para a recuperação de um registro com valor do índice maior ou igual a um certo valor i . Ao ser localizado o valor do índice imediatamente maior, o processo insere nessa posição o valor computado para D' junto com o ponteiro para o novo registro. No caso em que a inserção provocar uma sobrecarga do nó, o mesmo é desdobrado utilizando-se uma técnica similar à empregada nas *árvores-B pré-fixadas simples*, ou seja, o nó é dividido em uma posição que minimize o tamanho do valor do índice a ser incluído no nó pai. Para as *árvores-Bit* este processo não é tão óbvio, pois os valores do índice nos nós folhas não são visíveis. Assim, para evitar a leitura de todos os dados do arquivo para determinar aquele que tenha o menor tamanho, o procedimento escolhe como o melhor valor, dentre os candidatos a serem inseridos no nó pai, o elemento i_j tal que D_j seja mínimo.

2.13 Outras Variações

Para complementar o assunto estudado, abordaremos de forma sucinta outros tipos de árvores balanceadas que podem ser utilizadas como alternativa para as *árvores-B*.

2.13.1 Árvore- M (Multiway Tree)

Arnow e Tanenbaum [2] introduziram a *árvore- M* de ordem K , que é uma árvore não perfeitamente balanceada contendo 3 tipos de nós.

nós *folhas*: não possuem filhos;

nós *semi-folhas*: contêm m valores do índice e podem conter menos do que $m + 1$ filhos ($K \leq m \leq 2K$);

nós *internos*: contêm m valores e $m + 1$ filhos ($K \leq m \leq 2K$);

Um nó com $2K$ valores do índice e nenhum filho é considerado uma folha e uma semi-folha ao mesmo tempo. Nas *árvores-M* os valores do índice dentro de um nó são ordenados. A figura 2.17 ilustra um exemplo de *árvore-M* com $K = 2$ e os 3 tipos de nós.

Figura 2.17: Exemplo de uma *árvore-M* de ordem $K = 2$.

Para inserir um valor i do índice na *árvore-M*, realiza-se uma busca para localizar o nó folha ou semi-folha apropriado. No caso de um nó folha com menos do que $2K$ valores, i é inserido naturalmente. Para a situação de uma semi-folha, cria-se um novo nó folha n_f , filho dessa semi-folha, e inclui-se i em n_f . O procedimento de inserção pode causar a mudança de um nó semi-folha para um nó interno ou de um nó folha para um nó semi-folha. A figura 2.18 ilustra a transformação do nó semi-folha 2 da figura 2.17 em um nó interno, após a inserção do valor 16. A figura 2.19 ilustra a transformação do nó 6 em uma folha e semi-folha, após a inserção do valor 95.

Arnow e Tanenbaum realizaram uma comparação empírica entre a *árvore-M* e a *árvore-B* e chegaram aos seguintes resultados.

1. De forma geral, a *árvore-M* possui melhor tempo nas operações de buscas do que a *árvore-B*. Entretanto, em algumas situações particulares esse tempo pode ser melhor nas *árvores-B*.
2. A *árvore-M* tem fator de ocupação de espaço fortemente dependente da ordem K da árvore.
3. As operações de inserções nas

Figura 2.18: Inserção do valor 16.

Figura 2.19: Inserção do valor 95.

2.13.2 Árvore- T_B

Objetivando garantir um bom desempenho nas buscas realizadas dentro de um nó, Kwong e Wood [49] propuseram em 1980, uma estrutura que permite armazenar uma grande quantidade de dados, chamada de *árvore- T* . Esta árvore é uma *árvore de árvores*, isto é, os nós são organizados como árvores.

Seja $f : N \rightarrow N$ uma função tal que $2 \leq f(m) \leq \lceil \frac{m}{2} \rceil$, $m \geq 3$. Kwong e Wood definem uma *árvore- $f(m)$* como uma generalização da *árvore- B* , onde o número mínimo de filhos em um nó interno, exceto a raiz, é igual a $f(m)$.

Uma *árvore- T* de ordem m é uma *árvore- $f(m)$* onde todos os nós são representados como algum tipo de árvore balanceada. Dessa forma, uma *árvore- T_B* de ordem (m, K) é uma *árvore- T* de ordem m , onde os nós são organizados como uma *árvore- B* de ordem

K . Nos nós folhas de uma *árvore-B*, os ponteiros para outros nós são indefinidos. Para as *árvores- T_B* esses ponteiros, dentro de cada nó n_T da árvore, fazem referência aos filhos de n_T .

Os autores afirmam que com esse tipo de estrutura é possível realizar as operações dentro de um nó em tempo logarítmico ao tamanho do nó, além de permitir concorrência nestas operações.

2.13.3 Árvores para Campos Múltiplos

Nas *árvores-B* as buscas são feitas usando-se valores de campos simples ou compostos. Se um campo é composto, mas deseja-se também realizar buscas independentes em cada um de seus componentes, seria necessário usar uma *árvore-B* para cada componente. Esse tipo de campo composto é denominado de *campo múltiplo* (“multiattribute”). Um arquivo com um campo múltiplo pode ser considerado como sendo organizado em um espaço de várias dimensões, uma para cada componente. Uma consulta pode significar a definição de um sub-espaço.

De acordo com a resenha apresentada por Bentley e Friedamn [14], para solucionar esse problema foram propostas várias estruturas de dados, tais como: *árvore-Quad* (“Quad-Tree”) [30], *árvore-KD* (“KD-Tree”) [13], *árvore-BD* (“BD-Tree”) [76], *K-D Trie* [77], além de estruturas que apresentam semelhanças com a *árvore-B*: *árvore-K-D-B* (“K-D-B-Tree”) [80] e *árvore-hB* (“hB-Tree”) [61]¹².

Para uma melhor compreensão da *árvore-hB* daremos uma sucinta explicação sobre a *árvore-KD*.

Árvore-KD

A *árvore-KD* foi desenvolvida em 1975, por Bentley [13] para armazenar registros de dados com K campos em um espaço K -dimensional e permitir buscas com campos múltiplos. Ela consiste de uma generalização da árvore de busca binária unidimensional.

Um arquivo representado como uma *árvore-KD* possui cada registro armazenado como um nó da árvore. Cada nó contém os K campos do registro e 2 ponteiros: *filho-alto*, *filho-baixo*. Esses ponteiros fazem referência a outros nós da árvore ou são indefinidos. Associado a cada nó n da árvore existe um discriminador $desc(n)$ cujo conteúdo é igual a

¹²Em virtude da não localização do artigo [80], devido a referências erradas em [61] e [77], limitamos nosso estudo à *árvore-hB*.

um valor inteiro entre 0 e $K - 1$, isto é, um dos K campos. A raiz possui discriminador 0, seus filhos imediatos discriminador 1, os nós do K -ésimo nível têm discriminador $K - 1$, os nós do nível $(K + 1)$ possuem discriminador 0 causando uma repetição do ciclo. Todos os nós em um nível da árvore têm o mesmo discriminador. A *árvore-KD* tem seus elementos organizados em uma determinada ordem, conforme descrito a seguir.

Considere n um nó da árvore com K campos, $\{c_0(n), c_1(n), \dots, c_{K-1}(n)\}$; $\text{filho-alto}(n)$ e $\text{filho-baixo}(n)$ seus dois ponteiros e $\text{desc}(n)$ seu discriminador igual a j , $0 \leq j \leq K - 1$. A ordenação é realizada de tal forma que para qualquer nó n' apontado por $\text{filho-baixo}(n)$, $c_j(n') < c_j(n)$ e para qualquer nó n'' apontado por $\text{filho-alto}(n)$, $c_j(n'') > c_j(n)$. Em caso de igualdade dos campos, a ordenação é feita comparando-se os próximos campos de maneira cíclica, isto é, $c_{j+1}, c_{j+2}, \dots, c_{K-1}, c_0, \dots, c_{j-1}$.

A figura 2.20 ilustra um exemplo dado por Bentley, de registros com 2 campos (c_0, c_1), onde (a) corresponde à estrutura em um espaço bidimensional (representação planar) em que as linhas rotuladas que dividem o espaço são linhas discriminadoras; (b) mostra a representação da respectiva *árvore-2D*. Nesse exemplo o nó B é filho-baixo de A, pois $10 = c_0(B) < c_0(A) = 50$; C é filho-alto de A, pois $80 = c_0(C) > c_0(A) = 50$. No próximo nível usa-se c_1 , $20 = c_1(D) < c_1(B) = 70$. O nível seguinte volta a usar c_0 e assim por diante.

Figura 2.20: Representação de uma *árvore-2D*.

Bentley menciona 4 tipos de buscas que explicaremos através do exemplo da figura 2.20.

1. Buscas exatas

Corresponde a se buscar um ponto, dadas as suas coordenadas. Essa busca é trivial

e não motiva o uso da estrutura, pois os campos podem ser concatenados formando uma estrutura unidimensional.

2. Buscas parciais

Corresponde a se tentar localizar uma região de pontos dadas apenas algumas das coordenadas. Por exemplo, os pontos de coordenadas $(10, *)$. Bentley propõe um algoritmo em que ao se atingir o nível de uma coordenada não especificada, ambos os ramos são percorridos.

3. Busca por regiões

Nesta busca determina-se uma região compreendida entre coordenadas dadas. Por exemplo: $(25-70, 20-60)$. Quando se atinge um ponto dentro da região procurada, toma-se os 2 ramos, como no caso do ponto A.

4. Busca de pontos mais próximos

O algoritmo para esta busca não foi explicado, apenas referenciado.

Árvore-hB

Na *árvore-hB*, proposta por Lomet em 1990 [61], os nós representam regiões retangulares do espaço K dimensional, chamadas de *bricks*, das quais podem ser removidas regiões menores originando os *holey bricks* que denominaremos de *elementos vazados* (retangulares), sendo esta a origem do nome (*hB-Tree* ou *holey brick B-tree*). A motivação desta estrutura surgiu do problema existente nas *árvores-K-D-B*, nas quais o desdobramento de um nó pode acarretar desdobramentos encadeados de nós descendentes deste. Isto encarece o processo e aumenta o fator de ocupação de espaço. Nas *árvores-hB* as regiões resultantes de desdobramentos não são retângulos, mas elementos retangulares vazados. A figura 2.21 dada em [61] representa um elemento vazado através de uma *árvore-KD*, onde 2 folhas estão referenciando a região do elemento denotado por B.

Figura 2.21: Representação de um elemento vazado.

A *árvore-hB* é similar à *árvore-B** no sentido de possuir nós folhas com registros de dados e nós internos com registros de índice, com vários valores do índice. Estes valores

também podem ser redundantes, isto é, já existentes nos nós folhas, entretanto eles abrangem um espaço K -dimensional e unidimensional. Os nós são organizados internamente como *árvores-KD*. Lomet escolheu as *árvores-KD* em função delas apresentarem vantagens em espaço e velocidade de consulta, em relação à outra técnica mencionada, que consiste em representar as fronteiras das regiões em listas.

Seja N_{hB} um nó da *árvore-hB*, organizado como uma *árvore-KD*, que chamaremos de $\mathcal{A}_{KD}$. As *árvores-KD* internas a um nó da *árvore-hB* contêm 4 tipos diferentes de nós: nó interno (n_{KD}); nó folha que faz referência aos registros de dados (no caso em que N_{hB} é uma folha); nó folha que aponta para outro nó da *árvore-hB* e um *marcador* que indica que uma parte de $\mathcal{A}_{KD}$ foi removida. Note-se que os nós folhas de uma *árvore-hB* também contêm uma *árvore-KD*, além dos registros de dados.

A busca de um elemento na *árvore-hB* é realizada percorrendo-se a árvore da raiz até um nó folha. As *árvores-KD* dentro dos nós internos servem para dirigir a busca para o próximo nível. No caso de uma busca de intervalo, isto é, uma consulta para um grupo de valores, quando o conteúdo de um nó n_{KD} está contido dentro do intervalo de consulta, o procedimento segue pelos 2 ponteiros (*filho-alto*, *filho-baixo*).

Do ponto de vista global as inserções e atualizações nas *árvores-hB* são realizadas da mesma forma que nas *árvores-B**. Entretanto, em uma visão mais detalhada elas diferem na maneira em que os nós são desdobrados e como um valor do índice é inserido em um nível acima.

Os desdobramentos são realizados de tal forma a garantir que cada novo nó da *árvore-hB* gerado por ele contenha entre $\frac{1}{3}$ e $\frac{2}{3}$ do conteúdo do nó original. A região removida é denominada de *região extraída* e a restante de *região permanente*. Para melhor entendimento do procedimento de desdobramento considere:

N'_{hB} = um nó sobrecarregado da *árvore-hB* (armazena registros de dados ou valores do índice);

$\mathcal{A}'_{KD}$ = *árvore-KD* interna a N'_{hB} ;

n'_{KD} = nó pertencente a $\mathcal{A}'_{KD}$;

$N'_{hB}(extr)$ = nó da *árvore-hB* contendo informações da região extraída (ocupa uma nova área do disco);

$N'_{hB}(perm)$ = nó da *árvore-hB* contendo informações da região permanente (ocupa a mesma área do disco que o nó original);

Observam-se 3 casos distintos.

Caso 1: O desdobramento de N'_{hB} ocorre na raiz de $\mathcal{A}'_{KD}$.

Neste caso, a sub-árvore à direita da raiz forma o nó $N'_{hB}(extr)$, enquanto que o

restante $N'_{hB}(perm)$.

Caso 2: O desdobramento de N'_{hB} ocorre em um nó n'_{KD} diferente da raiz.

Quando a raiz de $\mathcal{A}'_{KD}$ não divide o conteúdo entre $\frac{1}{3}$ e $\frac{2}{3}$ dos valores, então caminha-se por $\mathcal{A}'_{KD}$ até encontrar o nó n'_{KD} apropriado.

Caso 2.1: n'_{KD} não é um nó folha.

Como no caso 1, coloca-se uma parte de $\mathcal{A}'_{KD}$ em $N'_{hB}(extr)$ e o restante em $N'_{hB}(perm)$. Na parte de $\mathcal{A}'_{KD}$ pertencente a $N'_{hB}(perm)$ coloca-se um nó *marcador* para indicar que uma sub-árvore foi extraída.

Caso 2.2: n'_{KD} é um nó folha.

Esta situação ocorre em geral quando N'_{hB} é um nó folha. Algumas vezes, o procedimento não consegue gerar nós possuindo entre $\frac{1}{3}$ e $\frac{2}{3}$ do conteúdo do nó original, pois os nós folhas da *árvore-hB* possuem registros de dados além da *árvore-KD*.

Após o desdobramento de N'_{hB} é necessário indicar ao seu respectivo pai que ele foi dividido em $N'_{hB}(extr)$ e $N'_{hB}(perm)$. Na realidade é preciso informar o espaço K -dimensional ocupado pela região extraída, a qual representa um “brick”. Muitas vezes, alguns dos valores que limitam esta região já existem nos ancestrais de N'_{hB} , não sendo necessário incluí-los. Todos os possíveis valores estão no caminho que vai da antiga raiz de $\mathcal{A}'_{KD}$ até a raiz do novo nó $N'_{hB}(extr)$ a ser gerado. Resta escolher os nós deste caminho que realmente identificam $N'_{hB}(extr)$. Quanto ao procedimento de remoção, Lomet não faz referência.

Capítulo 3

TÓPICOS ESPECIAIS

Neste capítulo abordaremos alguns tópicos especiais como: técnicas de compressão de índices, algoritmos de compactação para as *árvores-B*, algoritmos que permitem uma organização “ótima” para as árvores que possuem elementos de tamanho variável, métodos que possibilitam o uso das *árvores-B* em sistemas com várias versões e o uso em sistemas com vários discos.

3.1 Compressão de Índices

Além da *árvore-B pré-fixada* (v. 2.6) e da *árvore-Bil* (v. 2.12), existem outras formas de compressão de índices, algumas das quais serão abordadas a seguir.

Em 1973, Wagner [93] sugeriu uma técnica de endereçamento relativo aplicada às situações em que um índice aponta para um conjunto contíguo de registros. Nesses casos, ao invés de se usar os endereços absolutos para cada registro usam-se deslocamentos de endereços em relação a uma base, a qual pode ser a origem do arquivo de índice ou de dados. A figura 3.1 ilustra o formato de um nó quando se usa essa técnica. Para se calcular o endereço do registro referenciado por end_j soma-se o valor de end_j com a $base$, ($1 \leq j \leq m$).

$base$	i_1	end_1	i_2	end_2	$\cdots$	i_m	end_m
--------	-------	---------	-------	---------	----------	-------	---------

Figura 3.1: Formato de um nó com endereçamento relativo.

Wagner também fez menção a outra técnica denominada de *compressão binária* que ele diz ter sido desenvolvida por L.J. Woodrum, a qual se aplica aos valores de índices. Esse método caracteriza-se por armazenar os valores do índice através de uma fatoração dos seus bits, organizando-os em uma árvore binária. Para melhor compreensão considere os seguintes valores:

$$\begin{array}{llll} i_1 = 0001 & i_2 = 0011 & i_3 = 0100 & i_4 = 0110 \\ i_5 = 1001 & i_6 = 1100 & i_7 = 1110 & i_8 = 1111 \end{array}$$

O procedimento consiste em dividir esses valores em 2 grupos baseando-se no valor do primeiro bit (da esquerda para direita). Em seguida, divide-se cada um desses sub-grupos, em novos grupos, com base no valor do segundo bit. O processo continua até que cada sub-grupo contenha um único elemento. Esses sub-grupos são organizados como uma árvore binária, não necessariamente balanceada, onde cada folha indica o valor de um dos i_j elementos ($1 \leq j \leq 8$, neste exemplo). Cada nível l da árvore representa um dos bits sendo comparado. Em um nó n do nível l , a sub-árvore à esquerda de n contém o conjunto dos elementos com valor 0 no bit l e a sub-árvore à direita contém o conjunto dos elementos com o valor 1 no bit l . A figura 3.2 ilustra a árvore para os valores dados; ao lado de cada nó colocamos o intervalo dos índices dos elementos abrangidos - isso é possível pois, se $j > k$ então $i_j > i_k$.

Figura 3.2: Árvore binária para os valores i_j , $1 \leq j \leq 8$.

O índice teria o seguinte formato.

Grupos	Posição do bit	Ponteiro 0	Ponteiro 1
$G1$	1	$G2$	$G3$
$G2$	2	$G4$	$G5$
$G3$	2	i_5	$G6$
$G4$	3	i_1	i_2
$G5$	3	i_3	i_4
$G6$	3	i_6	$G7$
$G7$	4	i_7	i_8

Notemos que a aplicação desse método nas *árvores-B* não acarreta uma grande economia de espaço quando os valores têm tamanho pequeno. Ao invés de existir o par (valor, ponteiro) para cada entrada haverá uma tripla (posição do bit, ponteiro para o valor 0, ponteiro para o valor 1), onde esses ponteiros referenciam outras entradas no nó, ou os registros associados aos valores.

Além do método de compressão binária existem as técnicas de compressão de caracteres: *compressão frontal* e *posterior*, empregadas para diminuir o tamanho dos valores de índices.

A compressão frontal caracteriza-se por eliminar os prefixos comuns entre valores de índices adjacentes. Por exemplo, sejam $i_1 = 12345$ e $i_2 = 12356$ dois valores adjacentes em um nó. Aplicando-se a compressão frontal, os caracteres 123 serão retirados de i_2 , sendo apenas armazenado o valor 56. Por outro lado, a compressão posterior elimina os caracteres insignificantes, do ponto de vista de busca na árvore, do final de um determinado valor do índice. O procedimento compara o valor i do índice em um nó com o valor do índice à direita de i , retirando de i os caracteres à direita do primeiro caractere diferente. Por exemplo, considere $i_1 = 11800$, $i_2 = 11810$ e $i_3 = 11900$ 3 valores adjacentes em um nó. O uso da compressão posterior abreviaria i_1 e i_2 para $i'_1 = 1180$ e $i'_2 = 118$, respectivamente. Note-se que a compressão posterior, diferentemente da frontal, não permite que se determine o valor exato do índice. No caso das *árvores-B** é preciso atingir as folhas para determinar o valor do índice. Observamos que o uso da compressão posterior só se aplica para cadeias de caracteres e números com tamanho fixo.

O sistema VSAM da IBM [37, 38, 39], a ser abordado no capítulo 5, utiliza essas 2 técnicas de compressão de caracteres, de forma combinada.

Por sua vez, o sistema MUMPS desenvolvido pela MEDIDATA [65, 66], que utiliza a estrutura de *árvore-B** para organizar e manter os arquivos, emprega a técnica de endereçamento relativo e de compressão frontal dos valores do índice. Esse sistema adota uma

organização hierárquica em que os valores dos índices são campos compostos, formando um caminho para o nó (página) que deverá conter a informação desejada.

O endereçamento relativo é realizado nos nós da região de índices, reduzindo o espaço ocupado pelos ponteiros para uma palavra de 2 bytes (2 caracteres), quando normalmente seria de duas palavras. Esses ponteiros são endereçados em relação a um nó particular, denominado de *nó-diretório*, que aponta para a raiz do arquivo.

Para possibilitar a compressão dos valores do índice, cada valor contém uma palavra de controle associada indicando se o valor é (1) ou não (0) comprimido, o número de palavras comprimidas (NPC), isto é, a quantidade de palavras iguais (prefixos) ao valor anterior e o número de palavras armazenadas (NPA) com o restante da informação. As palavras que identificam o tipo dos valores do índice (inteiro, cadeia de caracteres, decimal-compactado, etc.) também são compactadas. Neste caso, quando existir 2 ou mais valores adjacentes dentro de um nó, de mesmo tipo, a palavra que indica este tipo aparece apenas para o primeiro valor. Notemos que o fato dos campos serem compostos justifica o uso da compressão frontal, pois nesses casos é grande a quantidade de informações comuns nos valores do índice em um nó. A seguir daremos um exemplo do uso da compressão frontal no sistema MUMPS.

Exemplo

$i_1 = Casa$	$i_2 = Casinha$	$i_3 = Casinhola$
$i_4 = Casinhola, 1$	$i_5 = Dado$	$i_6 = Dado, 1$

Sem compactação o número de palavras ocupadas por esses valores seria como mostrado na tabela a seguir, onde (controle) significa a palavra de controle e (tipo valor) a palavra contendo o tipo do valor.

Valor	Número de palavras
i_1	4 = 1(controle) + 1(tipo valor) + 2(valor)
i_2	6 = 1(controle) + 1(tipo valor) + 4(valor)
i_3	7 = 1(controle) + 1(tipo valor) + 5(valor)
i_4	9 = 1(controle) + 1(tipo valor1) + 5(valor1) + 1(tipo valor2) + 1(valor2)
i_5	4 = 1(controle) + 1(tipo valor) + 2(valor)
i_6	6 = 1(controle) + 1(tipo valor1) + 2(valor1) + 1(tipo valor2) + 1(valor2)
Total	36 palavras

Usando a técnica de compactação tem-se que:

Valor	NPC	NPA
i_1	0	4
i_2	$2 = 1(\text{tipo valor}) + 1(\text{Ca})$	4
i_3	$4 = 1(\text{tipo valor}) + 3(\text{Casinh})$	3
i_4	$6 = 1(\text{tipo valor}) + 5(\text{Casinhola})$	3
i_5	$1 = 1(\text{tipo valor})$	3
i_6	$3 = 1(\text{tipo valor}) + 2(\text{Dado})$	3
i_7	16	20

Para melhor compreensão vejamos como seriam alocados os elementos i_1 , i_2 e i_3 em um nó, onde cada palavra está delimitada por || e T representa o tipo do valor. Note-se que para i_2 e i_3 não há necessidade de se armazenar os seus tipos (T), pois estes são iguais ao de i_1 .

0	0	4	T	Ca	sa	1	2	4	si	nh	a	1	4	3	ol	a	...
---	---	---	---	----	----	---	---	---	----	----	---	---	---	---	----	---	-----

3.2 Compactação de Árvores-B

3.2.1 Método de Rosenberg e Snyder

Em virtude das características da *árvore-B*, Rosenberg e Snyder [81, 82] afirmaram em 1979 e 1981, que as mesmas permitem várias representações diferentes não-isomórficas para um mesmo arquivo, isto é, um arquivo com I valores do índice pode ter mais do que uma *árvore-B* de uma mesma ordem K armazenando estes I elementos. Seja $\mathfrak{S}_I^K$ a floresta de todas as *árvores-B* de ordem K contendo I valores. A diferença entre cada árvore pertencente à floresta $\mathfrak{S}_I^K$ é medida pelos custos de tempo, isto é, número total de nós visitados para referenciar um registro de dados com um dos valores do índice e de espaço, a quantidade total de nós internos. O custo de espaço é reduzido através da diminuição do espaço desperdiçado, aumentando a quantidade de valores em cada nó.

Uma boa *árvore-B* é aquela que diminui ambos os custos. Rosenberg e Snyder afirmam que na prática isso nem sempre é possível. Seus estudos indicam que:

Se uma árvore é ótima em espaço, então ela é quase ótima em tempo; mas se é ótima em tempo, não necessariamente é ótima em espaço.

Para conseguir uma diminuição no custo de espaço eles sugeriram uma compactação da árvore, a qual pode ser realizada periodicamente de maneira concorrente com as operações periódicas de back-ups.

O algoritmo de compactação proposto por Rosenberg e Snyder¹, consiste de um esquema de reestruturação *in situ*, permitindo que a nova árvore seja construída na mesma área da antiga, requerendo apenas o espaço adicional de h nós livres, onde h é a altura da nova árvore a ser criada. O algoritmo foi apresentado para as *árvores-2-3*. A generalização para as *árvores-B* é simples e será apresentada a seguir.

O algoritmo recebe como entrada a árvore a ser compactada, a altura h da nova árvore compactada e um vetor *perf* com h posições, que contém o perfil detalhado da nova árvore a ser gerada, visto a seguir. A saída é um ponteiro para a raiz da *árvore-B compactada*.

O perfil detalhado consiste do número de nós, em cada nível, com $K, K + 1, \dots, 2K$ valores do índice que proporcionará uma árvore de pouca altura. Ele permite que se escolha o fator de ocupação de espaço de cada nível. Por exemplo, se não houver inserções esse fator deve ser 100%. O número mínimo de nós de cada nível $l, 0 \leq l < h$, de uma *árvore-B* de altura h e ordem K é $N_l = \lceil \frac{N_{l+1}}{2K} \rceil$; o número máximo é $N_l = \max(1, \lfloor \frac{N_{l+1}}{K} \rfloor)$. Para se determinar o perfil detalhado basta observar que em um nível l , os N_l nós devem conter ao todo N_{l+1} filhos, e que um nó com m valores tem $m + 1$ filhos. Os autores propõem uma distribuição dos valores entre N_l nós, de tal forma que r nós contenham m valores e os demais $N_l - r$ nós, $m + 1$ valores, onde: $m = \lfloor \frac{N_{l+1}}{N_l} \rfloor - 1$ e $r = N_l - N_{l+1} \bmod N_l$.

De forma geral, o algoritmo é composto por 2 rotinas básicas *Busca-profe Construção*. A primeira é uma rotina recursiva que realiza uma busca em profundidade na árvore, a partir da raiz até o nível mais baixo, recuperando os valores do índice da mesma. Sempre que um valor for determinado ele é fornecido para a rotina *Construção*, não recursiva, que constrói a árvore compactada. Em qualquer instante do procedimento, a rotina *Construção* referencia um nó da árvore. Com base no perfil detalhado da nova árvore (fornecido como dado de entrada do algoritmo), a rotina *Construção* insere os elementos dentro do nó corrente. Sempre que a construção for iniciada em um novo nível, a rotina *Construção* aloca um novo nó na árvore, tornado este o nó corrente; por isso a necessidade de h nós extras. Em virtude da utilização da busca em profundidade, verifica-se que o algoritmo proposto é de tempo linear no tamanho da árvore.

Arnow e Tanenbaum [2] realizaram um estudo que faz comparações empíricas entre as *árvores-B*, *árvores-B compactadas* de Rosenberg e Snyder e *árvores-M* (v. 2.13.1).

¹Na realidade, estes autores, junto com Miller e Pippenger em [68], já haviam proposto antes um algoritmo para construir uma *árvore-2-3 compactada*. No entanto, esse algoritmo exigia a existência simultânea de 2 cópias da árvore.

Eles concluíram que as *árvores-B compactadas* são vantajosas apenas no contexto de arquivos não-voláteis, isto é, aqueles que não contêm um volume grande de inserções de novos elementos². Por outro lado, no que diz respeito ao tempo de busca, as *árvores-B compactadas* com uma ordem K pequena, são consideradas melhores do que as *árvores-B* e *árvores-M* com o mesmo K . O fator de ocupação de espaço independe do valor de K , variando entre 92% e 100% em suas experiências. Já o custo de inserção nas *árvores-B compactadas* pode chegar a ser 2 vezes maior do que nas *árvores-B* e *árvores-M*, em arquivos grandes.

3.2.2 Algoritmo de Permutação

Seguindo o mesmo intuito de se obter uma *árvore-B* ótima em termos de espaço, Klein, Parzygnat e Tharp propuseram em 1991 [43] um algoritmo para construção da mesma. Este algoritmo recebe dados ordenados pelos seus valores e executa uma permutação dos mesmos organizando-os de forma especial. Essa permutação posiciona os elementos de maneira a permitir que cada um seja inserido na árvore em um determinado nó, após este ter sofrido um desdobramento.

O algoritmo tem como entrada um conjunto I de valores ordenados e a ordem K da árvore. A saída é um conjunto C_e formado pelos elementos de I . O conjunto C_e é usado para construir a *árvore-B* através do algoritmo de inserção tradicional (v. 1.4.2).

Para evitar uma diminuição da ocupação de espaço decorrente de futuras inserções aleatórias, realiza-se periodicamente uma reconstrução da árvore, através da reaplicação do algoritmo, nos horários em que a estrutura não esteja sendo muito utilizada.

A seguir temos o *algoritmo de permutação*. A figura 3.3 ilustra os passos do exemplo, enquanto a figura 3.4 compara 2 *árvores-B* construídas (a) sem e (b) com a aplicação do algoritmo de permutação para a mesma instância de entrada. Note-se que em (b) os elementos C, D, H, I serão inseridos após o desdobramento dos nós onde os mesmos devem residir.

Algoritmo Permutação (I):

1. $M := 2K + 1$;
2. Retire os K primeiros elementos de I e coloque-os em C_e ;

²Rosenberg e Snyder também já haviam observado este fato, através da afirmação de que “as *árvores-B compactadas* são ideais para arquivos estáveis, onde o número de operações de inserções realizadas entre compactações não ultrapassa 10%”.

3. Divida os elementos de I em grupos de M elementos;
4. Se o último grupo não contém $2K + 1$ elementos, ele deve ser completado com repetição de um valor especial diferenciado dos demais (“dummy”);
5. Rearranje cada grupo colocando os K primeiros elementos após os $K + 1$ restantes;
6. Se o número de grupos for maior do que K , então

$M := M * M$;

Vá para 2;

Senão,

Retire os valores especiais acrescentados e coloque os elementos restantes de I em C_e ;

Passo 0:	$I = \{A B C D E F G H I J K L\}$	$C_e = \{\}$
Passo 1:	$I = \{C D E F G H I J K L\}$ $I(G_1) = \{C D E F G\}$	$C_e = \{A B\}$ $I(G_2) = \{H I J K L\}$
Passo 2:	$I = \{E F G C D J K L H I\}$ $I(G_1) = \{E F G C D\}$	$C_e = \{A B\}$ $I(G_2) = \{J K L H I\}$
Passo 3:	$C_e = \{A B E F G C D J K L H I\}$	

Figura 3.3: Ilustração das etapas do algoritmo de permutação ($2K = 4$).

Figura 3.4: Comparação das *árvores-B* construídas (a) sem e (b) com a aplicação do algoritmo.

Notemos que a idéia de Küspert [47] de desdobramento marginal (v. 1.4.2) é melhor do que a apresentada nesta seção, pois também produz uma árvore compactada a partir de

3.3. ORGANIZAÇÃO ÓTIMA DAS ÁRVORES-*B* COM ELEMENTOS DE TAMANHO VARIÁVEL

uma entrada ordenada, sem necessidade de formação dos grupos. Por sua vez, fazendo-se uma comparação do método de Rosenberg e Snyder com o algoritmo visto, temos que este último é mais fácil de ser implementado, pois não requer o acesso direto a cada nó nem a construção da árvore final. Apenas faz uso do algoritmo tradicional de inserção das *árvores-B* que é mais simples.

3.3 Organização Ótima das Árvores-*B* com Elementos de Tamanho Variável

Com o intuito de diminuir a altura da *árvore-B* que utiliza elementos³ com tamanho variável, foram propostos alguns *algoritmos de paginação*. Estes algoritmos distribuem os elementos de uma página (nó) sobrecarregada ou subcarregada em outras páginas, identificando os respectivos separadores. Em 1977, McCreight [64] elaborou um algoritmo de análise local para reorganizar a árvore, em tempo $O(m \log_2 m)$, onde m é o número de elementos a serem distribuídos. Com base neste algoritmo e fazendo uso da técnica de programação dinâmica, Diehr e Faaland sugeriram em 1984 [23], um algoritmo de análise global de mesma complexidade e um outro que requer tempo $O(m^3 \log_2 m)$ para construir uma *árvore-B* de altura mínima.

Uma página da *árvore-B* com registros de tamanho variável é limitada pelo número de palavras⁴ P que a mesma pode conter e não pelo número de elementos; inicialmente este limite varia entre $\frac{P}{2}$ e P palavras. Define-se *scroll* como uma estrutura exatamente igual a uma página, mas com tamanho maior, construído sempre que uma página da árvore se torne sobrecarregada ou ficar com menos do que $\frac{P}{2}$ palavras. Um scroll é composto pelos elementos desta página acrescido do conteúdo de uma ou duas páginas vizinhas e dos elementos da página pai que as referenciam (separadores).

3.3.1 Algoritmo de Análise Local

McCreight desenvolveu este algoritmo com base no princípio de que *para se obter árvores de pouca altura, deve-se ter registros de tamanho pequeno próximos à raiz, e os de tamanho grande junto das folhas*. Este fato depende da distribuição dos registros no conjunto de entrada.

³Como estas técnicas podem ser usadas tanto para os registros de índices ou de dados de tamanho variável, será usado o termo *elemento* para referenciar um item de qualquer um desses registros.

⁴Palavra é a menor unidade endereçável da memória primária.

O método consiste em tentar distribuir os elementos de um scroll de maneira que cada página contenha em média, pelo menos $\frac{2P}{3}$, lembrando a idéia da *árvore- B^O* (v. 2.3). Como podem haver várias paginações possíveis, utiliza-se a heurística de desdobrar o scroll de forma que a soma do comprimento dos elementos separadores a serem inseridos na página pai seja mínima. Esta estratégia foi denominada de *MS - Minimizes the Sum* (minimização da soma). O algoritmo desenvolvido para encontrar este desdobramento local é explicado a seguir.

Quando uma página $\mathcal{P}$ da árvore torna-se sobrecarregada, tenta-se construir um scroll tal que o seu desdobramento não aumente o número de páginas na árvore. Utilizam-se 2 páginas vizinhas a $\mathcal{P}$, $\mathcal{P}$ e os elementos separadores destas. Se o desdobramento ocasionar um aumento no número de páginas, então constrói-se o scroll contendo $\mathcal{P}$, um único vizinho e os respectivos separadores. Este scroll é dividido em 3 páginas (divisão de 2 para 3). Na situação em que uma página $\mathcal{P}$ encontra-se subcarregada, concatenam-se com $\mathcal{P}$ 2 páginas vizinhas e os separadores, tentando desdobrar em menos do que 3 páginas. Se não for possível, a divisão é realizada em exatamente 3 páginas (divisão de 3 para 3).

Verifica-se que o problema se resume em encontrar os separadores que determinam uma paginação possível e contenham uma soma mínima de seus comprimentos. Para isso, McCreight sugeriu o uso de uma fila de prioridade implementada como uma árvore heap, com o elemento de menor tamanho no topo do heap. Para todo elemento separador da esquerda E existe um conjunto de separadores à direita $D(E)$ os quais determinam uma paginação possível junto com E . O menor membro de $D(E)$ é o melhor elemento.

Para um scroll com m elementos e fazendo uso de um heap, determina-se o melhor $D(E)$ em tempo $O(\log_2 m)$. Como este deve ser verificado para todo E , então tem-se que o tempo total é dado por $O(m \log_2 m)$.

McCreight comparou seu algoritmo com outra técnica denominada de *ES - Equal Size* (tamanho igual), na qual o scroll é dividido em páginas com aproximadamente o mesmo tamanho. Ele construiu uma *árvore- B* com 33.381 elementos usando as duas técnicas e constatou que a árvore resultante pela aplicação do método *MS* tinha 3 páginas e os 4 gerados pela técnica *ES*.

3.3.2 Algoritmo de Análise Global

Diehr e Faaland [23] estenderam a idéia do algoritmo de McCreight propondo um algoritmo que determinasse, de forma rápida, uma possível paginação em que a soma do tamanho dos separadores fosse mínima e o número destes separadores superior a 3.

Suponha um scroll com m elementos $e_1, e_2, \dots, e_m$ e $w(j)$ o tamanho (número de

3.3. ORGANIZAÇÃO ÓTIMA DAS ÁRVORES-B COM ELEMENTOS DE TAMANHO VARIÁVE

palavras) de e_j ($1 \leq j \leq m$). Seja $G(V_G, A_G)$ um grafo onde: V_G é formado pelos m elementos do scroll acrescidos de dois novos vértices (e_0, e_{m+1}), tais que $w(o) = w(m+1) = 0$; A_G é constituído pelo conjunto de pares (e_x, e_y) , $x = 0, 1, \dots, m$ e $y = x+1, \dots, m+1$, tal que, $\frac{2P}{3} \leq \sum_{j=x+1}^{y-1} w(j) \leq P$. Assim, verifica-se que todo par $(e_x, e_y) \in A_G$ corresponde a separadores de uma possível paginação.

Dado $G = (V_G, A_G)$, define-se um caminho em G do nó e_0 ao nó e_{m+1} , como uma subsequência $f_0, f_1, \dots, f_{m+1}$ de nós, tal que $e_0 = f_0 < f_1 < \dots < f_{m+1} = e_{m+1}$, onde $(f_x, f_{x+1}) \in A_G$, $x = 0, \dots, m$. Os nós $f_1, \dots, f_m$ do caminho são separadores de uma possível paginação. Considere $\sum_{j=1}^m w[f(j)]$ (soma dos separadores), como o comprimento do caminho. Assim, ao se determinar o caminho de menor comprimento, encontram-se os separadores tal que a soma de seus tamanhos é mínima. O problema resume-se em encontrar o caminho de menor tamanho.

Diehr e Faaland utilizaram programação dinâmica para encontrar a solução desejada. Para tal definiram a função

$c(y)$ = comprimento do menor caminho do nó e_0 ao nó e_y

= soma mínima do tamanho de um conjunto de possíveis separadores

onde a condição inicial é dada por $c(0) = 0$ e a condição recursiva é executada fazendo-se $c(y) = \min\{c(x) + w(x)\}$, tal que $(e_x, e_y) \in A_G$ e $y = 1, \dots, m+1$.

O cálculo de $c(1), c(2), \dots, c(m+1)$ requer tempo $O(m^2)$, entretanto a estrutura do grafo permite um procedimento que gasta tempo $O(m \log_2 m)$ para determinar o menor caminho. A idéia consiste em calcular $c(y+1)$ aproveitando o cálculo de $c(y)$ e utilizando um heap.

Suponha que no início do cálculo de $c(y)$ já se conheça o intervalo $[x', x'']$ correspondente ao domínio do cálculo de $c(y)$, isto é, $\forall x \in [x', x''], (e_x, e_y) \in A_G$. Assim,

1. $x' =$ menor valor de x para o qual $\sum_{j=x+1}^{y-1} w(j) \leq P$;
2. $x'' =$ maior valor de x para o qual $\sum_{j=x+1}^{y-1} w(j) \geq \frac{2P}{3}$;

Após a determinação de x' e x'' , calculam-se $\{c(x) + w(x)\}$ para $x = x', \dots, x''$, colocando estes valores em um heap com o menor valor na raiz. Observe que o tempo para determinar $c(y)$ é constante.

Para o cálculo de $c(y+1)$ considere $soma' = \sum_{j=x'+1}^{y-1} w(j)$, $soma'' = \sum_{j=x''+1}^{y-1} w(j)$ e a seguinte atualização.

$soma' := soma' + w(y)$
 $soma'' := soma'' + w(y)$

 Enquanto $soma' > P$ faça

 Enquanto $soma'' - (w(x'') + 1) \geq \frac{2P}{3}$ faça

 $soma' := soma' - w(x' + 1)$
 $soma'' := soma'' - w(x'')$

 Retire $(c(x') + w(x'))$ do heap

 Insira $(c(x'') + w(x''))$ no heap

 $x' := x' + 1$
 $x'' := x'' + 1$

Fim-enquanto

Fim-enquanto

Note-se que em um heap com m elementos a inserção ou remoção de um elemento leva tempo $O(\log_2 m)$. Como cada $(c(y) + w(y))$ é inserido e removido no máximo uma única vez, então o tempo para calcular os separadores é $O(m \log_2 m)$.

Gagliardi e Merelli [31] utilizaram o algoritmo de Diehr e Faaland em um sistema denominado de DOMINO, que faz uso da *árvore-B* para manusear os dados. O sistema foi implementado em um IBM 3090/150 com VM/CMS, contendo páginas de 512 bytes. Elas compararam o uso do mesmo nas *árvores-B* e *árvores-B⁰* com elementos não ordenados (aleatórios), ordenados de forma ascendente e descendente e chegaram à conclusão de que o desempenho nas *árvores-B⁰* é superior ao das *árvores-B*. Há uma redução de 20% no número de páginas e de 25% na altura da árvore. O uso de elementos desordenados causa um ligeiro aumento no número de páginas, mas requer maior tempo. Não foram identificadas diferenças quando os elementos estão ordenados ascendente ou descendente.

3.3.3 Algoritmo para Construir Uma Árvore de Altura Mínima

Uma *árvore-B* pode ser construída a partir de um scroll formado por todos os valores do índice, da seguinte maneira.

Se todos os elementos do scroll cabem em uma única página, o procedimento está concluído. Caso contrário, identificam-se os elementos separadores tais que a soma do tamanho dos elementos compreendidos entre cada par de separadores seja menor ou igual a P . Esses elementos separadores formarão um outro scroll no nível imediatamente acima, requisitando a reaplicação do procedimento, até o instante em que se determine um scroll com no máximo P elementos. Note-se que outras *árvores-B* podem ser construídas desta maneira, escolhendo-se outros separadores possíveis em cada nível. A partir da construção de todas essas árvores, escolhe-se a de altura mínima.

Diehr e Faaland [23] sugeriram um algoritmo para determinar uma *árvore-B* de altura

3.3. ORGANIZAÇÃO ÓTIMA DAS ÁRVORES-B COM ELEMENTOS DE TAMANHO VARIÁVE

mínima onde não é necessário fazer a enumeração de todas as árvores, sendo suficiente identificar todos os scrolls que podem aparecer em cada nível da árvore.

Considere t um nível da árvore. Define-se $G(t) = (V_{G(t)}, A_{G(t)})$ como o grafo do nível t em que: $V_{G(t)}$ contém todos os m elementos do scroll do nível t , $(e_1^t, e_2^t, \dots, e_m^t)$, acrescido de (e_0^t, e_{m+1}^t) ; $A_{G(t)}$ é formado pelo conjunto de pares de vértices (e_x^t, e_y^t) , $x = 0, 1, \dots, m$, $y = x + 1, \dots, m + 1$, tal que $c(t-1; x, y) \leq P$. A função $c(t; x, y)$ corresponde ao menor caminho entre os nós e_x^t, e_y^t em $G(t)$. Ou seja, os elementos que aparecem em um determinado nível t da árvore, são elementos promovidos do nível $(t-1)$, onde o menor caminho entre estes elementos no nível $(t-1)$ não ultrapassa P .

Para o nível $t = 1$, tem-se que $c(1; x, y) = \sum_{j=x+1}^{y-1} w(j)$, para $x = 0, \dots, m$ e $y = x + 1, \dots, m + 1$, onde $w(j)$ é igual ao tamanho de e_j ($1 \leq j \leq m$). A partir desse ponto o procedimento executa o trecho abaixo:

Enquanto $c(t; 0, m + 1) > P$, faça

$t := t + 1$;

Determine os elementos de $A_{G(t)}$;

Dado $A_{G(t)}$, $\forall (e_x^t, e_y^t)$ calcule

$$c(t; x, y) = \min\{c(t; x, k) + w(k)\}, \forall k, \text{ tal que } x < k < y \text{ e } (e_k, e_y) \in A_{G(t)};$$

Fim-enquanto

Quando a condição $c(t; 0, m + 1) \leq P$ for verdadeira, o trecho acima é encerrado, concluindo-se que a árvore tem t níveis.

Diehr e Faaland afirmam que a estrutura de heap só deve ser usada no nível das folhas da árvore, pois neste nível cada elemento é inserido ou removido no máximo 1 vez, fato que não se verifica para os demais níveis. O algoritmo gasta tempo $O(m^3)$ para computar todos os $c(t; x, y)$ em cada iteração de t . Como a árvore tem no máximo $\log_2 m$ níveis, então o tempo total para construir a árvore de profundidade mínima é $O(m^3 \log_2 m)$.

O método apresenta como desvantagem o fato de não garantir que as páginas não estão subcarregadas, podendo acarretar uma baixa ocupação de espaço. Outro ponto negativo consiste do tempo de execução, tornando-o impraticável para a maioria das aplicações. Entretanto, para arquivos considerados estáticos, pode-se justificar o uso do algoritmo.

3.4 Sistemas de Versões para as Árvore- B^+

Em 1989, Driscoll, Sarnak, Sleator e Tarjan [25] denominaram de *estruturas de dados efêmeras* aquelas que ao sofrerem mudanças em suas informações destroem as antigas, guardando apenas a mais recente, isto é, a versão mais atualizada. Todavia, existem aplicações nas áreas de geometria computacional, editoração de texto e arquivo, e programas em linguagens de programação de alto nível, que necessitam manter as várias versões de seus dados. As estruturas que permitem o acesso e o armazenamento de todas as versões foram chamadas por esses autores de *persistentes*, sendo de *persistência parcial* (“partially persistent”) quando todas as versões podem ser obtidas, mas apenas a mais nova é modificada, e de *persistência total* (“fully persistent”) quando todas as versões são obtidas e modificadas. Eles desenvolveram o denominado *método do nó gordo* que possibilita o armazenamento de várias versões para estruturas de dados ligadas. Baseando-se nesse método Lanka e Mays [52] desenvolveram, em 1991, outras técnicas para tornar as *árvores- B^+* com persistência total.

3.4.1 Método do Nó Gordo Extendido para as Árvore- B^+

Lanka e Mays apresentaram esta extensão, utilizando uma estrutura especial denominada de *nó gordo* (“fat-node”). A seguir daremos algumas definições que serão úteis na descrição do método.

Nó gordo: É um nó composto de um *bloco de versão* e de um ou mais *blocos de índices*.

Bloco de Versão: É uma lista de pares ordenados (v, p) , onde v é um identificador de versão e p um ponteiro para um bloco de índice.

Bloco de Índice: Corresponde a um nó da *árvore- B^+* tradicional, com ponteiros e valores do índice $(p_0, i_1, p_1, \dots, p_{m-1}, i_m, p_m)$, $K \leq m \leq 2K$. Os ponteiros fazem referência para outros nós gordos ou para registros de dados. Dentro de um nó gordo, uma entrada do bloco de versão junto com o bloco de índice associado corresponde a uma versão do nó. Quando uma nova versão é criada, apenas os nós gordos que sofreram modificações são atualizados, ou seja, a representação de uma *árvore- B^+* de uma certa versão reapresenta os nós existentes em versões anteriores e que não sofreram mudanças. Observe-se que não é necessário criar uma nova versão para cada atualização, isto é, uma versão pode incorporar um conjunto de atualizações. A figura 3.5 ilustra (a) a estrutura dos nós gordos e (b) a representação completa de uma *árvore- B^+* com persistência total (com 2 versões).

Grafo de Versão: O método faz uso de um grafo enraizado dirigido acíclico $G_v = (V_{G_v}, A_{G_v})$,

que guarda uma ordenação parcial das várias versões de uma *árvore- B^+* . V_{G_v} é o conjunto de identificadores de versões para uma árvore e $(v_l, v_{l+1}) \in A_{G_v}$, quando a versão v_{l+1} é obtida pela atualização de v_l . A raiz do grafo corresponde à primeira versão.

*Raiz**: É uma estrutura adicional similar a um bloco de versão. Esta estrutura foi criada em virtude de cada versão poder ser enraizada por um nó diferente. Para cada entrada (v_l, p_l) de *raiz**, tem-se que p_l aponta para a raiz da versão v_l na árvore.

Anc(v_l): É o conjunto dos ancestrais da versão v_l , isto é, o conjunto de todos os identificadores de versões, de todos os caminhos no grafo de versão, da raiz do grafo até o vértice que contém o identificador de v_l .

Figura 3.5: (a) Estrutura dos nós gordos; (b) representação de uma *árvore- B^+* com persistência total.

A operação de busca de um registro com valor i em uma versão v_l , começa na raiz*. Em cada nó gordo visitado escolhe-se o par (v_r, p_r) , onde v_r corresponde ao valor máximo em $\text{Anc}(v_l)$. Se o bloco de índice apontado por p_r não for folha, escolhe-se o ponteiro p_j tal que $i_j \leq i$, ($0 \leq j \leq m$) e vai-se para o nó gordo apontado por p_j . Se o bloco apontado por p_r for folha, escolhe-se o valor igual a i , caso exista, recuperando o nó que contém o registro de dados com valor i . O número de acessos em uma operação de busca é, no pior caso, $(2 + 2 \log_{K+1} \frac{I+1}{2})$, onde I é o número total de registros em uma versão e K a ordem da árvore. O termo (2) equivale aos dois acessos realizados para o nó raiz* e para recuperar o registro de dados; já o termo $(2 \log_{K+1} \frac{I+1}{2})$ corresponde aos nós visitados (blocos de índices e blocos de versões).

Considere a operação de inserção de um registro com valor i , em uma versão v_l , originando a versão v_{l+1} . Inicialmente executa-se uma busca para determinar o bloco de índice B correspondente a versão v_l em um nó gordo folha n_g . Em seguida, realiza-se uma cópia de B em um novo bloco B_1 , inserindo-se i em B_1 . B_1 corresponde à versão v_{l+1} em n_g . Se B_1 tornar-se um nó sobrecarregado, o mesmo é desdobrado em um novo bloco de índice B_2 , o que proporciona a criação de um novo nó gordo $n_{g'}$, contendo B_2 na versão v_{l+1} . Em seguida, insere-se uma entrada no nó gordo pai de n_g , para refletir que n_g e $n_{g'}$ são irmãos. Essa inserção pode causar uma sobrecarga no pai de n_g e uma consequente reaplicação do procedimento de desdobramento. Quando há uma propagação até a raiz, cria-se uma nova raiz para a versão v_{l+1} , proporcionando uma nova entrada em raiz*.

O procedimento de remoção de um registro com valor i em uma versão v_l , gerando a versão v_{l+1} , é análogo à inserção. Entretanto, se o bloco de índice B_1 correspondendo à versão v_{l+1} no nó gordo n_g tornar-se subcarregado ele deve ser concatenado ou distribuído com um de seus irmãos. Seja B_2 esse irmão, pertencente ao nó gordo $n_{g'}$. Se B_2 tem no máximo $K+1$ valores, colocam-se os elementos de B_2 em B_1 e indica-se ao nó gordo, pai de $n_{g'}$, que $n_{g'}$ não é mais seu filho na versão v_{l+1} . Isto pode acarretar uma propagação deste processo na árvore. Por outro lado, se B_2 tem mais do que $K+1$ valores, os elementos de B_1 e B_2 são distribuídos entre B_1 e um novo bloco B_3 que corresponde a v_{l+1} em $n_{g'}$.

3.4.2 Método do Campo Gordo

Com o intuito de aumentar a utilização de espaço, Lanka e Mayas [52] propuseram um método, que é uma extensão do método do nó gordo, denominado de *método do campo gordo* (“fat-field”). Nesse método não é necessário utilizar um novo nó para armazenar os registros de uma nova versão. Foram feitas algumas mudanças na estrutura empregada no método do nó gordo que serão descritas a seguir.

Seja $n : p_0, i_1, p_1, \dots, i_m, p_m$ um nó interno de uma *árvore- B^+* tradicional, que está

sendo estendida para comportar as várias versões. Tem-se que:

Bloco de Índice : É um conjunto de K a $2K$ quádruplas ordenadas (*entrada*, *saída*, i , p) correspondente a um nó da *árvore- B^+* tradicional. No bloco de índice correspondente a n , cada quádrupla é a extensão de um par (i_j, p_j) , $1 \leq j \leq m$ de n . A quádrupla relativa a p_0 tem valor vazio. Assim,

entrada (saída): É um *identificador de versão* que indica a versão em que o registro foi inserido (removido);

i = Valor i_j de n , $1 \leq j \leq m$;

p = Ponteiro p_j de n , $1 \leq j \leq m$, referenciando um nó gordo ou um registro de dados;

Registros Acessíveis: Para uma versão v_l , são os registros dentro de um bloco de índice formados pelas quádruplas em que o conteúdo do campo *entrada* pertence a $\text{Anc}(v)$, e o campo *saída* não contém nenhum elemento de $\text{Anc}(v)$.

Bloco Balanceado: É aquele em que o número de registros acessíveis, para toda versão no bloco, é pelo menos K .

Nó Gordo Balanceado: Contém todos os seus blocos de índices balanceados.

Árvore- B^+ com Versões Balanceada: É a *árvore- B^+* em que todos os seus nós gordos são balanceados.

A figura 3.6 ilustra um trecho de uma árvore com o método do campo gordo. O símbolo (*) indica que o campo *saída* está vazio.

A operação de busca de um registro com valor i , em uma versão v_l , é similar a do método do nó gordo. Em um bloco de índice não folha visitado, escolhe-se uma quádrupla em que: o conteúdo do campo *entrada* é maximal em $\text{Anc}(v)$, o conteúdo do campo *saída* não pertence a $\text{Anc}(v)$, o valor i_j é tal que $i_j \leq i$ e não existe i' tal que $i' < i_j \leq i$. Em um bloco de índice folha visitado, escolhe-se uma quádrupla com as características acima, exceto que $i_j = i$.

A inserção de um registro com valor i , em um bloco de índice B , em uma versão v_l , originando a versão v_{l+1} é realizada de maneira similar que no método do nó gordo. Quando B for um bloco completo⁵, executa-se uma operação de cópia. Como o método do campo gordo permite que o valor i seja armazenado em um bloco que não contém registros acessíveis na versão v_l , então realiza-se a operação de cópia para evitar que um desdobramento do bloco venha a gerar uma árvore desbalanceada. O procedimento de

⁵Aqui, a definição de bloco de índice completo é análoga a de nó completo, vista na seção 1.2.

cópia consiste em eliminar os registros não acessíveis da versão v_l no bloco B . Isto é, todos os registros de B acessíveis a v_l , são copiados em um novo bloco B_1 e o valor i é inserido em B_1 . Após realizada a operação de cópia, se B_1 estiver sobrecarregado ele é desdobrado como no método do nó gordo.

O procedimento de remoção em um bloco B marca o registro como removido, inserindo a versão atual (v_{l+1}), no campo *saída*. Se B for um bloco completo, ele é inicialmente copiado em B_1 . Se B_1 contiver menos do que K registros acessíveis na versão v_{l+1} , ele é concatenado ou redistribuído com um bloco irmão adjacente. Seja B_2 este irmão. Na situação em que B_2 tem mais do que K registros acessíveis na versão v_{l+1} , redistribui-se; quando B_2 tem exatamente K registros acessíveis, concatena-se. Se após uma dessas operações algum bloco se tornou completo, executa-se o procedimento de cópia com este bloco.

Figura 3.6: Trecho de uma representação do método do campo gordo.

3.4.3 Método do Bloco de Versão Pura

Neste método, (“pure-version”), Lanka e Mayas [52] propuseram uma estrutura onde os nós folhas de uma *árvore-B⁺* não apontam para registros de dados, mas para um nível adicional da árvore contendo blocos de versão pura.

Um *bloco de versão pura* é uma coleção de registros de tamanho variável, ordenados pelo valor do índice, com formato $(i, < v, p >)$, onde:

i = valor do índice;

v = identificador de versão;

p = ponteiro para o registro de dados com valor i na versão v ;

Um registro de dados com valor i pode ser modificado em uma ou mais versões. Para cada versão existe um par (identificador, ponteiro), como mostrado na figura 3.7.

A operação de busca de um valor i em uma versão v_l é realizada como na *árvore- B^+* . Ao atingir o bloco de índice folha, escolhe-se o ponteiro para o bloco de versão que contém i . Este bloco de versão é percorrido para determinar a entrada cujo identificador de versão tem valor maximal em $\text{Anc}(v_l)$ e o ponteiro diferente de *nulo*. O procedimento faz acesso a no máximo $(2 + \log_{K+1} \frac{I+1}{2})$ nós da árvore, onde I é o número total de registros de dados referente a todas as versões. O termo (2) é relativo ao acesso ao bloco de versão pura e a recuperação do registro de dados.

Figura 3.7: Representação de uma *árvore- B^+* no método de versão pura.

A inserção de um registro com valor de índice i , na versão v_l , gerando a versão v_{l+1} , é realizada considerando-se 2 casos distintos.

Caso 1: o valor i ainda não existe na árvore.

Nesta situação o registro $(i, \langle v_{l+1}, p \rangle)$ é inserido no mesmo bloco de versão pura que contém o valor do índice antecessor ou sucessor a i . A inclusão pode acarretar um desdobramento no bloco de versão pura.

Caso 2: o valor i já existe na árvore.

A árvore não é modificada, mas o registro que contém o valor i em um bloco de versão pura precisa ser atualizado, isto é, insere-se o par $\langle v_{l+1}, p \rangle$. Se a inserção gerar uma sobrecarga no bloco de versão, este é desdobrado.

A remoção de um registro em uma versão v_l originando a versão v_{l+1} é realizada fazendo-se uma busca para determinar o bloco de versão apropriado. Em seguida, insere-se o par $\langle v_{l+1}, \text{nulo} \rangle$. O bloco terá a forma $(i, \langle v_l, p \rangle, \langle v_{l+1}, \text{nulo} \rangle)$. No caso de uma sobrecarga do bloco, o mesmo é desdobrado.

Lanka e Mays consideram o método de versão pura mais eficiente em espaço do que o método do campo gordo, uma vez que os registros de índices nunca são duplicados.

3.5 Árvores- B com Vários Discos

Uma maneira de se paralelizar as estruturas do tipo *árvore- B* , consiste em se distribuir seus dados ao longo de vários discos, através de 3 formas distintas, apresentadas a seguir, segundo Seeger e Larson [86].

1. Distribuição de Registros

A idéia desta técnica é utilizar para um único arquivo de dados várias *árvores- B* , colocando cada uma em um disco diferente. Quando um registro de dados é designado a um disco, ele se mantém no mesmo até ser removido. Seeger e Larson, mencionam que existem 3 critérios distintos para realizar a divisão dos registros entre os discos.

(a) Particionamento em Função do Intervalo dos Valores do Índice

Um registro é inserido no disco j quando possui o valor do índice pertencente ao conjunto dos valores deste disco, por exemplo I_j . Nas operações de inserção, remoção e consulta de um valor i , determina-se primeiro o disco que contém i , e em seguida prossegue-se normalmente fazendo as operações como nas *árvores- B* . A busca por intervalo pode envolver o acesso a vários discos, o que é realizado em paralelo.

(b) Particionamento por Hashing

Outra maneira de se separar os registros nos discos é através do uso de uma função de hashing aplicada aos valores do índice, cujo resultado determina o número do disco. A utilização de uma função apropriada pode garantir uma distribuição balanceada dos registros nos vários discos. Note-se que a busca por intervalo é feita examinando-se simultaneamente todos os discos.

(c) Particionamento Cíclico

Os registros podem ser distribuídos de forma cíclica por ordem de chegada, isto é, o j -ésimo registro deve ser inserido no disco $(j \bmod l + 1)$, onde l é a quantidade total de discos. Este esquema possibilita um balanceamento aproximado, mas

tem como desvantagem a necessidade de se visitar todos os discos durante a execução de uma determinada operação, uma vez que não é possível saber em qual disco o registro foi alocado.

2. Árvores-B com Super-páginas

Neste método um nó da árvore é implementado com várias páginas que formam uma “super-página”, contendo l páginas, as quais são alocadas em l discos diferentes. A recuperação da totalidade de uma super-página requer l leituras diferentes. O ideal seria determinar a página dentro da super-página, que contém o registro sendo procurado, trazendo para a memória primária unicamente aquela página.

Conforme já demonstrado, a organização Bounded Disorder (v. 2.11) e a *árvore-BD* (v. 2.10), caracterizam-se por conter super-páginas e realizar o acesso a uma única página l a cada recuperação de um registro. Isto é possível através do uso de técnicas de hashing. Outro exemplo de utilização desse método foi dado por Pramanik e Kim⁶, nas *árvores-PNB*, a qual difere da *árvore-B* pelo fato de não conter as entradas de uma super-página obrigatoriamente em ordem; no entanto, dentro de uma página todas as informações estão ordenadas. Segundo Seeger e Larson, a principal vantagem desta estrutura é o aumento no desempenho das consultas por intervalos com faixas amplas.

3. Distribuição de Páginas

Estes casos caracterizam-se por conter as páginas de uma *árvore-B* distribuídas em l discos, requisitando que cada entrada do índice seja expandida para conter a indicação do disco que possui o nó apontado pela mesma. A seguir são mostradas as 2 variações para executar a distribuição das páginas.

(a) Distribuição Aleatória

Quando uma página é criada ou desdobrada ela é alocada aleatoriamente em um disco. Isso pode ser feito através do uso de um gerador de números aleatórios ou por uma função de hashing. Seeger e Larson garantem que esta técnica permite um bom balanceamento das informações do ponto de vista global, porém não há garantia do balanceamento local; isto é, as páginas de uma sequência contígua (valores vizinhos do índice) de páginas de dados, não estão distribuídas em discos diferentes.

(b) Distribuição Cíclica

As páginas são inseridas no disco de maneira sistemática mantendo um número balanceado de páginas em cada disco. Não há garantia de balanceamento local.

Esses métodos de distribuição de páginas não possuem um bom balanceamento local, resultando em um alto tempo de resposta para as buscas de intervalo. Note-se que no

⁶Como não foi possível localizar este artigo, limitamos-nos aos comentários de Seeger e Larson.

pior caso, todos os elementos do intervalo podem estar no mesmo disco, impedindo uma paralelização da consulta. Seeger e Larson desenvolveram uma técnica para melhorar essa distribuição através do algoritmo dado a seguir.

O algoritmo tem como parâmetro de entrada a *página* criada a ser alocada em um disco. Conj-Disco é o conjunto dos discos que podem recebê-la; as funções Página-Dir e Página-Esq determinam as páginas à direita e à esquerda, respectivamente, da página corrente; a função Disco retorna o disco que contém uma determinada página; o vetor Quant contém o número de páginas em cada disco, devendo existir um para cada nível da árvore. Na situação em que existem mais de 2 discos possíveis para receber a nova página, deve-se escolher aquele que contém menos páginas; no caso de ocorrência de uma segunda coincidência opta-se por aquele de menor número.

Algoritmo Localiza-Disco (página):

Conj-Disco := $\{1, 2, \dots, l\}$;

Esq := Dir := *página*;

Para $j := 1$ até $\lceil \frac{l}{2} - 1 \rceil$ faça

Dir = Página-Dir(Dir);

Esq = Página-Esq(Esq);

Conj-Disco = Conj-Disco - {Disco(Esq), Disco(Dir)};

Fim-para;

Retorne($\text{Min}j$, tal que $\text{Quant}[j] = \min_{y \in \text{Conj-Disco}} \text{Quant}[y]$);

A figura 3.8 ilustra um exemplo do algoritmo quando ocorre a inserção do valor 24, gerando a criação da página *J*, resultante do desdobramento da página *D*, onde: (a) *árvore-B* com 3 discos antes da inserção de 24 e (b) *árvore-B* após a inclusão com o desdobramento de *D*.

Os autores garantem que quando as páginas são inseridas nos discos através do algoritmo, sem existência de remoções, então não existem 2 páginas de uma subsequência de páginas contíguas de $\lceil \frac{l}{2} \rceil$ páginas, que são armazenadas no mesmo disco. Isto pode ser claramente verificado pelo algoritmo, onde nenhuma página das $\lceil \frac{l}{2} - 1 \rceil$ à esquerda e das $\lceil \frac{l}{2} - 1 \rceil$ à direita da nova página são colocadas no mesmo disco. Na ocorrência de remoções esta condição não é garantida, uma vez que as remoções podem acarretar concatenações de páginas. Assim, quando a condição for violada, isto é, existir páginas vizinhas X_j e

$Y_j, 1 \leq j \leq \lceil \frac{l}{2} \rceil$ alocadas no mesmo disco j , então aplica-se o algoritmo Localiza-Disco para a página Y_j . Os autores afirmam que nas consultas com grandes faixas não se faz acesso a nenhum disco mais do que 2 vezes.

Figura 3.8: Exemplo da execução do algoritmo Localiza-Disco.

3.6 Segurança

O termo *segurança* é utilizado no contexto de bancos de dados para significar a proteção do mesmo contra a exibição, alteração ou destruição desautorizada das informações. Isso é um grande problema em ambientes de multi-usuários.

Quando as *árvores-B* usam memórias virtuais, o isolamento do espaço de dados do usuário é facilmente obtido pelo mecanismo de proteção de memória associado às páginas destas. Quando se faz necessário proteger o arquivo externo, deve-se usar algumas técnicas criptográficas.

Em 1976, Bayer e Metzger [10] desenvolveram um esquema de segurança para os arquivos mantidos na memória secundária em forma de páginas (nós), como as *árvores-B*

e suas variações. Esses arquivos residem na memória secundária de forma *criptográfica* e apenas pequenas porções do mesmo ficam na memória principal em forma *normal*, isto é, não codificadas.

Para melhor compreensão serão introduzidos alguns termos da teoria de criptografia baseando-se nas definições de Date [22]. Os dados originais são denominados de *texto simples* (“plain text”). Este texto simples é *criptografado* quando submetido a um *algoritmo de criptografia*, (*função criptográfica*, ou *cifrador*), que tem como entradas o texto simples e uma *chave criptográfica*; a saída do algoritmo, isto é, a forma criptografada do texto simples é chamada de *texto cifrado* (“cipher text”). Os detalhes do algoritmo podem ser públicos, entretanto a chave é mantida em segredo. Dessa forma, pelo método de Bayer e Metzger, o texto cifrado, que deve ser ininteligível para qualquer processo que não contenha a chave, é aquele armazenado na memória secundária.

Seja $S = s_1, s_2, \dots, s_n$ o texto simples dividido em segmentos, T a função criptográfica (cifrador), que dá como resultado o texto cifrado, e C_0 a chave criptográfica para T . Assim, T é distribuída sobre S como na fórmula a seguir, que usa a concatenação dos vários segmentos criptográficos e cada C_i é calculada por uma função f , onde $C_i = f(C_0, i)$.

$$T(S, C_0) = T(s_1, C_1) \cdots T(s_n, C_n)$$

Baseando-se nestas informações, as funções criptográficas T podem ser classificadas em:

1. Função de bloco (*cifrador-b*) quando o custo para se calcular qualquer chave C_i independe de i .
2. Função progressiva (*cifrador-p*) quando o custo para se calcular qualquer chave C_i aumenta com i , isto é, as chaves devem ser calculadas por uma relação de recorrência da forma: $C_i = g(C_{i-1})$.

Qualquer esquema de criptografia para os arquivos que são mantidos na memória secundária em forma de páginas deve obedecer as seguintes propriedades:

1. O método de codificação utilizado não deve causar alterações nos algoritmos de busca, inserção e remoção.
2. O método deve ser aplicável a qualquer tipo de função criptográfica utilizada, *cifrador-p* ou *cifrador-b*.
3. O método deve ser adequado mesmo quando a transformação de codificação for executada por hardware.

O Método

Bayer e Metzger sugeriram um esquema denominado de *chave da página* (“page-key”), para criptografar estruturas do tipo *árvore-B*. Para cada página do arquivo existe uma chave criptográfica, uma vez que a existência de uma única chave para todo o arquivo facilitaria o acesso ao mesmo, tornando pública as informações do arquivo.

Para uma melhor compreensão do método, considere um arquivo A com m páginas (nós), onde n_i , $1 \leq i \leq m$, é o número de cada página, utilizado para localizá-la na memória secundária. Seja $\mathcal{S}(n_i)$ o texto simples de uma página n_i e $\mathcal{C}(n_i)$ o texto criptografado de n_i . Cada página n_i tem ainda uma identificação chamada de n'_i .

O processo utiliza 2 cifradores. Um deles é o *cifrador de texto* U , que deve ser inversível, isto é, permitir tanto a codificação como a decodificação do texto, podendo ser um *cifrador-b* ou um *cifrador-p*. O outro é um *cifrador de chave de página* E , o qual não precisa ser inversível. Utiliza-se também uma *chave de arquivo* C_E para E . Esta última é também denominada de *chave de árvore* quando se trata de *árvores-B*.

Dado n'_i , para qualquer página n_i , $1 \leq i \leq m$, E é utilizado para calcular a chave de página C_{n_i} correspondente, isto é,

$$C_{n_i} = E(n'_i, C_E)$$

Já o conteúdo da página n_i é criptografado através da transformação

$$\mathcal{C}(n_i) = U(\mathcal{S}(n_i), C_{n_i})$$

onde o texto simples $\mathcal{S}(n_i)$ de n_i pode ser dividido em vários segmentos. O conteúdo de n_i é decifrado por

$$\mathcal{S}(n_i) = U^{-1}(\mathcal{C}(n_i), C_{n_i})$$

Existem 2 maneiras para se calcular os identificadores das páginas, como explicado a seguir.

1. Método C - Método do Cifrador Puro

As páginas são identificadas diretamente pelos números das páginas ($n'_i = n_i$). Assim, todas as chaves das páginas são calculadas quando a chave do arquivo C_E for escolhida da seguinte forma:

$$C_{n_i} = E(n_i, C_E)$$

2. Método T - Método da Tabela

Neste método as páginas são identificadas por valores arbitrários, os quais são armazenados em uma tabela organizada pelos números das páginas. Assim, as chaves

de cada página podem ser alteradas individualmente através da modificação da tabela, entretanto tem como desvantagem a necessidade de se buscar na tabela a identificação de uma página, sempre que se precise da chave da mesma.

Esses 2 métodos permitem que as chaves das páginas não estejam necessariamente explicitamente armazenadas mas, sempre que preciso, possibilitam uma derivação através da chave do arquivo C_E . Por conseguinte, o método C é preferível ao T, pois não requer memória adicional para armazenar a tabela nem procedimentos extras para recuperar as informações da mesma.

Uma questão a ser vista é a de como um cifrador de texto U deve ser aplicado às páginas individuais da *árvore-B*. Para isso deve-se considerar como a criptografia afeta os algoritmos de acesso e atualização e quais os custos acarretados. A solução para criptografar as informações contidas em uma sequência ordenada de elementos indexados é dividir o texto em segmentos de tamanho apropriado e codificá-los com um *cifrador-p* ou *cifrador-b*, dependendo do custo de utilização de cada um destes.

Segundo Bayer e Metzger, a experiência tem mostrado que quando a codificação e decodificação podem ser executadas por hardware, então é apenas necessário que o sistema de paginação forneça a chave para cada página que tenha sofrido modificação, e dessa forma, quando retornada à memória secundária, a mesma estará corretamente codificada. Por outro lado, quando a codificação é feita por software, podem surgir algumas complicações. O sistema de paginação tem que saber como recodificar as páginas que devem ser reescritas na memória secundária, além dos detalhes de quais as partes das páginas que foram decifradas. Isso deve ser feito através de um *descriptor de decodificação* para cada página mantida na memória principal. Como mostrado na figura 3.9, este descriptor contém a chave C_{n_i} da página e um bit de modificação b . Para o caso de um *cifrador-b* haverá uma cadeia de bits especificando os segmentos decodificados (*decod*). Já para o *cifrador-p* o descriptor conterá um item l especificando o segmento criptografado e um valor de chave $C_{n_i}^l$, cujo conteúdo corresponde a chave do segmento codificado a ser usado.

Figura 3.9: Descriptor de decodificação: (a) para *cifrador-b*; (b) para *cifrador-p*.

O uso de diferentes chaves criptográficas para diferentes páginas aumenta a segurança criptográfica, pois a quebra da codificação de uma página não ajuda na quebra de outras

páginas, uma vez que as chaves das mesmas não possuem relacionamentos diretos. Por outro lado, existem alguns problemas que devem ser tratados, quando o segredo de alguma página for descoberto.

Problemas

1. Se a página decodificada não for uma folha, a quebra da informação de algum valor e ponteiro da mesma permitiria o acesso a algum de seus filhos, sendo ainda mais perigoso quando os filhos forem folhas. Uma forma de contornar essa situação, tornando a localização dos filhos das páginas mais difícil, é armazenar os ponteiros codificados pelo cifrador E em páginas, dentro dos textos simples.
2. Quando uma determinada página é invadida, através do conhecimento de sua chave, todas as novas informações que venham a surgir na mesma, ocasionadas pelos desdobramentos, concatenações ou redistribuições, serão facilmente conhecidas. A solução para este problema consiste em modificar as chaves das páginas a cada mudança na mesma, ou pelo menos a toda “grande mudança”. Isso pode ser feito facilmente quando os identificadores são mantidos em tabelas (método T); contudo, no caso do método do cifrador puro (C), essa troca não pode ser executada sem que a chave da árvore (chave do arquivo) seja modificada. A alternativa empregada é enganar o intruso, isto é, move-se o conteúdo do nó para uma nova página, atribuindo-lhe uma nova chave. Deixa-se o nó antigo intocável para que o intruso não perceba a nova mudança, até o instante em que a chave do arquivo possa ser trocada. Esta estratégia pode levar a um número muito grande, até mesmo inaceitável, de páginas não utilizadas.
3. Ao se decodificar uma página n_i , $1 \leq i \leq m$, o intruso é capaz de deduzir a chave da página C_{n_i} e o identificador n'_i . Dessa forma, como $C_{n_i} = E(n'_i, C_E)$ o intruso deve tentar inverter E para obter C_E . Se o conjunto dos possíveis valores para as chaves da árvore não for grande, a mesma poderá ser encontrada através de tentativas. Uma solução é utilizar um conjunto muito grande.

Capítulo 4

CONCORRÊNCIA

4.1 Introdução

As aplicações de multiprocessamento e multiprogramação também estão presentes em sistemas de banco de dados, sendo de grande interesse para essa área a construção de banco de dados que sejam manipulados de maneira concorrente e correta, por vários processadores e tarefas.

A utilização da estrutura de *árvore-B*, juntamente com suas variações, em ambientes de multiusuários, é bastante frequente. Os problemas existentes em qualquer sistema de concorrência continuam presentes, como por exemplo: garantir a corretude e integridade das informações em função da interferência de um processo com outro; escalonamento de processos (“scheduling”), de tal forma que 2 processos não sejam realizados concorrentemente quando houver conflitos entre suas tarefas; execução interminável de um determinado processo (“livelock”) ou a existência de “deadlock”, isto é, a situação em que 2 ou mais processos estão em estado de espera simultâneo, devido a um bloqueio cíclico, ou seja, cada processo está esperando por algum evento que não ocorrerá.

Objetivando evitar estes tipos de problemas utiliza-se um mecanismo para regular o acesso concorrente a objetos compartilhados. O controle de concorrência empregado nas soluções de sistemas de *árvore-B* que serão apresentadas adiante é formado por 3 componentes: *bloqueios*, *grafo de compatibilidade e conversão (GCC)* e *escalonamento*. O *bloqueio* em um nó consiste de uma espécie de “marcação” no mesmo, a qual reserva o acesso deste nó a algum processo, afim de assegurar a integridade do procedimento; esta é garantida através do tipo de bloqueio utilizado. Em algum outro instante da execução do processo, o *bloqueio* é liberado, configurando o ato de *desbloquear*, isto é, o nó ficará

livre para ser utilizado por outros processos. Certas soluções permitem que se façam conversões de um tipo de bloqueio aplicado a um nó, para outro tipo. Generalizando os tipos de bloqueios existentes tem-se o *bloqueio exclusivo* e o *particionado*. O primeiro garante que se um determinado processo P contém um bloqueio *exclusivo* em um nó n , então nenhum outro processo P' pode bloquear n até que P o libere. No caso de n receber um bloqueio *particionado* de um processo P , então qualquer outro processo P' pode realizar um bloqueio *particionado* sobre n , mas não é permitida a execução de *bloqueios exclusivos* até que os *bloqueios particionados* sejam liberados.

A relação de compatibilidade e conversão existente entre os diversos tipos de bloqueios pode ser representada pelo *GCC* introduzido por Bayer e Schkolnick [11]. Sejam α e β dois tipos de bloqueios possíveis de serem aplicados a um nó n . Se α e β têm compatibilidade completa, isto é, α é aplicado a n por um processo P_1 , enquanto que um outro processo P_2 pode bloquear n simultaneamente com β , ou vice-versa, tem-se o grafo mostrado na figura 4.1 (a). Porém, se α e β têm compatibilidade assimétrica, isto é, um processo P_1 com bloqueio β em um nó n permite outro processo P_2 colocar um bloqueio α em n , porém o contrário não pode, tem-se o grafo da figura 4.1 (b). Já a conversão entre os tipos de bloqueios é indicada como na figura 4.1 (c), onde um nó n com um bloqueio do tipo α , aplicado por um processo P_1 , pode sofrer uma conversão de seu tipo, pelo próprio processo P_1 , para β por exemplo; ou como na figura 4.1 (d) onde a conversão é completa (de α para β ou de β para α).

Figura 4.1: *GCC* - Grafo de Compatibilidade e Conversão.

O *escalonamento* consiste de uma fila, associada a cada nó n , de processos e de seus pedidos de bloqueios. Como exemplo de uma disciplina bastante utilizada tem-se aquela onde os pedidos são executados por ordem de chegada, ou seja, o primeiro pedido a entrar na fila será o primeiro a sair (PEPS)¹. Geralmente, quando um processo deseja executar um bloqueio em um nó n , coloca-se um pedido de bloqueio na fila do mesmo. O processo fica em estado de espera até que o seu pedido atinja o início da fila.

Quanto aos tipos de processos existentes estes podem ser classificados em: *leitor*, o

¹Esta disciplina é também conhecida na literatura como FIFO (First In First Out).

qual executa a operação de busca de um determinado valor i do índice e o *atualizador*, responsável pelas operações de inserção e remoção de um valor. O processo *atualizador* é composto de duas fases: a primeira fase (*fase 1*) consiste na busca do nó onde (de onde) será inserido (removido) um valor; a segunda (*fase 2*) diz respeito à inserção ou remoção propriamente dita, e da conseqüente reestruturação da árvore, quando houver necessidade de desdobrar um nó (inserção), concatenar ou redistribuir os elementos de alguns nós (remoção) e estes efeitos se propagarem na árvore. Denomina-se de *leitor-puro* o processo que executa simplesmente a operação de busca de um valor do índice; de *leitor-atualizador* aquele responsável pela fase 1 do processo de atualização e de *atualizador-gravador* ou simplesmente *gravador* o executor da fase 2. O conjunto dos nós do caminho percorrido por um processo de atualização que são alterados durante uma reestruturação decorrente desse processo é denominado de *escopo*. Os nós da árvore visitados por um determinado processo P formam o *caminho de acesso* de P .

Um nó n é chamado de *seguro* (“safe”) quando durante uma inserção ou remoção nenhum dos seus antecedentes serão afetados; caso contrário, será chamado de *não-seguro* (“unsafe”). Assim, para uma inserção em uma *árvore-B* onde K representa a ordem da árvore, um nó é considerado *seguro* se a quantidade de valores do mesmo for menor do que $2K$; já para uma remoção, um nó está *seguro* quando seu número de valores é maior do que K .

Uma solução para o problema de concorrência consiste em permitir que processos leitores manipulem a estrutura simultaneamente com outros processos leitores, mas para o caso dos processos atualizadores estes são *serializados*. De acordo com Date [22]², os processos são *serializados* quando estes são sincronizados, através de algum mecanismo de concorrência, tal que a execução dos mesmos produz resultados iguais a alguma execução sequencial destes em qualquer ordem. Com o intuito de conseguir esta *serialização*, um processo atualizador, antes de sua execução, ganha o controle exclusivo de toda a árvore, evitando que esta seja manipulada por qualquer outro processo. Todavia, como esta alternativa só deve ser utilizada quando o nível de atividade for baixo, surgiram outras soluções.

As primeiras soluções para o problema de concorrência em *árvore-B* foram dadas por Samadi [85] em 1976 e Parr [78]³ em 1977, fazendo uso da técnica padrão de semáforos (Dijkstra [24]), para controlar a concorrência.

Bayer e Schkolnick [11] propuseram em 1977 uma melhora no método de Samadi, indicando um esquema parametrizado, cuja variação ocorre de acordo com o grau e o tipo de concorrência desejada. Este modelo possibilita um pequeno uso de bloqueios exclusivos,

²Segundo Date a noção de *serializável* foi primeiro desenvolvida, usando outra terminologia, em um trabalho de K.P.Eswaran e colaboradores.

³Como não foi possível localizar esse artigo, limitamo-nos às citações de [48, 50].

proporcionando um aumento no número de processos executados concorrentemente.

Por outro lado, Held e Stonebraker [34] realizaram em 1978 estudos sobre a possibilidade do uso de *árvores-B* em concorrência, mostrando os problemas que podem surgir. Por exemplo, supondo que dois processos, um atualizador e outro leitor-puro estão fazendo acessos simultâneos na árvore e que o processo leitor-puro está verificando um determinado nó n da árvore no momento em que ocorre uma divisão de algum outro nó, causada pelo processo de atualização, poderá acontecer que o processo de leitura detecte um elemento errado ou não existente. Outra dificuldade, também verificada pelos referidos autores, ocorre quando dois processos estão inserindo informações em dois nós adjacentes que estejam ambos completos. Os dois processos bloquearão as páginas que estão sendo atualizadas e deverão examinar os nós vizinhos, a fim de redistribuir os elementos entre eles (técnica de overflow, v. 1.4.2). Entretanto, como estão bloqueados, haverá um impasse (deadlock), restando apenas a opção de se escolher aleatoriamente um dos processos em questão, para que este seja interrompido e sejam liberados todos os bloqueios realizados pelo mesmo, “quebrando-se” a situação de deadlock.

Em 1978, Miller e Snyder [67] investigaram um esquema que bloqueia uma região da árvore de tamanho delimitado, fazendo uso de 3 tipos de bloqueadores: *acesso* (“access”), *pioneiro* (“pioneer”) e *seguidor* (“follower”).

Também em 1978, Guibas e Sedgewick [32] sugeriram uma modificação do algoritmo de atualização, a qual pode ser adaptada como uma nova solução para o problema de concorrência. Esta modificação consiste em se fazer desdobramentos prévios nos nós da *árvore-B* durante a fase 1 de algum processo de atualização. Isto é, à medida que se caminha na árvore, de cima para baixo, e detecta-se um nó completo, este é desdobrado e o separador apropriado inserido no respectivo nó pai. Note-se que o pai nunca se encontra completo e conseqüentemente não há propagação do processo.

Com base em uma das soluções dada por Bayer e Schkolnick e seguindo idéia de Lamport [51], Ellis [27] apresentou em 1980 uma solução para o problema de concorrência nas *árvores-2-3*, em particular para os casos de busca e inserção. Esta solução permite que processos leitores e atualizadores tenham acesso a um determinado nó ao mesmo tempo, em direções opostas: o processo leitor visita os nós da árvore no sentido de cima para baixo e examina os valores internos de um nó da esquerda para direita; o processo atualizador reestrutura a árvore de baixo para cima e faz trocas nos conteúdos dos nós da direita para esquerda. Outra ressalva do método consiste na indivisibilidade das operações de leitura e escrita de um valor e de seu respectivo ponteiro, isto é, a leitura (escrita) de um par (valor, ponteiro) é executada como um único passo. A chave principal dessa solução é permitir processos leitores-puros no escopo de processos atualizadores. Com o intuito de aumentar a concorrência entre os processos atualizadores, Ellis propôs outro algoritmo denominado de *algoritmo pipeline*. A idéia é permitir que a fase 1 dos processos atualizadores seja

executada da mesma forma que um processo leitor.

Lehman e Yao [53], definiram em 1981 um método que utiliza um número constante de nós bloqueados, o qual foi aperfeiçoado em 1986 por Sagiv [84], apresentando uma solução em que apenas 1 único nó é bloqueado.

Em 1980/1982 Kwong e Wood [48, 49, 50], utilizando a técnica de *side-branching*, a ser vista adiante, apresentaram uma melhora das soluções de Bayer e Schkolnick e de Ellis.

Com base na modificação do algoritmo de atualização proposto por Guibas e Sedgwick, em 1985 Mond e Raz [70] sugeriram uma nova solução para o problema de concorrência, onde cada processo executa apenas 2 bloqueios simultâneos nos nós visitados.

Em 1988 Keller e Wiederhold [42] fizeram um estudo sobre o uso de concorrência em *árvores-B* contendo entradas de tamanho variável.

Finalmente, Souza e Carvalho [89] propuseram em 1992 um sistema denominado de LEMIX, que possui um gerenciador de banco de dados baseado em *árvores-B**, introduzindo um método de controle de concorrência em que um caminho hierárquico fica residente na memória e é percorrido várias vezes até ser bloqueado.

Todas as soluções existentes para o problema de concorrência podem ser classificadas, de acordo com Kwong e Wood em *tipo 1* e *tipo 2*. Nas soluções de *tipo 1* o caminho da árvore percorrido por um processo atualizador não varia durante a fase de reestruturação, fazendo o uso de bloqueio exclusivo em todos os nós deste caminho, afim de que nenhum outro processo atualizador possa compartilhar este percurso. Todavia, existem soluções que permitem o compartilhamento do caminho por processos que executam simplesmente a busca de um elemento (leitores-puros). Observa-se que nas soluções de *tipo 1* o número de nós bloqueados por cada processo atualizador é da ordem $O(h)$ da altura da árvore. Nas soluções de *tipo 2* o caminho de acesso percorrido por um processo atualizador é modificado durante a fase de reestruturação. Assim, apenas os nós que sofrem modificações em cada passo desta fase de atualização são bloqueados, não permitindo, temporariamente, o acesso a eles por outros processos atualizadores; estes nós formam o *caminho local*. Com isso, essas soluções permitem vários processos atualizadores no percurso de outros. O número de nós bloqueados por cada processo atualizador em qualquer instante do procedimento é $O(C)$, onde C é uma constante.

Sagiv [84] propôs uma outra classificação para as soluções existentes, agrupando-as em *ascendentes* (“bottom-up”) e *descendentes* (“top-down”). Nas soluções *descendentes* um processo atualizador bloqueia todo seu percurso de tal forma a evitar outros atualizadores neste caminho; entretanto, algumas variações destas soluções permitem processos leitores tendo acesso simultâneo a este percurso na fase de busca e/ou na de reestruturação.

Já nas *ascendentes*, os processos atualizadores comportam-se como leitores na fase de busca e durante a reestruturação eles se movimentam na árvore, no sentido para cima, bloqueando simultaneamente apenas alguns nós na árvore (no máximo 3), ao invés de todo o percurso, e fazendo as transformações necessárias. Essas soluções garantem um alto grau de concorrência.

Vejamos as várias soluções para o problema de concorrência em ordem cronológica.

4.2 Solução devida a Samadi

Consiste de uma solução do *tipo 1* sendo considerada como a primeira solução válida para o problema de concorrência em sistemas que utilizam *árvore-B*. Todavia, Kwong e Wood [48, 50] afirmam que em 1977 Parr [78] propôs independentemente uma solução similar.

4.2.1 Definições

A solução a ser apresentada utiliza um único tipo de bloqueio, chamado de *bloqueio semáforo*, tanto para os processos leitores-puros como para os atualizadores, cujo *GCC* dado por Kwong e Wood é mostrado na figura 4.2. Dois processos distintos não podem bloquear simultaneamente um mesmo nó. São válidas apenas as operações de bloquear e desbloquear um nó, as quais são executadas através da técnica padrão de *semáforos* (Dijkstra). Um *semáforo* é uma variável protegida cujo valor é alterado apenas pelas operações *P* e *V* (v. Tanenbaum [91]). Seja $sem[n]$ um *semáforo* associado a um nó n da *árvore-B*, composto de um campo inteiro, de um contador de processos c , e de uma fila f . $P(sem[n])$ equivale a atitude de bloqueio e $V(sem[n])$ de desbloqueio. Inicialmente $sem[n].c = 1$, para todo nó n da árvore.

Figura 4.2: *GCC* para a solução de Samadi.

$P(sem[n]):$	$V(sem[n]):$
$sem[n].c := sem[n].c - 1;$	$sem[n].c := sem[n].c + 1;$
Se $sem[n].c < 0$, então	Se $sem[n].c \geq 0$, então
Insira o processo corrente em $sem[n].f$,	Remova um processo da fila $sem[n].f$
colocando-o em estado de <i>espera</i> ;	colocando-o pronto para ser <i>executado</i> ;

4.2.2 Descrição de Funcionamento

Busca

O procedimento de busca de um valor i do índice executado por um processo leitor-puro consiste em fazer inicialmente um bloqueio na raiz e em seguida continuar caminhando na árvore, no sentido da raiz até uma folha, bloqueando e desbloqueando nós, de tal forma que ao se visitar um nó este é bloqueado e libera-se o bloqueio no seu pai.

Inserção / Remoção

Um processo atualizador desejando executar uma inserção ou remoção de um valor i na árvore- B , realiza inicialmente um bloqueio na raiz da mesma. Indutivamente, durante seu percurso na árvore, os nós visitados são bloqueados e examinados, porém, quando neste caminho for detectado um nó seguro, liberam-se todos os bloqueios de seus ancestrais.

Em virtude da maneira como os procedimentos são realizados, é necessária uma ordenação para todos os processos que compartilham um caminho comum. Nenhum processo pode ultrapassar outro, em seu caminho. No caminho bloqueado por um processo atualizador P_a , isto é, do nó seguro de maior profundidade pertencente ao caminho, até uma folha, não pode haver nenhum outro processo sendo executado, ou até mesmo esperando nas filas destes nós. De fato, suponhamos um caminho contendo o nó n , bloqueado por um processo atualizador, em que n não é o nó seguro mais profundo do caminho. Se existisse um processo leitor P_l esperando na fila de n , o pai de n teria sido bloqueado por P_l , mas P_a certamente deve ter bloqueado esse pai, o que é um absurdo.

Vantagens e Desvantagens

O problema de deadlock é evitado usando-se a própria estrutura da árvore para associar uma ordenação natural a todas as atividades de acesso ao arquivo de índices. Como todos os bloqueios são executados no sentido de cima para baixo na árvore, não existirá nenhuma possibilidade de demanda circular.

A utilização de um bloqueio consiste em uma forma de melhorar a concorrência em relação a serialização dos processos. Entretanto, conforme será mostrado adiante, utilizando-se vários tipos de bloqueios, aumenta-se ainda mais a quantidade de processos que podem ser executados simultaneamente. A presente solução deve ser usada apenas nas situações em que o nível de atividade for baixo.

4.3 Soluções devidas a Bayer e Schkolnick

O esquema a ser apresentado permite um maior grau de concorrência quando comparado com a solução de Samadi. Ele é composto por um conjunto de algoritmos dirigidos por um processo supervisor. O estudo foi feito baseando-se na estrutura de *árvore-B**, podendo também ser aplicado a outras estruturas que tenham as propriedades desta, como por exemplo *árvore-B pré-fixada* (v. capítulo 2).

As 4 soluções que serão analisadas adiante são consideradas por Kwong e Wood como soluções de *tipo 1*.

4.3.1 Definições Gerais

Para cada solução que será apresentada serão mostrados protocolos para os processos atualizadores e leitores-puros. Todas essas soluções utilizam nos nós da árvore alguns dos tipos de bloqueios abaixo:

Bloqueio leitor = ρ -lock;

Bloqueio exclusivo = ε -lock;

Bloqueio alternativo = α -lock;

O ρ -lock pode ser visto como o tipo de bloqueio mais fraco, isto é, quando um nó n apresenta bloqueio desta natureza, executado por algum processo, n pode ser utilizado por outros processos simultaneamente. Por exemplo, conforme será visto, nas soluções 2 e 3

um nó com bloqueio ρ -lock pode ser compartilhado por processos leitores e atualizadores. O ε -lock é considerado o mais forte, geralmente utilizado por processos atualizadores. Um nó n que contém este tipo de bloqueio não pode ser utilizado por nenhum outro processo, até o momento em que seja desbloqueado. O α -lock é empregado na solução 3 durante a fase 1 de algum processo atualizador, e permite que os nós com este tipo de bloqueio possam ser utilizados por outros processos ao mesmo tempo. Um nó não pode ter, no mesmo instante, um ρ -lock e um ε -lock e nem um ε -lock e um α -lock, mas pode possuir um ρ -lock e um α -lock.

O modelo permite que sejam feitas conversões de tipos entre nós que contenham α -locks para ε -locks, nesta ordem. Essa transformação não pode ser executada no caso em que existe um nó n contendo um ρ -lock e um α -lock, uma vez que ρ -locks e ε -locks são incompatíveis. A solução é esperar até que o ρ -lock seja liberado e em seguida executar a modificação. As figuras 4.3 (a) e (b) mostram o *GCC* entre os tipos de bloqueios e as conversões possíveis.

Figura 4.3: (a) *GCC* das soluções 1 e 2; (b) *GCC* da solução 3.

Para cada nó n da árvore existe uma *fila associada*, a qual armazena os pedidos de bloqueios para n , requisitados por algum processo. Os pedidos são executados por ordem de chegada, isto é, o primeiro a chegar é o primeiro a ser executado, com exceção da solução 3, na qual um pedido de conversão de bloqueio é posicionado no início da fila.

Em relação aos algoritmos que se seguem, no artigo de Bayer e Shcolnick não ocorrem os últimos passos de desbloqueios depois de executar a operação desejada. Adicionaremos estes passos para completar os algoritmos.

4.3.2 Solução 1

Conforme citado em [11], esta solução foi na realidade desenvolvida por J.K. Metzger. Consiste de uma solução similar à apresentada por Samadi, utilizando agora 2 tipos de bloqueios: ρ -lock e ε -lock.

Descrição de Funcionamento

Busca

Este procedimento é realizado através de um protocolo para processos *leitores-puros* utilizando bloqueios do tipo ρ -lock nos nós da árvore. O procedimento executa um ρ -lock na raiz tornando esta o nó corrente. Indutivamente é estabelecido um ρ -lock no nó filho apropriado do nó corrente, é liberado o bloqueio do nó corrente e seu filho passa a ser o nó corrente. O processo continua até que uma folha da árvore seja atingida. Em função do tipo de bloqueio utilizado é possível que mais de um processo leitor-puro visite simultaneamente o mesmo nó, além de poder haver ultrapassagem entre os processos leitores.

Algoritmo 1.1:

1. Faça um ρ -lock na raiz;
2. Torne a raiz o nó corrente;
3. Enquanto o nó corrente não é folha
 - Estabeleça um ρ -lock no filho imediato do nó corrente a ser utilizado;
 - Libere o ρ -lock do nó corrente;
 - Transforme o filho do nó corrente no nó corrente;Fim-Enquanto;
4. Verifique a existência do valor sendo procurado no nó corrente;
5. Libere o ρ -lock do nó corrente;

Inserção / Remoção

Estas operações são executadas por processos atualizadores, fazendo uso de bloqueios do tipo ε -lock nos nós da árvore que são visitados, pois elas podem redundar na reorganização

da árvore, em particular no caminho percorrido. O procedimento consiste em estabelecer um ε -lock na raiz da árvore, tornando-a o nó corrente. Indutivamente, até o momento em que o nó corrente não é uma folha, executa-se outro ε -lock no filho apropriado do nó corrente, fazendo deste filho o nó corrente. Se o nó corrente for seguro, isto é, a inclusão ou remoção de um novo elemento neste nó não acarreta modificações em seus antecessores, então todos os bloqueios existentes nesses antecessores são liberados, permitindo que outros processos atualizadores ou leitores possam fazer uso desses nós. Quando o nó corrente não é seguro, então todos os seus antecedentes permanecem com os bloqueios exclusivos. Como já foi visto no método de Samadi, não pode haver processos leitores no caminho bloqueado por processos atualizadores nem nas filas dos nós deste caminho.

Algoritmo 1.2:

1. Faça um ε -lock na raiz;
2. Torne a raiz o nó corrente;
3. Enquanto o nó corrente não é folha
 - Estabeleça um ε -lock no filho imediato do nó corrente a ser utilizado;
 - Transforme esse filho no nó corrente;
 - Se o nó corrente é seguro, então
 - Libere todos os *locks* dos ancestrais do nó corrente;Fim-Enquanto;
4. Se for possível realizar a atualização, então
 - Execute a atualização;
5. Libere todos os *locks* referentes a este processo;

Vantagens e Desvantagens

A solução apresentada tem a vantagem de usar algoritmos simples para conseguir uma concorrência razoável, entretanto apresenta o custo de bloquear a raiz de uma sub-árvore com um ε -lock, para atualização da raiz, incapacitando outros processos de manipular esta sub-árvore. Comparando com a proposta de Samadi, a presente solução permite um maior grau de concorrência.

4.3.3 Solução 2

Com o objetivo de aumentar a concorrência Bayer e Schkolnick propuseram outra solução que possibilita aos processos atualizadores comportarem-se como processos leitores na região de índices.

Descrição de Funcionamento

Busca

A operação de busca nesta solução é idêntica àquela apresentada na solução 1.

Inserção / Remoção

Nesta solução os processos atualizadores exploram a árvore como se os mesmos fossem processos leitores, até o momento em que um nó folha é atingido, ou seja, todos os nós do caminho de acesso de um procedimento recebem ρ -locks. Quando um nó folha n for atingido este recebe um ε -lock. Se n não for seguro, o procedimento despreza toda a análise realizada até então, desbloqueia todos os nós e inicia todo o processo novamente pela raiz utilizando o algoritmo para processos atualizadores da solução 1 (Algoritmo 1.2).

Algoritmo 2.2:

1. Faça um ε -lock na raiz;
2. Torne a raiz o nó corrente;
3. Enquanto o nó corrente não é folha
 - Se o filho do nó corrente a ser utilizado não é uma folha, então
 - Faça um ρ -lock nesse filho;
 - Se o filho do nó corrente a ser utilizado é folha, então
 - Faça um ε -lock nesse filho;
 - Libere o *lock* do nó corrente;
 - Transforme o filho no nó corrente;
- Fim-Enquanto;

4. Se o nó corrente não está seguro, então

Libere todos os *locks* e repita o acesso à árvore usando agora o protocolo de *atualização* dado na solução 1;

5. Se o nó corrente está seguro, então

Se for possível realizar a atualização, então

Efetue a atualização;

Libere o *lock* do nó corrente;

Vantagens e Desvantagens

A solução apresenta um aumento no número de processos concorrentes, em função da diminuição da quantidade de bloqueios exclusivos executados nos nós da árvore, e permite que processos leitores e atualizadores compartilhem os níveis mais altos da árvore. Entretanto, pode acontecer que nas árvores com grande profundidade o tempo gasto com análises sem sucesso seja bastante elevado, uma vez que é necessário repetir a busca na região de índice.

4.3.4 Solução 3

Esta solução surgiu como alternativa para o problema do uso da solução 2 em árvores altas. Ela utiliza bloqueios do tipo α -lock, ρ -lock e um ε -lock, permitindo que sejam feitas conversões de α -lock para ε -lock. Um pedido de conversão de tipo de bloqueio em um nó n tem prioridade sobre qualquer outro pedido, sendo sempre colocado no início da fila de n . No caso em que um processo requisita uma conversão incompatível com o tipo de bloqueio existente, isto é, o nó n encontra-se bloqueado com um ρ -lock e α -lock, a transformação não é executada e o processo é mantido em estado de espera no início da fila, até que o bloqueio ρ -lock seja liberado e a conversão possa ser concretizada, pois ρ -lock e ε -lock são incompatíveis.

Descrição de Funcionamento

Busca

Nessa solução a operação de busca também é realizada como aquela vista na solução 1.

Inserção / Remoção

Assemelhando-se com aquele usado na solução 1, o algoritmo para processos atualizadores difere no fato de que o mesmo faz uso de bloqueios do tipo α -lock ao invés de ε -lock, permitindo assim o compartilhamento dos nós bloqueados com α -locks entre os processos leitores e atualizadores. Quando um nó folha é atingido faz-se uma conversão, no sentido raiz-folhas, entre os nós que apresentam bloqueios do tipo α -lock para ε -lock, evitando repetição da análise, como ocorre na solução 2. Dessa forma, os ε -locks serão apenas atribuídos aos nós que sofrerão modificações, diminuindo substancialmente o número de nós que não podem ser usados por outros processos simultaneamente.

Algoritmo 3.2:

1. Faça um α -lock na raiz;
2. Torne a raiz o nó corrente;
3. Enquanto o nó corrente não é folha
 - Estabeleça um α -lock no filho imediato do nó corrente a ser utilizado;
 - Transforme esse filho no nó corrente;
 - Se o nó corrente está seguro, então
 - Libere todos os *locks* dos ancestrais do nó corrente;
 - Fim-Enquanto;
4. Se for possível realizar a atualização, então
 - Converta todos os α -locks em ε -locks de cima para baixo;
 - Efetue a atualização;
5. Libere todos os *locks* referentes a este processo;

Vantagens e Desvantagens

Como na solução 1, a principal desvantagem observada diz respeito ao impedimento temporário de determinados nós da árvore serem acessados por outros processos, quando os mesmos forem bloqueados com ε -locks. Outro ponto negativo é o tempo gasto com as conversões dos tipos de bloqueios. Bayer e Schkolnick afirmam que na prática, as transformações dos nós contidos nos níveis superiores da árvore são raras, pois estes geralmente são *seguros* e consequentemente desbloqueados. Assim, como alternativa para melhorar o tempo gasto com as conversões nos nós localizados nos níveis mais baixos da árvore, os

autores propuseram uma alteração no algoritmo, a qual consiste em atribuir um ε -lock a estes nós, no instante imediato em que os mesmos forem atingidos.

A solução permite a vantagem do compartilhamento de nós entre processos leitores e atualizadores, além de não haver a possibilidade de análises desnecessárias como na solução 2.

4.3.5 Solução 4 (Generalização)

Bayer e Schkolnick admitem que dentre as 3 soluções vistas nenhuma pode ser considerada como a melhor, pois dependendo da situação uma apresenta vantagens sobre as outras. Por essa razão eles propuseram uma quarta solução consistindo de uma combinação generalizada das demais.

Definições

Para a solução generalizada são necessários 4 tipos de bloqueios:

Bloqueio leitor-leitor = ρ_l -lock;
 Bloqueio leitor-atualizador = ρ_a -lock;
 Bloqueio alternativo = α -lock;
 Bloqueio exclusivo = ε -lock;

O bloqueio ρ_l -lock é utilizado apenas em processos leitores (leitores-puros) tendo a mesma função do ρ -lock para estes processos nas soluções 1, 2 e 3. Já o ρ_a -lock é estabelecido pelos processos atualizadores, tendo a mesma função do ρ -lock nestes processos, na solução 2. Os bloqueios α -lock e ε -lock têm o mesmo efeito que nas soluções anteriores. Um nó n pode conter ao mesmo tempo um bloqueio do tipo ρ_l -lock e ρ_a -lock, ou ρ_l -lock e α -lock. As conversões entre os tipos de bloqueios são tanto de α -lock para ε -lock como vice-versa, conforme mostrado no *GCC* da figura 4.4.

A solução para os processos atualizadores é parametrizada através dos parâmetros W e Z , que contêm, no algoritmo que segue, o número máximo pré-estabelecido de níveis na árvore que podem conter um ρ_a -lock e um ε -lock respectivamente. H é um nó auxiliar que contém a altura h da árvore e um ponteiro para a raiz, podendo ser bloqueado.

Figura 4.4: GCC referente a solução 4.

Descrição de Funcionamento

Busca

O algoritmo para a operação de busca é o mesmo apresentado na solução 1 com a diferença de utilizar o ρ_l -lock ao invés do ρ -lock.

Inserção / Remoção

Todo o procedimento é guiado pelos valores dos parâmetros, executando-se ρ_a -locks nos nós que estão até um determinado nível ($\overline{W}$), seguido de α -locks e de ϵ -locks. O passo 9 estabelece novos ϵ -locks em nós que sofrerão modificações, por causa de concatenações ou desdobramentos.

Algoritmo 4.2:

1. Se o valor de W é diferente de zero, então
 - Faça um ρ_a -lock em H ;
 - Senão,
 - Faça um α -lock em H ;
2. Torne H o nó corrente;
3. Faça $\overline{Z} := \min\{h, Z\}$;

$$\overline{W} := \min\{W, h - \overline{Z}\};$$

$$\overline{\alpha} := h - \overline{Z} - \overline{W};$$

4. Execute $\overline{W}$ vezes

Faça um ρ_a -lock no filho imediato do nó corrente a ser utilizado;

Libere o ρ_a -lock do nó corrente;

Transforme o filho do nó corrente no nó corrente;

5. Execute $\overline{\alpha}$ vezes

Faça um α -lock no filho imediato do nó corrente a ser utilizado;

Transforme o filho do nó corrente no nó corrente;

Se o nó corrente está seguro, então

Libere todos os *locks* dos ancestrais do nó corrente;

6. Execute $\overline{Z}$ vezes

Faça um ε -lock no filho imediato do nó corrente a ser utilizado;

Transforme o filho do nó corrente no nó corrente;

Se o nó corrente está seguro, então

Libere todos os *locks* dos ancestrais do nó corrente;

7. Se ainda existe algum nó com ρ_a -lock, então

Libere todos os *locks*;

Faça $W := 0$ e $Z := 0$;

Repita o algoritmo e pare;

8. Se ainda existe algum nó com α -lock, então

Converta, de cima para baixo, todos os ε -locks em α -locks;

Converta, de cima para baixo, todos os α -locks em ε -locks;

9. Faça as atualizações em todos os nós com ε -locks executando outros ε -locks

quando necessário;

10. Libere todos os bloqueios;

Notemos que no passo 8 são executadas conversões de ε para α e depois de α para ε . A primeira conversão serve para permitir que um outro processo, em estado de espera em um nó que contém ε -lock, possa prosseguir. Ao se efetuar a conversão de α para ε em um nó, esse outro processo deverá necessariamente ter concluído o processamento deste nó.

Vantagens e Desvantagens

Reunindo as ideias das outras 3 soluções, a solução 4 permite que seja realizada a parte relativa a alguma das outras, dependendo da situação que melhor se encaixe. Isso é estipulado através dos parâmetros. Bayer e Schkolnick garantem que a solução 4 é livre de deadlock, uma vez que cada uma das outras 3 soluções também são, pois os bloqueios são realizados obedecendo à própria estrutura da árvore. Assim, qualquer processo nunca estará impedido de completar sua tarefa por causa de outros bloqueios requisitados por outros processos.

4.4 Solução devida a Miller e Snyder

Conforme visto nas soluções anteriores devidas a Bayer e Schkolnick [11], um dos seus pontos negativos está no fato das mesmas bloquearem certas páginas da árvore que não sofrerão modificações. Em função desta dificuldade, Miller e Snyder [67] propuseram um método do *tipo 2* para *árvores-B* (e não para B^* como no caso anterior). Neste método, as páginas bloqueadas estão a uma certa distância do nó modificado, diminuindo a porção da árvore bloqueada e aumentando o número de processos que são executados concorrentemente, conforme é retratado pelo diagrama comparativo da porção da árvore que se torna bloqueada em um processo de inserção, ilustrado na figura 4.5.

Figura 4.5: Diagrama comparativo entre a porção bloqueada; (a) solução de Bayer e Schkolnick; (b) solução de Miller e Snyder.

4.4.1 Definições

Na estrutura utilizada para o método cada nó n da árvore apresenta uma fila de pedidos de bloqueios denominada $Q(n)$ e os 3 tipos de bloqueios abaixo, onde o *GCC* está ilustrado na figura 4.6.

Bloqueio de acesso = a ;
 Bloqueio pioneiro = p ;
 Bloqueio seguidor = s ;

Figura 4.6: *GCC* para a solução de Miller e Snyder.

O bloqueio de acesso é utilizado para controlar os nós durante uma busca; já os bloqueios pioneiro e seguidor são usados durante uma inserção ou remoção, como será mostrado adiante. Cada entrada em $Q(n)$ é uma tripla ordenada (r, c, i) onde:

r = tipo de bloqueio requisitado (a, p, s);
 c = tipo de operação a ser executada (inserção, remoção ou consulta);
 i = o valor do índice envolvido;

Todas as operações que envolvem um nó n são inseridas em $Q(n)$ e executadas pela ordem de chegada, isto é, será executado primeiro o pedido que estiver no início da fila.

A solução apresentada por Miller e Snyder faz uso de 2 fluxos:

1. Fluxo de Consulta

Consiste no percurso de um caminho, no sentido para baixo, em uma operação de consulta à árvore. Assim, um determinado processo que necessita buscar um valor i do índice ou localizar a posição para inseri-lo ou removê-lo, deve iniciar fazendo um bloqueio do tipo a (acesso) na raiz da árvore. Caso a mesma não seja o nó procurado, então é feito um pedido de bloqueio de acesso para seu filho apropriado. Essa requisição é colocada na fila Q deste filho, seguida da liberação do bloqueio do nó corrente. O processo flui dessa forma na árvore, bloqueando apenas um vértice por vez, até encontrar o nó apropriado.

2. Fluxo de Mutação

Diz respeito ao percurso de um caminho, no sentido para cima, por um processo que executa as operações de inserção ou remoção. Em virtude do fato das inserções e remoções causarem modificações na árvore, estas operações necessitam que uma parte da mesma seja bloqueada com p ou s (pioneiro ou seguidor).

No caso em que as modificações se propagam na árvore, isto é, as inserções acarretam desdobramentos nos nós e as remoções causam concatenações, ocorrem desbloqueios dos nós mais baixos, enquanto que os ancestrais apropriados são bloqueados. Esse movimento de bloqueio e desbloqueio constitui o fluxo de mutação. Os desdobramentos e concatenações dos nós fazem com que às filas Q associadas aos mesmos também sofram divisões e uniões.

Como os fluxos de consulta e mutação ocorrem em sentidos inversos, pode acontecer um encontro do caminho dos dois durante a execução dos processos. Neste caso, o fluxo de consulta tem permissão para percorrer o sub-grafo bloqueado pelo fluxo de mutação. Isto só é possível graças as filas Q associadas aos nós do caminho de acesso que guardam os pedidos pendentes.

4.4.2 Descrição de Funcionamento

Busca

Como as operações de busca são também executadas como primeiros passos das operações de inserção e remoção, faz-se necessário determinar o tipo de operação para a qual se está buscando um elemento. Quando se trata de um processo de leitura-pura ou remoção, o resultado da busca é um nó contendo um bloqueio de acesso. No caso de uma inserção (processo leitor-atualizador) o procedimento devolve um nó com um bloqueio pioneiro.

A busca tem início na raiz da árvore fazendo-se um bloqueio de acesso no nó corrente, ou colocando-se um pedido de bloqueio na fila do mesmo. O bloqueio do nó corrente é liberado antes de se bloquear o seu filho. Quando este filho é um nó folha e o processo é de inserção, executa-se um pedido de bloqueio pioneiro.

Inserção

O procedimento de inserção começa fazendo uma busca do valor i do índice a ser inserido na árvore. Quando a busca devolver um nó n que não é folha, significa que o valor i

já existe na árvore. Neste caso, o processo desbloqueia n e pára. Entretanto, se o nó n devolvido for uma folha, realizam-se bloqueios pioneiros em até 3 ancestrais próprios de n , isto é, no seu pai, avô e bisavô, além de bloqueios seguidores nos nós irmãos adjacentes de n e de seu pai, conforme ilustrado no trecho de procedimento *BloqIns* e na figura 4.7⁴, onde: $\otimes$ = bloqueio pioneiro, $\odot$ = bloqueio seguidor, $\bigcirc$ = nós desbloqueados.

Na situação em que n é desdobrado em n e n' , o conteúdo da fila $Q(n)$ é separado entre os elementos que permanecerão em $Q(n)$, os que serão inseridos em $Q(n')$ e aquele que será movido para a fila do pai de n . Como o desdobramento de um nó pode se propagar em todo caminho de acesso ascendente na árvore, então a cada divisão de um nó executa-se o fluxo de mutação deslocando-se para cima a sub-árvore bloqueada, da seguinte maneira. Sejam n_1 o nó mais baixo da sub-árvore que contém um bloqueio pioneiro e n_2, n_3, n_4 seu pai, avô e bisavô respectivamente (figura 4.7). Os bloqueios de n_1 e de seus irmãos são liberados, o pai de n_4 é bloqueado com um bloqueio pioneiro enquanto que os irmãos de n_3 com um bloqueio seguidor. Esse mecanismo se repete até que não seja mais necessário se fazer divisões de nós, isto é, detecta-se um nó seguro. No final liberam-se todos os bloqueios ainda existentes.

BloqIns (n):

$n1 := n;$

$n2 := \text{pai}(n1);$

$p(n2);$ /* estabelece um bloqueio pioneiro em $n2$ */

$n3 := \text{pai}(n2);$

$p(n3);$ /* estabelece um bloqueio pioneiro em $n3$ */

$n4 := \text{pai}(n3);$

$p(n4);$ /* estabelece um bloqueio pioneiro em $n4$ */

$s(\text{irmãos}(n1));$ /* estabelece um bloqueio seguidor nos irmãos de $n1$ */

$s(\text{irmãos}(n2));$ /* estabelece um bloqueio seguidor nos irmãos de $n2$ */

⁴Como descrito no procedimento de busca, o nó n recebeu um bloqueio pioneiro no final desta operação, pois ele é uma folha e o processo é de inserção.

Figura 4.7: Esquema de bloqueio no procedimento de inserção.

Remoção

A operação de remoção de um valor i começa fazendo uma busca. Quando o procedimento de busca devolve um nó n não folha, então deve-se procurar o nó folha n' que contém o valor i' , predecessor ou sucessor imediato de i . Antes de iniciar a procura de n' , marca-se i como removido e libera-se o bloqueio de n . Note-se que n não pode permanecer bloqueado durante a busca de i' , pois no momento da troca de i por i' haveria a possibilidade de ocorrência de deadlock. Quando n' for encontrado, executa-se um bloqueio pioneiro nele e no seu respectivo pai. Nesse instante, pode ocorrer que i não esteja mais presente em n , em função da execução de algum outro processo, tendo que ser novamente procurado. Ao se localizar i , efetua-se a troca de i por i' .

Como no procedimento de inserção, executam-se bloqueios pioneiros e seguidores, conforme mostrado no trecho de algoritmo *BloqRem* e ilustrado na figura 4.8, onde: $\otimes$ = bloqueio pioneiro, $\odot$ = bloqueio seguidor, $\bigcirc$ = nós desbloqueados.

Após a remoção de i' , pode ocorrer a necessidade de se efetuar concatenações ou redistribuições envolvendo n' . Quando as concatenações se propagarem, executa-se o fluxo de mutação, deslocando-se para cima a árvore bloqueada. As filas Q dos nós concatenados e redistribuídos também são alteradas.

Algumas vezes podem existir pedidos de acessos pendentes para o valor i' , nos nós localizados no(s) nível(eis) compreendido(s) entre o nível n e n' . Quando i' for inserido em

n , esses pedidos devem ser colocados em posições corretas. Esta operação foi denominada por Miller e Snyder de *intercepção* e independe da reestruturação da árvore.

Figura 4.8: Esquema de bloqueio no procedimento de remoção.

BloqRem (n'):

$n1 := n'$;

$n2 := \text{pai}(n1)$;

$p(\text{pai}(n2))$; /* estabelece bloqueio pioneiro no pai de $n2$ */

$n3 := \text{pai}(n2)$;

$p(\text{pai}(n3))$; /* estabelece bloqueio pioneiro no pai de $n3$ */

$n4 := \text{pai}(n3)$;

$s(\text{irmãos}(n1))$; /* estabelece bloqueio seguidor nos irmãos de $n1$ */

$s(\text{irmãos}(n2))$; /* estabelece bloqueio seguidor nos irmãos de $n2$ */

$n_d := \text{irmão à direita de } n2$;

$s(\text{filhos}(n_d))$; /* estabelece bloqueio seguidor nos filhos de n_d */

Note que quando *BloqRem* é executado, tanto n' como seu pai já possuem bloqueios

pioneiros.

Vantagens e Desvantagens

A solução apresentada bloqueia uma pequena quantidade de nós da árvore a cada operação executada, quando comparada com as soluções apresentadas anteriormente. Com isso há um aumento no número de processos concorrentes, além de permitir processos leitores-puros no percurso de atualizadores.

Os 3 tipos de bloqueios existentes, a combinação em que os mesmos aparecem em cada operação executada e seus requisitos de compatibilidade, evitam a situação de deadlock.

Verifica-se facilmente a simplicidade dos algoritmos de busca e inserção em contrapartida com a complexidade do de remoção. Miller e Snyder propõem que se marque os elementos a serem removidos e apenas quando a quantidade destas marcas forem significativas fazer a reestruturação da árvore, ou seja, removê-los de fato.

4.5 Solução devida a Lehman e Yao

Expandindo algumas ideias de Kung e Lehman [46], onde são discutidos os problemas de concorrência em árvore de busca binária, Lehman e Yao [53] propuseram um método de operações concorrentes em *árvore- B^L* , o qual utiliza um número pequeno (constante) de nós bloqueados para qualquer processo que esteja manipulando a árvore, além de não impedir a leitura de outro nó durante uma determinada consulta. O método caracteriza-se por ser de *tipo 2*.

4.5.1 Definições

Para cada processo existe uma quantidade de memória principal disponível, não compartilhada com outros processos e um número fixo de páginas (nós) que podem ser examinadas simultaneamente por cada processo. Os processos têm permissão de bloquear e desbloquear páginas dos discos, entretanto cada página $\mathcal{P}$ só poderá ser bloqueada por um único processo de cada vez, sem haver o impedimento de leitura destas páginas por outros processos. Os pedidos de bloqueios são executados de maneira disciplinada pela ordem de chegada, ou administrados por um processo supervisor. A estrutura de dados utilizada é *árvore- B^L* (v. 2.8), cuja estrutura foi motivada por este algoritmo de concorrência.

4.5.2 Descrição de Funcionamento

Busca

Se durante uma busca de um valor i em um nó n , o processo detectar que o maior valor existente em n é menor do que i , isso significa que algum outro processo, em execução concorrente, produziu uma mudança em n . Esta mudança ainda não chegou a provocar uma modificação no pai de n , no momento em que o processo de busca estava examinando este pai. Nesses casos, a operação de busca deve visitar o nó vizinho ao nó corrente, auxiliado pelo ponteiro existente para este, ao invés de verificar o nó filho como esperado. Por outro lado, observa-se que a busca não ocasiona nenhum tipo de bloqueio nos nós da árvore durante sua execução, permitindo outros processos de manipularem os nós que estão sendo utilizados pela mesma.

Inserção

Durante o percurso de busca da posição correta para inserção de um valor i , guarda-se em uma pilha P , os nós mais à direita visitados em cada nível ⁵. No momento em que uma folha n for detectada esta é bloqueada e verificada a existência ou não de i na mesma. Se i já pertence a n , então o procedimento é interrompido. Entretanto, na situação em que houver necessidade de se visitar o nó à direita de n , como explicado acima no procedimento de busca, então o mesmo é bloqueado, n desbloqueado e o irmão passa a ser o nó corrente. Este movimento de locomoção horizontal é repetido até o momento em que for encontrada a posição correta de inserção, ou o próprio valor i .

Considere n o nó folha onde i deve ser inserido. Caso n seja *seguro*, então realiza-se a inclusão, n é desbloqueado e o processo se completa. No caso em que n não é seguro, a inclusão ocasiona um desdobramento do mesmo. Este desdobramento propaga-se pela árvore, sendo guiado pelos nós armazenados na pilha P , até que se encontre um nó seguro. O nó desdobrado é desbloqueado após o bloqueio realizado no seu respectivo pai. Em qualquer situação um nó é apenas bloqueado antes de ser modificado.

Lehman e Yao afirmam que no método a inserção ocasiona o bloqueio simultâneo de no máximo 3 nós. Esses bloqueios ocorrem quando a inclusão de um novo valor em um nó proporciona o desdobramento do mesmo e ao se tentar colocar o elemento separador no pai deste, percebe-se que este pai não corresponde à posição correta de inclusão. O nó pai também sofreu um desdobramento provocado pela execução de outro processo

⁵O intuito de se guardar o nó mais à direita visitado deve-se à possibilidade, dada pela estrutura, de se percorrer a árvore horizontalmente em qualquer nível.

concorrente. Com o objetivo de se localizar a posição correta de inclusão, visita-se o nó vizinho desse pai, conforme descrito, fazendo antes um bloqueio no mesmo. Os 3 bloqueios correspondem a uma das metades do nó desdobrado (a metade escrita na mesma página do nó original) e aos 2 nós do nível imediatamente acima desta (nó pai e seu respectivo vizinho).

A corretude do processo de inserção é verificada através do fato de que qualquer nó desdobrado incorpora o ponteiro para o nó vizinho. Por outro lado, observa-se que em função da maneira com que os bloqueios são executados, isto é, seguindo uma ordenação, é possível garantir a ausência de deadlock. Para tal, considere uma ordenação $\prec$ dos nós da árvore, onde:

1. Em qualquer instante t , para 2 nós n_1 e n_2 , que não estão no mesmo nível, diz-se que $n_1 \prec n_2$, (n_1 é menor do que n_2 , n_2 é maior do que n_1), se e somente se, n_2 está a uma distância da raiz menor do que n_1 (n_2 pertence a um nível mais alto na árvore).
2. Se n_1 e n_2 pertencem ao mesmo nível, então $n_1 \prec n_2$, quando n_2 está à direita de n_1 (n_2 é alcançável através de n_1).

Através do algoritmo de inserção e da técnica de desdobramento de um nó, tem-se que:

se $n_1 \prec n_2$ em um instante t_0 , então $n_1 \prec n_2$ em qualquer instante $t > t_0$, pois o desdobramento de um nó n dá origem a 2 nós n'_1 e n'_2 onde:

$$n'_1 \prec n'_2 \text{ e}$$

$$\forall n', n' \prec n \iff n' \prec n'_1$$

$$\forall n', n \prec n' \iff n'_2 \prec n'$$

Dessa forma, como os bloqueios são sempre executados seguindo essa ordem, isto é, bloqueiam-se os nós menores antes dos maiores⁶, então ao ser realizado um bloqueio em um nó n , nenhum nó abaixo ou à esquerda de n será bloqueado, não acarretando o problema de deadlock.

⁶Os bloqueios são executados apenas na fase de reestruturação quando se caminha na árvore de baixo para cima.

Remoção

O método utilizado é simples e permite nós folhas com menos do que K valores do índice. Antes de se remover um valor do índice de um nó folha n , o mesmo é bloqueado.

O método proposto não faz uso das técnicas de concatenação e redistribuição. Lehman e Yao justificam essa ausência em virtude das operações de inserções serem mais frequentes do que as de remoções. Nos casos excepcionais, em que o excesso de remoções causar uma baixa inaceitável de utilização de espaço na árvore, eles sugerem uma completa reorganização através do bloqueio total da árvore.

Vantagens e Desvantagens

Um ponto positivo no método estudado diz respeito à simplicidade do processo de busca. Esse processo comporta-se como um procedimento não concorrente, tratando os ponteiros para os nós vizinhos da mesma maneira que os ponteiros para os nós filhos e sem executar algum tipo de bloqueio.

Outra vantagem está no baixo número máximo de nós bloqueados ao mesmo tempo por um processo e na ordem em que os mesmos são realizados, evitando a situação de deadlock. Porém, a solução dada pelo algoritmo de inserção para encontrar a posição de inclusão, tendo que percorrer a árvore horizontalmente, acarreta um gasto extra de tempo.

A não utilização das técnicas de concatenação e redistribuição no método de remoção, torna-o bastante simples. Entretanto, há uma descaracterização da *árvore-B*, sem falar dos espaços perdidos e na altura da árvore que pode se tornar maior do que o necessário.

Um fato a ser ressaltado é a possibilidade de um processo ficar em execução indefinidamente (livelock). Isso pode ocorrer quando um processo continuar seguindo o caminho formado pelos ponteiros para os nós vizinhos, criados por outros processos. Na prática, essa situação é muito difícil de acontecer, exceto quando os processos são executados com diferentes velocidades. Nesses casos introduzem-se mecanismos especiais, como: atribuir a cada processo uma prioridade baseando-se no seu tempo de vida, isto é, os processos mais antigos recebem prioridades maiores. Assim, garante-se que todos os processos terminarão.

4.6 Solução devida a Sagiv

Esta solução, também de *tipo 2*, consiste em uma melhora daquela proposta por Lehman e Yao [53]. Ela faz uso da mesma estrutura (*árvore- B^L*), porém ocasiona o bloqueio em apenas um nó. Conforme será descrito, a solução executa um processo que redistribui os dados na árvore para garantir que cada nó contenha pelo menos K valores.

4.6.1 Definições

No modelo de Sagiv [84] a *árvore- B^L* apresenta um bloco de memória especial (*bloco especial*), que contém o número de níveis da árvore (altura) e um vetor de ponteiros para cada nó mais à esquerda de cada nível. Como nunca há variação da posição do nó mais à esquerda criado em cada nível, então o bloco especial é modificado apenas quando ocorre um aumento na altura da árvore (desdobramento da raiz). Esta variação consiste na adição de um ponteiro para a nova raiz e do aumento no número de níveis.

Cada nó da árvore só pode ser bloqueado por um único processo de cada vez. Um bloqueio em um nó não impede outros processos de lerem este nó. Os bloqueios dessa solução são usados exclusivamente quando um processo executa uma alteração no nó como uma operação atômica. Essa operação envolve o passo de trazer um nó da memória secundária para a primária, trocar seus dados e depois reescrevê-lo na memória secundária. Isso difere das outras soluções onde o bloqueio de um nó tem a função de garantir a não sobreposição entre os processos.

Cada procedimento de atualização faz uso de uma pilha para guardar todos os últimos nós visitados em cada nível durante a fase 1 de busca do procedimento. O bloco especial permite que um processo desempilhe toda a pilha e nesse instante descubra que uma nova raiz foi criada, permitindo o acesso à mesma.

4.6.2 Descrição de Funcionamento

Busca

A operação de busca dada por Sagiv é a mesma que aquela apresentada por Lehman e Yao. Ela não ocasiona nenhum bloqueio nos nós visitados e quando for necessário percorre-se a árvore horizontalmente, com auxílio dos ponteiros para os nós vizinhos.

Inserção

O processo de inserção assemelha-se bastante com o apresentado por Lehman e Yao. A diferença consiste em se fazer apenas um único bloqueio quando for necessário percorrer a árvore horizontalmente. Seja i o valor a ser inserido e n o nó folha determinado pela busca. Se o maior valor em n for menor do que i , desbloqueia-se n e tenta-se localizar a folha n' correta que deverá conter i , percorrendo-se os nós à direita de n . Ao ser detectada, n' é bloqueada e trazida novamente à memória primária, uma vez que ela pode ter sofrido um desdobramento entre o instante em que foi lida pela primeira vez e o momento em que foi bloqueada, deixando de ser a posição correta de inclusão de i .

Na situação em que um nó folha n torna-se sobrecarregado, a sequência de operações do procedimento de desdobramento é feita de maneira atômica. O nó n deve ser bloqueado no momento em que é detectado, sendo liberado apenas quando for regravado na memória secundária e antes de se executar um bloqueio em qualquer outro nó.

Suponha o caso em que seja necessário criar uma nova raiz⁷. Isto é, deseja-se incluir um par (valor, ponteiro) no nó localizado em um nível acima daquele que sofreu um desdobramento, mas a pilha desse processo que contém os nós do percurso de busca encontra-se vazia e o bloco especial indica que não existem níveis superiores. O procedimento bloqueia a antiga raiz até que a nova seja criada, para evitar a criação de 2 raízes simultaneamente por 2 processos distintos. Em seguida, reserva uma nova página para a nova raiz, faz a redistribuição dos elementos, atualiza o bloco especial e libera, no final, o bloqueio da antiga raiz. O bloco especial não precisa ser bloqueado quando estiver sendo modificado, pois sua alteração só acontece quando a raiz encontra-se bloqueada.

Outro caso também não tratado por Lehman e Yao diz respeito à situação em que a pilha encontra-se vazia, mas um procedimento P requisita uma inserção em um nível acima já existente. Isto ocorre quando há um aumento do número de níveis da árvore durante a execução de P , ocasionada por outros processos. Ao se detectar que a pilha está vazia, o procedimento faz do nó mais à esquerda do nível de cima, o nó corrente, permitindo com isso a movimentação neste nível, até que a posição de inclusão seja localizada. Isso é possível graças ao bloco especial.

Apesar da idéia principal da criação de uma nova raiz ser simples, podem surgir alguns problemas de concorrência. Considere a seguinte situação. Seja r a raiz da árvore e P_1 um processo que está criando uma nova raiz R para a árvore. Suponha que r foi desdobrada em r_1 e r_2 . No caso de existir um processo P_2 desejando inserir um elemento em r_2 , em virtude de um desdobramento de algum filho de r_2 e este já estiver completo, isto deverá causar o desdobramento de r_2 em r_2 e r_3 e uma nova inserção em R . Como P_1 ainda não

⁷Caso ignorado pela solução de Lehman e Yao.

atualizou o bloco especial, então P_2 , ao ler o bloco especial, não encontra o ponteiro para o nível imediatamente mais alto (para R). Uma solução consiste em deixar P_2 em estado de espera por um tempo e em seguida ler o bloco especial novamente. Outra solução é bloquear r_2 quando o mesmo for criado e apenas liberá-lo quando r_1 for desbloqueado, prevenindo P_2 de inserir elementos em r_2 antes da atualização do bloco especial.

Remoção

O procedimento de remover um valor de um nó folha é igual ao apresentado por Lehman e Yao, aceitando também nós com menos elementos do que o permitido. Para amenizar essa situação, Sagiv propôs um processo para redistribuir periodicamente os dados na árvore e garantir que os mesmos estejam com pelo menos metade dos seus elementos. O processo foi denominado de *compressão* e pode ser executado imediatamente após a remoção de algum valor. Ele consiste na redistribuição dos valores de pares de nós irmãos adjacentes. Exigindo o bloqueio simultâneo de 3 nós: o par de irmãos e seu respectivo pai. Devido a esses bloqueios o processo pode ser executado concorrentemente com as operações de busca, inserção e remoção.

Vantagens e Desvantagens

Este método apresenta basicamente as mesmas vantagens e desvantagens vistas na solução de Lehman e Yao, acrescentando os seguintes fatos: cada processo bloqueia apenas um único nó; toma cuidados extras no caso de desdobramento da raiz da árvore e de aumento da altura da mesma.

Sagiv afirma que a solução estudada também está livre de deadlock, pois os processos de inserção e remoção realizam bloqueios em apenas um nó a cada instante. Mesmo no caso da reorganização (compressão) onde são executados bloqueios em 3 nós simultaneamente, não há perigo de deadlock, em função da maneira com que esses são realizados, evitando qualquer situação cíclica. Eles são feitos na seguinte ordem: bloqueio em um nó n , seguido do bloqueio a um filho n_1 de n e finalmente bloqueando-se outro irmão adjacente a n_1 primeiro (de cima para baixo, da esquerda para direita). Observe que a execução concorrente do processo de compressão e de inserção não carrega deadlock. Entretanto, se esse processo de reorganização fosse executado junto com a operação de inserção de Lehman e Yao, poderia ocorrer um deadlock conforme explicado a seguir.

Seja P_1 um processo que está executando uma compressão na árvore e P_2 um processo de inserção de Lehman e Yao. Suponha que em um certo instante t , P_1 realizou um bloqueio no nó n e em um filho n_1 de n , onde n_1 é uma folha. Ao tentar bloquear n_2 , irmão

adjacente de n_1 , P_1 detecta que n_2 já contém um bloqueio realizado por P_2 e permanece em estado de espera. A inserção realizada por P_2 em n_2 acarreta um desdobramento no mesmo fazendo-se necessário bloquear n para inserir o novo separador em n , o qual se encontra bloqueado por P_1 . Notemos que essa situação não ocorre no caso do procedimento de inserção de Sagiv, pois esse só tenta executar um bloqueio em n , para inserir o separador, após o desbloqueio de n_2 .

4.7 Solução devida a Kwong e Wood

Tentando melhorar as soluções de Bayer e Schkolnick [11] e de Ellis [27], Kwong e Wood [48, 49, 50] propuseram uma nova solução de *tipo 1* para as *árvores-B**.

4.7.1 Definições

Baseando-se na idéia da solução 3 de Bayer e Schkolnick (v. 4.3.4), a solução a ser estudada faz uso dos mesmos 3 bloqueios (*leitor*, *alternativo* e *exclusivo*), tendo o mesmo *GCC*.

Para conseguir um bom resultado de concorrência, o método utiliza 3 técnicas. A primeira, denominada por Kwong e Wood de *lock-coupling*, sendo assim conhecida na literatura, foi na realidade proposta por Samadi [85] e Parr [78]. Ela consiste de 2 esquemas: (a) no decorrer da execução de um processo leitor, um nó n é desbloqueado apenas quando seu filho for bloqueado; (b) no decorrer da execução de um processo atualizador, todos os nós do percurso são bloqueados, até que um nó n seguro seja visitado. Neste instante, liberam-se os bloqueios dos ancestrais de n pertencentes ao caminho de acesso. A segunda técnica consiste na idéia apresentada por Bayer e Schkolnick, em que um processo atualizador pode interromper a execução de outro processo, realizando ε -locks nos nós visitados pelo processo atualizador. Essa técnica foi denominada de *driving-off*. A terceira, introduzida por Kwong e Wood, é conhecida como *side-branching*. Ela caracteriza-se por não desdobrar um nó quando o mesmo estiver sobrecarregado, mas copiar metade de seus elementos em um novo nó, deixando o nó original intacto. Quando um nó seguro for encontrado, removem-se, de cima para baixo, as metades apropriadas de cada nó do caminho de acesso.

4.7.2 Descrição de Funcionamento

Por simplicidade os autores da solução assumem que a raiz da árvore é sempre um nó seguro.

Busca

O algoritmo de busca de um valor i , processo leitor-puro, é executado através da técnica de *lock-coupling*. Bloqueia-se a raiz da árvore com um ρ -lock, e indutivamente determina-se o filho apropriado, bloqueia-se-o, desbloqueia-se o pai deste filho e repete-se o procedimento até que uma folha da árvore seja atingida.

Inserção

A fase 1 do procedimento de inserção é executada através da técnica de *lock-coupling*, fazendo bloqueios do tipo α -locks nos nós percorridos. Começando na raiz da árvore até atingir uma folha, faz-se α -locks nos nós do percurso. Quando um nó seguro para inserção for atingido, liberam-se os bloqueios dos ancestrais deste do caminho de acesso.

Quando um nó precisa ser desdobrado, executa-se a técnica de *side-branching*, de baixo para cima, até atingir o nó seguro para inserção mais profundo. Seja n_s este nó. Nesse instante, converte-se o bloqueio de n_s para ε -lock (*driving-off*), insere-se o separador para o novo nó filho em n_s e libera-se o bloqueio de n_s . Indutivamente, esse mecanismo continua, de cima para baixo, até chegar no nível das folhas. A figura 4.9, na qual foi assumido por simplicidade, que as metades dos nós são criados do lado direito, ilustra esse procedimento. Note-se que antes da determinação do nó seguro, o processo está apenas construindo uma parte lateral da árvore que é inacessível a outros processos.

Resumindo os principais passos do método tem-se:

1. Bloqueio do caminho de acesso com α -lock;
2. Construção da cópia das metades dos nós (*side-branching*) que serão desdobrados;
3. Adição, no nó seguro mais profundo, do ponteiro para a última metade criada, após a conversão dos tipos de bloqueios;
4. Remoção das metades apropriadas de cada nó do caminho de acesso, depois de interromper a execução dos processos leitores, através dos ε -locks;

Figura 4.9: Aplicação da técnica de *side-branching* no procedimento de inserção, onde: (a) parte nova criada; (b) reestruturação completa.

Como foi observado os autores supuseram que a raiz é sempre segura. Notemos que isso pode ser conseguido se após a inserção do $2K$ -ésimo valor na raiz esta for imediatamente desdobrada, evitando sua futura sobrecarga.

Remoção

A fase 1 do processo de remoção é semelhante a do processo de inserção, liberando-se do caminho de acesso os bloqueios dos ancestrais de um nó seguro para remoção. Quando a remoção de um valor i , em um nó folha n , torná-lo subcarregado, executam-se concatenações e/ou redistribuições, junto com o método de *side-branching*, da seguinte maneira. Seja n' um nó irmão de n e n_p seu pai. n' recebe um α -lock. Se n' é seguro para remoção, então executa-se uma redistribuição. Convertem-se os α -locks de n_p , n e n' em ε -locks e realiza-se uma redistribuição entre os elementos destes nós. Em seguida, liberam-se os ε -locks de n' e de n_p , os α -locks ainda existentes efetuados por esse processo e o ε -lock de n , nesta ordem. Se n' não é seguro para remoção, então executa(m)-se concatenação(ões). Converte-se o bloqueio de n' para um ε -lock e colocam-se todos os elementos de n , exceto o valor a ser removido, juntamente com o separador do nó pai de n , em n' . Em seguida, libera-se o ε -lock de n' fazendo de n_p o nó corrente. Note-se que o nó n torna-se redundante, porém é deixado temporariamente intacto na árvore. A figura 4.10 ilustra a concatenação dos nós através da técnica de *side-branching*, onde β é o conteúdo de n' ; * representa os elementos a serem removidos de n e γ é o conteúdo do

nó n (corrente) sem o par (i^*, p^*) . O procedimento continua sendo executado na árvore, no sentido para cima, até que o nó corrente seja considerado seguro para remoção. Ao se detectar este nó seguro faz-se um ε -lock no mesmo, (*driving-off*), retira-se o par separador (valor, ponteiro) apropriado e desbloqueia-se-o. Indutivamente, segue-se na árvore, de cima para baixo, fazendo as conversões dos tipos de bloqueios nos nós que passaram a ser redundantes do caminho de acesso, seguindo da eliminação dos mesmos, conforme verificado na figura 4.11.

Figura 4.10: Concatenação dos nós através da técnica *side-branching*.

Resumindo os principais passos do método tem-se:

1. Bloqueio do caminho de acesso com α -lock;
2. União dos nós irmãos mantendo os originais (*side-branching*), até determinar um nó seguro, seguido da conversão do tipo de bloqueio deste nó seguro e execução do procedimento apropriado;
3. Remoção dos nós redundantes do caminho de acesso depois da interrupção da execução dos processos leitores, através dos ε -locks;

o processo desloca primeiro o valor seguido do ponteiro associado ao mesmo, isto é, move-se o valor 10, depois o ponteiro p_3 , valor 5, ponteiro p_2 e assim sucessivamente. A figura 4.12 (b) mostra o nó após o instante em que o valor 5 foi deslocado e antes de p_2 ter sido copiado. Neste momento, o processo leitor, dando continuidade a busca do valor 6, prosseguirá pelo nó apontado por p_3 ao invés de p_2 , pois p_3 está associado aos valores compreendidos entre 5 e 10, ocasionando um erro.

Esse problema pode ser “parcialmente” resolvido se o processo atualizador deslocar primeiro o ponteiro e depois o valor, como mostrado no exemplo 2.

Exemplo 2

Considerando o nó da figura 4.12 (a) e fazendo o deslocamento do ponteiro antes do valor, tem-se a movimentação de p_3 , seguida do valor 10, de p_2 , valor 5 e assim por diante. Suponha o nó mostrado na figura 4.12 (c), logo após o deslocamento do valor 5. O processo P_1 continuando a busca para o valor 6 prosseguirá pelo ponteiro correto. Todavia, ainda existem informações erradas que podem atrapalhar outros processos leitores. Essa técnica foi denominada de *técnica de proteção* (“shielding technique”), a qual sozinha não garante a corretude dos processos leitores.

Exemplo 3

Considerando o mesmo nó da figura 4.12 (a), os mesmos processos e a situação após o ponteiro p_3 , valor 10 e ponteiro p_2 terem sido deslocados, mas antes do valor 5 ter sido copiado, tem-se a situação da figura 4.12 (d). O processo P_1 sendo executado da esquerda para direita, localizará a posição onde está o ponteiro p_2 , após detectar que o mesmo localiza-se no intervalo entre 5 e 10 e tentará ler o nó apontado por p_2 . Entretanto, antes de fazer esta leitura, o processo P_2 continua movendo os valores e os ponteiros, deslocando o valor 5 e o ponteiro p_1 (figura 4.12 (e)). Dessa forma, a continuidade do processo P_1 causará a busca na sub-árvore errada. Note-se que esse tipo de problema é consequência direta da natureza assíncrona dos processos envolvidos.

A técnica de *adição R^nW* tenta corrigir esses problemas, guardando em uma variável extra o valor associado ao último ponteiro visitado. Por exemplo, para o ponteiro p_j que aponta para a sub-árvore contendo os valores do índice compreendidos entre i_j e i_{j+1} , guarda-se i_{j+1} na variável auxiliar ($0 \leq j \leq 2K$). Assim, após a determinação do ponteiro, compara-se o conteúdo da variável auxiliar com o valor associado a este ponteiro. Se os valores forem iguais, então não houve deslocamentos provocados por algum processo atualizador. Nos casos de diferenças entre os valores, o procedimento não aceita aquele ponteiro e reinicializa o passo de localização da posição correta através da leitura do próximo valor à direita. Esse mecanismo é chamado de *método de leitura repetitiva* (“repeated reading technique”).

Para a situação do exemplo 3, tem-se que o processo P_1 armazena o valor $i_3 = 10$ na

variável auxiliar. Quando P_1 for fazer o acesso ao nó referenciado por p_2 , o procedimento compara o conteúdo da variável auxiliar com o atual valor de i_3 , que no caso é igual a 5. Como são diferentes, o processo dará continuidade lendo o valor à direita, ao invés de prosseguir pelo nó referenciado por p_2 .

Note-se que cada processo atualizador ao adicionar um par (ponteiro, valor) pode causar a um processo leitor no máximo uma repetição de leitura de um ponteiro. Isso parece razoável quando comparado com a eliminação da restrição de indivisibilidade.

Figura 4.12: Ilustração dos exemplos 1, 2, 3.

Vantagens e Desvantagens

A solução apresentada oferece um alto grau de concorrência sem causar modificações na estrutura da árvore, em função da técnica de *side-branching*. Permite também que processos leitores-puros sejam executados no mesmo local de atuação dos processos atualizadores em qualquer uma de suas 2 fases.

De acordo com análises feitas por Kwong e Wood, o número de bloqueios do tipo ε -lock executados em cada nó foi reduzido em 50% quando comparado com a solução de Ellis. Isto se deve ao uso da técnica de *side-branching*, onde os nós só recebem ε -locks uma única vez, quando faz-se necessário afastar os processos leitores para remoção das metades redundantes.

A principal desvantagem do método de *side-branching* está relacionada com o fato do aumento do número de passagens dos processos atualizadores em seus caminhos de

acessos; cada nó é visitado uma vez a mais, no sentido de cima para baixo na árvore, no momento de remoção dos valores considerados desnecessários (repetidos). Como cada nó representa uma página, tem-se um aumento na transferência de páginas entre a memória principal e a secundária. Kwong e Wood afirmam que, em média, os níveis mais baixos da árvore contêm nós seguros. Dessa forma, é provável que todos os nós do caminho de acesso de um processo atualizador, transferidos para a memória principal permaneçam na mesma até que a reestruturação se complete.

Outra vantagem diz respeito à técnica de *adição R^nW* que resolve os problemas de concorrência intra-nós entre processos leitores e atualizadores, sem precisar fazer leitura e escrita dos valores e ponteiros de forma atômica.

4.8 Outras Soluções

Baseando-se na idéia de Guibas e Sedgewick [32], de executar pré-desdobramentos dos nós completos durante a fase 1 dos processos de atualização, Mond e Raz [70] propuseram um mecanismo de controle de concorrência para as *árvores- B^** , o qual não é considerado nem de tipo 1 nem de tipo 2.

Eles sugeriram uma pequena modificação na *árvore- B^** de ordem K , na qual cada nó contém no máximo $2K + 1$ valores do índice e no mínimo $K - 1$, excetuando a raiz que pode conter 1 elemento. As operações de inserção e remoção são as mesmas realizadas na *árvore- B^** com a diferença de que durante a fase 1 de um processo atualizador o procedimento executa desdobramentos e concatenações nos nós percorridos no caminho de cima para baixo. Assim, quando o processo é de inserção (remoção) e um determinado nó do percurso contém $2K + 1$ ou $2K$ ($K - 1$ ou K) valores, então realiza-se uma divisão (concatenação) do mesmo (com seu irmão adjacente, caso seja possível). Note-se que os desdobramentos (concatenações) não se propagam na árvore.

O método realiza 2 tipos de bloqueios: de leitura (ρ -lock) e de atualização que corresponde ao bloqueio exclusivo das soluções de Bayer e Schkolnoick, Kwong e Wood (ε -lock). Um nó com ρ -lock só pode ser usado por outros processos de leitura; um nó com ε -lock não pode ser usado por nenhum outro tipo de processo. Esses bloqueios são realizados a partir da raiz, onde o nó corrente e seu respectivo pai são bloqueados com o tipo de bloqueio apropriado, dependendo da natureza do processo sendo executado. Antes de se examinar o próximo nó no caminho, filho do nó corrente, libera-se o bloqueio do pai deste. Observe que dessa maneira apenas 2 nós estão bloqueados simultaneamente pelo mesmo processo. O mecanismo não acarreta deadlock, pois os processos são tratados de forma independente e a estrutura da árvore introduz uma ordenação nos bloqueios.

Mond e Raz citam as teses de mestrado de Mond [69] que implementou um sistema de simulação para o método e de Navon [73]⁸, que também realizou outra implementação com sucesso. Eles verificaram que as operações preparatórias acarretam gastos extras em função do aumento do número de operações executadas na árvore. Todavia, à medida que a árvore aumenta, o número relativo de nós não seguros diminui.

Keller e Wiederhold [42] afirmam que o método de Guibas e Sedgewick e de Mond e Raz apresenta falhas no caso em que os valores dos nós internos são de tamanho variável. Eles acreditam que nestas situações é impossível garantir que o novo valor do índice a ser inserido em cada nó caberá no espaço livre existente, mesmo após um pré-desdobramento. Cremos que essa dificuldade pode ser contornada se o pré-desdobramento for executado de maneira a deixar, em cada nó, um espaço livre igual ao tamanho máximo definido para os valores de um índice. No capítulo 6 abordaremos essa idéia de forma mais detalhada.

Keller e Wiederhold sugeriram a solução denominada de *promoção vizinha* (“sibling promotion”), a qual foi baseada na técnica desenvolvida por Nurmi, Soisalon-Soinien e Wood [75]⁹. Esta última consiste de uma expansão do método de remoção proposto por Sagiv.

Nurmi et al. propuseram para os casos de inserções que acarretariam uma divisão do nó pai, a criação de um nó temporário, o qual comportaria apenas 1 separador com 2 ponteiros, uma para cada um dos nós resultantes do desdobramento do nó filho. Desta forma a divisão não se propagaria possibilitando o aumento do número de procedimentos concorrentes. Entretanto, para conseguir esse efeito, faz-se necessário a inserção de um bit de marcação no pai, para indicar que no futuro o nó sofrerá uma divisão e que há um temporário associado a ele. Esse nó pai é desdobrado e o temporário é removido quando qualquer processo percorrer a árvore e encontrá-lo.

Já a técnica de *promoção vizinha* cria um ponteiro temporário para o nó vizinho do nó filho, quando o pai encontra-se completo. Toma-se metade das informações do nó filho que será dividido colocando-as em um novo nó, vizinho ao primeiro, criando um apontador para este. Em seguida, marca-se o nó pai deste filho indicando que o mesmo necessita ser dividido. No próximo acesso ao nó pai, por algum processo, esta marcação é retirada e o ponteiro do nó vizinho removido para o nó pai. Se esse último passo ocasionar uma sobrecarga no nó avô, o algoritmo é novamente utilizado. Note-se que essa técnica não atrapalha as operações de busca, pois os ponteiros para os vizinhos se comportam como ponteiros para os filhos.

A figura 4.13 demonstra este processo, representando os nós avô, pai e filho nas se-

⁸Não tivemos acesso a essas teses.

⁹Em virtude da impossibilidade de localização do artigo desses autores, fomos obrigados a nos limitar às citações feitas por Keller e Wiederhold.

guintes situações: (a) deverão sofrer modificações; (b) após o desdobramento; (c) após a modificação em todo o trecho. O (*) significa *particionamento pendente*.

Figura 4.13: Demonstração da técnica de promoção vizinha.

Em função da dificuldade de determinar se um nó é ou não seguro em estruturas de *árvores-B* que armazenam registros de tamanho variável, Souza e Carvalho [89] propuseram algoritmos de busca e inserção para acesso concorrente que não utilizam o conceito de nó seguro. Esses algoritmos permitem o uso da técnica de overflow quando um nó se encontra sobrecarregado. Notemos que a técnica de overflow (v. 1.4.2) não foi usada em nenhuma das soluções para o problema de concorrência existentes na literatura, pois poderia ocasionar a situação de deadlock.

Os algoritmos propostos são considerados “híbridos”, pois adotam características das soluções de *tipo 2* durante a fase 1 de um processo atualizador e de *tipo 1* na fase 2. Eles fazem uso dos 3 tipos de bloqueios: ρ -lock, α -lock e ε -lock.

Para facilitar a detecção de alterações do caminho de acesso e do escopo de um processo, foram criados 2 campos extras em cada nó da árvore: um campo *bit-modificação* e outro *time-stamp*. O primeiro tem a função de detetar alterações no caminho de acesso, sendo apenas ligado quando ocorrer uma modificação em um nó durante alguma reestruturação. Todos os procedimentos de inserção desligam, no final de sua execução, os bits-modificação que tenham sido ligados. O campo *time-stamp* tem a função de detetar alterações no escopo e é incrementado sempre que um nó for alterado por um procedimento de inserção. A detecção é feita comparando-se os *time-stamps* guardados durante a fase de busca de um processo de atualização com aqueles lidos antes de se entrar na fase de reestruturação.

O algoritmo de busca de um valor i , processo leitor-puro, é igual ao apresentado por K-wong e Wood [50] (v. 4.7.2). Sempre que um nó for visitado verifica-se seu *bit-modificação*. Se este estiver ligado, desfaz-se o bloqueio do nó e recomeça-se o procedimento pela raiz. Essa solução não é ineficiente, pois o sistema caracteriza-se por guardar na memória primária, para uso compartilhado, todos os nós do caminho de acesso a serem usados

pelos processos em execução.

Os procedimentos de inserção são divididos em 3 fases: *fase de busca*, igual a fase 1; *fase de alteração simples*, que corresponde a inserção de um novo registro sem causar reestruturação na árvore e *fase de reestruturação*, executada quando um nó torna-se sobrecarregado. Nesse caso, o procedimento tenta fazer uma redistribuição com a técnica de overflow com cada vizinho imediato ou, quando isso não for possível, é executado um desdobramento.

A fase de busca de um processo atualizador é igual ao algoritmo de busca de um processo leitor-puro, fazendo-se um α -lock no nó folha n atingido e guardando-se os time-stamps dos nós do caminho de acesso.

A fase de alteração simples caracteriza-se pela conversão do tipo de bloqueio em n , isto é, de α -lock para ε -lock, seguido da inserção do registro, incremento do time-stamp de n e liberação do bloqueio, encerrando o procedimento.

Antes de dar início a fase de reestruturação, o procedimento libera o bloqueio de n e executa um ε -lock em todo o caminho de acesso¹⁰. Depois compara os time-stamps guardados na fase de busca. Se algum time-stamp foi modificado, desbloqueia-se o caminho e recomeça-se da fase de busca; senão inicia-se a fase de reestruturação.

Na fase de reestruturação o procedimento realiza um α -lock em um irmão adjacente n' de n e converte o ε -lock de n em um α -lock. Observemos que isso não está claro no artigo, sendo uma distinção adicional desse método em relação aos demais encontrados na literatura. No caso de haver possibilidade de fazer redistribuição realizam-se conversões nos tipos de bloqueios (de α -lock para ε -lock) de n e n' , nesta ordem. Após a redistribuição, ligam-se os bits-modificação e incrementam-se os time-stamps desses nós. No final, liberam-se os bloqueios de n e n' e tenta-se inserir o novo separador no respectivo nó pai, que se torna o novo n . Se necessário, executam-se os mesmos passos de redistribuição para este nó pai.

Por outro lado, quando ocorrer um desdobramento, converte-se o α -lock do nó n sobrecarregado em um ε -lock, cria-se um novo nó, liga-se o bit modificação e atualiza-se o time-stamp de n , liberando-se depois o bloqueio. No final do procedimento de atualização desligam-se todos os bits-modificação.

Os autores da solução apresentada afirmam que não há deadlock, pois os bloqueios são realizados seguindo-se uma determinada sequência. Na fase de busca os bloqueios são executados na ordem do caminho de acesso; na fase de reestruturação só são realizados

¹⁰Notemos que no artigo original os autores mencionam esse ε -lock para toda a árvore, o que deve ser um erro.

bloqueios em um nível quando forem liberados todos os bloqueios do nível inferior. Consideramos essa argumentação insuficiente e acrescentamos que não há perigo de deadlock, pois a única possibilidade de uma situação cíclica seria na tentativa de se fazer uma redistribuição (overflow) por algum processo de inserção. Entretanto, como essa redistribuição é realizada entre 2 irmãos adjacentes cujo pai contém um ε -lock é impossível que um desses irmãos se encontre bloqueado por outro processo.

4.9 Comparação entre as Soluções

Kwong e Wood [50] fizeram uma comparação entre as soluções classificadas de *tipo 1*. A tabela 4.1 relata essa comparação através das propriedades dadas abaixo.

Propriedades:

1. A solução permite outros processos atualizadores no caminho de acesso de algum processo atualizador.
2. A solução permite ultrapassagem e concorrência intra-nós durante a execução de processos leitores-puros.
3. A solução permite processos leitores-puros, no caminho de acesso de um processo atualizador, durante a fase de busca deste (fase 1).
4. A solução permite processos leitores-puros, no caminho de acesso de um processo atualizador, durante a fase de redistribuição deste (fase 2).

	Soluções do <i>tipo 1</i>			
	Samadi/ Parr	Bayer/ Schkolnick	Ellis	Kwong/ Wood
1	Não	Não	Não	Não
2	Não	Sim	Sim	Sim
3	Não	Sol 1 - Não Sol 2 - Não Sol 3 - Sim	Sim	Sim
4	Não	Não	Sim	Sim

Tabela 4.1: Comparação de soluções do *tipo 1*.

Notemos que as soluções de tipo 2 (Miller e Snyder [67], Lehman e Yao [53] e Sagiv [84]), satisfazem todas essas propriedades. Por outro lado, Lehman e Yao e Sagiv afir-

mam que suas soluções permitem um número maior de procedimentos sendo executados simultaneamente, mesmo quando comparadas com as soluções do tipo da de Guibas e Sedgwick [32] e de Mond e Raz [70]. Estas últimas necessitam de um espaço maior de memória e ocasionam um aumento no tempo de acesso.

De acordo com Srinivasan e Carey [90], com exceção feita ao trabalho realizado por Johnson e Shasha [40]¹¹, até pouco tempo atrás não existia uma análise bem feita sobre o desempenho dos algoritmos de concorrência para as *árvores-B* e suas variações. Em 1991, eles estudaram o desempenho de vários desses algoritmos, através do uso de um modelo de simulação para as operações da *árvore-B* em um sistema gerenciador de bancos de dados centralizado.

O modelo era composto por filas e vários terminais, com uma ou mais CPU's e discos, onde os pedidos para as CPU's são escalonados através da disciplina PEPS (FIFO), primeiro a entrar primeiro a sair. Cada disco contém uma fila, também manuseada pela mesma técnica. Por sua vez, o buffer do sistema é gerenciado pela estratégia LRU (least recently used). Um terminal pode requisitar qualquer uma das operações básicas da *árvore-B* (busca, inserção, remoção e atualização) através de transações, executadas uma por vez. O desempenho dos algoritmos de concorrência é dependente de certas características como por exemplo: quantidade das operações básicas, recursos do sistema (número de CPU's, discos e buffers), estrutura da *árvore-B* (tamanho do nó, número inicial de registros de dados) e nível de multiprogramação (número de terminais do sistema). Com base nesses aspectos, Srinivasan e Carey chegaram aos seguintes resultados:

1. Em um sistema composto por uma única CPU e um disco, não existe diferenças significativas no desempenho dos vários algoritmos de concorrência.
2. Quando se usa a técnica de *lock-coupling* com bloqueios do tipo exclusivo o desempenho é péssimo, mesmo quando as operações de buscas são dominantes, pois os procedimentos de atualizações não fazem uso das vantagens do paralelismo.
3. Os gastos com reinicializações (caso do algoritmo de Bayer e Schkolnick) e com os caminhos percorridos nas listas ligadas formadas pelos nós de um mesmo nível (caso dos algoritmos de Lehman e Yao e Sagiv), não afetam diretamente o desempenho dos algoritmos. Os algoritmos que fazem uso da estrutura *árvore-B^L* são considerados os melhores.

¹¹Este artigo não foi localizado.

Capítulo 5

EXEMPLOS DE SISTEMAS COMERCIAIS

Comer [19] afirmou: “The B-Tree is, de facto, the standard organization for indexes in a database system”. Apesar de relativamente antiga, essa afirmação ainda é válida, haja visto que as *árvores-B* continuam sendo a forma de organização de índices em talvez todos os sistemas gerenciadores de bancos de dados. Abordaremos aqui, para efeito ilustrativo, características de dois sistemas comerciais de gerenciadores de arquivos que fazem uso da estrutura *árvore-B*, ou de alguma variação da mesma. Escolhemos os sistemas BTRIEVE [74] e VSAM [37, 38, 39, 41], em virtude da facilidade do levantamento bibliográfico e não por serem mais ou menos importantes do que os demais existentes. Nessa abordagem procuramos mostrar não somente os aspectos relevantes às estruturas de *árvores-B*, mas também aqueles que caracterizam a manipulação dos dados, para se ter idéia das funções que esses produtos contêm, quando interessantes.

5.1 Sistema BTRIEVE

5.1.1 Introdução

Desenvolvido pela NOVELL INC.¹, o sistema BTRIEVE consiste de um pacote de rotinas para gerenciar arquivos indexados com *árvores-B*, para os sistemas operacionais: MS-DOS, OS/2 ou XENIX.

¹A NOVELL INC. concedeu licença a ABCOM INFORMÁTICA LTDA para comercializar o sistema BTRIEVE no Brasil, bem como reproduzir seu manual.

Aparentemente, a estrutura é a da *árvore-B* (v. 1.1.2). Deduzimos este fato devido à informação do manual [74] (seção *Estimating File Size*), onde a cada valor de índice na árvore são associados 8 bytes. Provavelmente 4 para cada ponteiro: um ponteiro para o nó descendente e o outro para o registro de dados. Também aparentemente, quando um índice for declarado como tendo valores duplicados, o sistema cria uma cadeia duplamente ligada entre os registros de dados, pois neste caso os registros contêm 8 bytes adicionais para cada índice desse tipo.

O sistema é composto por um programa gerenciador de registros denominado de BTRIEVE.EXE, o qual deve estar sempre residente na memória. O BTRIEVE.EXE contém 36 operações distintas para manusear os arquivos (v. 5.1.4). Estas operações são executadas por chamadas à função ou subrotina BTRV, em um programa de aplicação, isto é, programa escrito em linguagens hospedeiras (BASIC, PASCAL, COBOL, C e ASSEMBLY), havendo interfaces para vários compiladores de cada uma delas. Os argumentos de BTRV especificam a operação a ser executada, bem como as informações necessárias a esta. Além da manipulação de dados, a BTRV retorna um inteiro indicando o sucesso ou insucesso (status) da operação. Nas chamadas da BTRV, os índices são identificados por um número estabelecido pela ordem de declaração destes.

Os principais passos do funcionamento do sistema através de um programa de aplicação podem ser resumidos em:

1. O programa de aplicação executa uma instrução do BTRIEVE na forma de uma chamada à função ou subrotina BTRV com os argumentos necessários.
2. Uma rotina de interface existente no programa de aplicação armazena estes argumentos em uma área de memória e faz uma chamada para o programa gerenciador de registros.
3. BTRIEVE faz a validação dos argumentos e em seguida executa as instruções.
4. Em seguida, BTRIEVE retorna os dados apropriados e devolve o controle para o programa.
5. O programa de aplicação faz uso dos resultados da operação executada.

Além do BTRIEVE.EXE o sistema contém 2 programas utilitários: BUTIL.EXE e B.EXE, que permitem a execução do gerenciamento de arquivos e testes sem que haja necessidade de escrever um programa de aplicação. O BUTIL.EXE contém uma série de comandos para realizar a criação, manutenção e recuperação dos arquivos (v. 5.1.4). O B.EXE é usado interativamente através de menus, permitindo a execução das mesmas operações do BTRV, sem necessidade de se chamar essa rotina de dentro de um programa

em linguagem hospedeira. Assim, é possível testar a lógica de manipulação de arquivos de uma aplicação antes de se programar os aplicativos correspondentes.

5.1.2 Características Gerais do Gerenciador de Arquivos

No sistema BTRIEVE um *arquivo* consiste do conjunto de dados de maior nível ao qual se pode ter acesso, sendo formado por um conjunto de *páginas*. Um arquivo pode ser criado pelo comando CREATE do utilitário BUTIL.EXE, ou pela execução da rotina BTRV com um parâmetro de operação CREATE dentro de um programa de aplicação.

As *páginas* são as unidades transferidas entre a memória principal e secundária e são classificadas em: *página de cabeçalho* (*registro de controle de arquivo*), *página de dados* ou *página de índices*. O tamanho das páginas são especificadas durante a criação do arquivo.

1. Página de Cabeçalho

Presente em todos os arquivos do sistema BTRIEVE, constitui a primeira página de um arquivo. Contém informações sobre as características do arquivo, como tamanho do arquivo, tamanho da página, tamanho do registro, entre outras.

2. Páginas de Dados

Todos os registros de uma aplicação são armazenados em páginas de dados, que podem ser *páginas de registros de tamanho fixo* ou *páginas de registros de tamanho variável*. Os registros de tamanho variável devem obrigatoriamente conter uma parte inicial (à esquerda) com tamanho fixo. Quando um arquivo comportar registros de tamanho variável e/ou compressão em seus dados, ele conterá os dois tipos de páginas: a das partes fixas dos registros e a outra contendo a porção variável dos registros. Na situação em que a parte de tamanho variável exceder o tamanho da página definida para o arquivo, as informações da mesma são divididas em quantas páginas forem necessárias para armazenar o restante dos dados.

3. Páginas de Índices

Contêm os valores do índice e ponteiros associados para permitir o acesso aos nós da árvore e eventualmente aos registros de dados, estruturados como uma *árvore-B*.

Quando um arquivo contendo dados é criado, o usuário pode definir um ou mais valores para construir os índices iniciais. Estes índices são chamados, na nomenclatura do BTRIEVE, de *índices permanentes* e não podem ser removidos. O sistema permite que sejam definidos outros índices internos para um arquivo, após a criação do mesmo. Estes são denominados de *índices suplementares* e podem ser removidos sempre que a

aplicação não precisar mais deles. Seu espaço é aproveitado para novos dados ou para outras páginas de índices.

Os arquivos são classificados dependendo do tipo de página existente. Eles podem ser:

1. Arquivo Padrão

Contém uma página de cabeçalho seguida de páginas de índices e de dados, podendo ter registros de tamanho fixo e variável, com pelo menos um índice.

2. Arquivo Simples de Dados ("Data-only Files")

Este arquivo contém uma página de cabeçalho e as informações dos registros de dados permitindo, portanto, somente acesso sequencial. O acesso a esses registros é realizado pelas operações de STEP (v. 5.1.4). Em qualquer instante pode-se adicionar um índice a este arquivo, permitindo o acesso pelas operações GET.

3. Arquivo de Valores de Índices

Neste tipo de arquivo os dados resumem-se aos valores de um índice. É formado por uma página cabeçalho e por uma ou mais páginas de índices. Seus registros devem ter tamanho de 255 bytes; surpreendentemente os dados, uma vez inseridos, não podem ser atualizados ou removidos "manualmente". As únicas operações possíveis são GET, STEP e INSERT (v. 5.1.4). Contém valores únicos do índice (chaves).

Os arquivos que possuem registros de tamanho variável deixam uma certa quantidade de espaço livre em cada página, para permitir expansão destes registros durante alguma atualização. Este espaço é determinado pelo usuário durante a criação do arquivo.

A recuperação de um registro pode ser executada em 3 formas: sequencial, indexada e direta. Na primeira usam-se as operações de localizar o primeiro ou último registro, seguidas de operações STEP para frente ou para trás. No segundo caso são usadas várias formas da operação GET (v. 5.1.4) que referencia um índice. No acesso direto, após uma operação qualquer de busca de um registro, o seu endereço pode ser obtido usando uma operação especial (GET POSITION, v. 5.1.4). Uma outra operação (GET DIRECT) permite a utilização deste endereço para recuperar o registro em acesso direto. Esta operação possibilita também que se estabeleça um ponto de referência na *árvore-B* de um índice qualquer. Uma operação subsequente de busca, referenciando esse índice, partirá daquele ponto, nos casos de busca sequencial pelo índice.

O sistema utiliza uma área de memória denominada de *bloco de posicionamento*, que contém ponteiros para o caminho percorrido na *árvore-B* do último índice utilizado, a identificação deste índice e 3 ponteiros que apontam para o nó corrente, anterior e próximo na árvore. Consideramos que a manutenção desses 3 ponteiros é devida à utilização da

árvore-B em lugar da *árvore-B** ou *árvore-B⁺*, pois o acesso sequencial deve exigir a leitura de um nó pai para caminhar-se de um filho para seu irmão adjacente. .

A inserção de um registro em um arquivo BTRIEVE é realizada na primeira posição disponível, independente do valor do índice associado ao mesmo. Esta posição consiste da *localização física* ou *endereço do registro*, onde ele permanecerá até o momento em que seja removido. Observemos que não há índices primários.

Os valores do índice podem ter 6 tipos de atributos distintos, especificados na criação do arquivo e definidos a seguir.

1. Duplicado

O atributo padrão (“default”) é de valores únicos (não duplicados). Se um valor do índice foi definido como duplicado, então podem existir vários registros distintos com o mesmo valor associado.

2. Modificável

Permite modificações nos valores do índice durante a atualização dos registros. O padrão é que os valores não sejam modificados; nesse caso, uma modificação pode ser obtida eliminando-se previamente o registro, e em seguida inserindo-se o mesmo com as alterações.

3. Segmentado

Permite índices compostos, isto é, formado por valores de vários campos do registro, não necessariamente contíguos. Na nomenclatura do BTRIEVE esse índice chama-se de *segmentado*.

4. Descendente

O padrão é ordenar os valores ascendentemente, porém pode-se especificar uma classificação por ordem descendente.

5. Nulo

BTRIEVE permite que se defina um valor como zero, branco ou vazio como valor nulo. Assim, se um registro contém todos os bytes do valor de um índice igual ao valor nulo, este registro não será incluído, ou será excluído do índice. Quando o valor for segmentado, todos os bytes de cada segmento devem conter o valor nulo para que o registro seja removido.

6. Manual

Consiste de uma forma modificada do atributo nulo. Neste caso, o registro será removido do índice quando todos os bytes de um único segmento do valor do índice contiverem valor nulo.

Um arquivo pode ter redução na utilização de espaço em um disco, através do uso de um dos 3 métodos dados a seguir.

1. Truncamento de Brancos

Aplicado apenas aos arquivos que contêm registros de tamanho variável, este método elimina todos os caracteres brancos à direita da parte variável nos tipos cadeias de caracteres. Quando o registro é lido, isto é, trazido à memória principal, o mesmo é expandido à sua forma original.

2. Compactação de Índices

A técnica de compactação consiste do método de overflow (v. 1.4.2) aplicado quando uma página de índices se torna sobrecarregada.

3. Compressão de Dados

Ao criar um arquivo é possível especificar se os registros de dados serão ou não comprimidos. Esta compressão produz uma “redução significativa” do espaço necessário para armazenar registros, a qual aparentemente restringe-se à eliminação de caracteres repetidos. O sistema utiliza um *buffer de compressão*, isto é, uma área da memória principal reservada para executar a expansão do registro comprimido. O tamanho deste buffer é 2 vezes o tamanho do maior registro inserido no arquivo que contém dados comprimidos. O arquivo que possui dados comprimidos é criado como um arquivo de registros de tamanho variável, pois o tamanho final de um registro comprimido não pode ser conhecido até o momento em que o mesmo é gravado no arquivo.

5.1.3 Controle de Integridade, Concorrência e Segurança

Integridade

Para garantir a integridade dos arquivos, o sistema BTRIEVE faz uso de duas técnicas denominadas de *pré-imagem* e de *transações lógicas*.

1. Pré-imagem

Esta técnica visa proteger as operações individuais de inserção, atualização e remoção contra falhas do sistema, restaurando o arquivo ao estado anterior à execução de alguma operação. O processo consiste em criar um arquivo denominado de *pré-imagem* sempre que houver acesso a um arquivo pela primeira vez. Na memória principal aloca-se uma área de buffer reservada para este arquivo. Durante as atualizações seguintes, o sistema armazena as informações atualizadas em uma área

da *memória cache* e antes de executar a inserção da página na memória secundária, coloca uma cópia desta no buffer do arquivo de pré-imagem. No final da operação o sistema copia o conteúdo do buffer do arquivo de pré-imagem no próprio arquivo de pré-imagem da memória secundária e depois modifica o arquivo de dados. Se antes do término da operação alguma das áreas da memória principal (cache ou pré-imagem) ficarem cheias, então as informações destas são colocadas no arquivo pré-imagem da memória secundária, para liberar espaço. Para aumentar o desempenho dos arquivos, eles podem ser abertos em *modo de acesso acelerado*, isto é, as páginas só serão escritas na memória secundária quando um buffer estiver cheio.

2. Transações Lógicas

Para se conseguir um nível maior de integridade pode-se usar transações lógicas dentro do programa. O método de transação realiza o agrupamento de um conjunto de operações dentro das operações delimitadoras BEGIN TRANSACTION/END TRANSACTION (v. 5.1.4) da chamada de uma transação. As operações localizadas dentro destes delimitadores só serão consideradas completas quando todas forem executadas com sucesso. Se o sistema for interrompido no meio da execução de alguma operação da transação, então todas as operações executadas dentro daquela transação serão desfeitas, usando-se o arquivo de pré-imagem para restaurar o banco de dados à situação em que ele se encontrava quando foi executado o BEGIN TRANSACTION. Para cada transação podem haver até 12 arquivos envolvidos. Para facilitar a realização das operações utiliza-se um *arquivo de controle de transações*. Quando não se usam os BEGIN/END TRANSACTION, então cada operação de acesso ao banco de dados é considerada como uma transação lógica.

Concorrência

O sistema garante o controle de concorrência através de bloqueios nos registros, bloqueios de arquivos nas transações lógicas e concorrência passiva. Entretanto, o manual não faz referências sobre os dispositivos de concorrência específicos para as *árvores-B*, como visto no capítulo 4.

1. Bloqueios de Registros

Os registros bloqueados por uma aplicação não podem ser atualizados, removidos ou bloqueados por outra aplicação, até que sejam liberados; podem apenas ser lidos, desde que a operação de leitura não tenha a opção de bloqueio.

O sistema contém 2 tipos de bloqueios: *bloqueio único* e *múltiplo*, que podem ainda ser de cada um dos tipos *de espera* e *de não-espera*. Esses bloqueios podem ser especificados nas operações GET, STEP, OPEN. As aplicações que fazem bloqueios únicos caracterizam-se por realizar o bloqueio em um único registro por vez; as que

usam bloqueios múltiplos permitem que os programas bloqueiem vários registros consecutivamente. Ao se tentar bloquear um registro, já bloqueado ou pertencente a um arquivo referenciado dentro de uma transação lógica aberta com o bloqueio de espera, o sistema ficará esperando até que o registro seja liberado, para depois retornar o controle à aplicação. No caso do uso do bloqueio de não-espera, BTRIEVE retorna à aplicação, indicando que o registro está bloqueado (ocupado).

Um registro com bloqueio único permanece bloqueado até que a aplicação execute uma operação de desbloqueio (UNLOCK) no arquivo que o contém, ou atualize ou remova este registro, ou faça um bloqueio em outro registro do seu arquivo, ou feche este arquivo, ou referencie-o dentro de uma transação. Por outro lado, os registros com bloqueios múltiplos de um arquivo serão liberados quando a aplicação remover estes registros bloqueados, ou liberá-los em conjunto pela operação de UNLOCK, ou faça referência a esse arquivo dentro de uma transação.

2. Bloqueios nas Transações Lógicas

Sempre que um arquivo for referenciado dentro de um BEGIN /END TRANSACTION, ele fica bloqueado até o fim da transação, podendo apenas ser lido por outros programas em que as operações de leitura estejam fora de alguma transação.

Para evitar a situação de deadlock, todas as transações que fazem uso do mesmo conjunto de arquivos devem ser inicializadas com a opção de não-espera.

3. Concorrência Passiva

Esse controle, sempre ativo, permite um controle de concorrência independente de bloqueios explícitos ou por transações lógicas, podendo ser combinado com esses. Se houver alguma mudança em um registro entre o momento em que ele foi lido até o instante em que será atualizado ou removido, o sistema envia uma mensagem de conflito, indicando uma modificação do registro. Nesse caso, o programa deve realizar uma nova leitura no registro.

Esta técnica caracteriza-se por ser usada quando as aplicações realizam operações que não possuem conexões lógicas, permitindo que os registros sejam lidos, atualizados ou removidos sem o uso de bloqueios ou de transações.

Segurança

A segurança é obtida através da restrição de acessos aos arquivos, realizada pelo método de atribuir senhas aos arquivos através da operação SET OWNER, ou abrir um arquivo em *modo exclusivo*, isto é, nenhuma outra aplicação pode abrir um arquivo que tenha sido aberto desta maneira. Um arquivo com uma senha associada só permite aberturas de arquivos com menção dessa senha. A operação SET OWNER permite a especificação do tipo de acesso que será permitido ao arquivo daí para frente: somente com a especificação

da senha ou leitura livre e com verificação de senhas para outras operações. Em ambos os casos pode-se declarar que o arquivo deve ou não ser criptografado (a senha é usada como chave). A senha é eliminada pela operação CLEAR OWNER.

5.1.4 Comandos e Operações

A seguir daremos os comandos do utilitário BUTIL.EXE para criar, recuperar e gerenciar os arquivos, e as operações para manipular os arquivos usando BTRV. Queremos com isso ilustrar quais são as operações fornecidas por um sistema completo de gerenciamento de arquivos com *árvores-B*.

Comandos do Utilitário BUTIL.EXE

Comandos	Funções
CLONE	Cria um arquivo com especificações iguais a outro já existente
COPY	Copia o conteúdo de um arquivo em outro
CREATE	Cria um arquivo
DROP	Remove um índice suplementar
EXTEND	Espalha as informações de um arquivo em 2 discos
INDEX	Cria um arquivo de índice externo
LOAD	Insere os registros de um arquivo sequencial que não é do BTRIEVE em um arquivo do BTRIEVE
RECOVER	Lê registros de um arquivo do BTRIEVE criando um arquivo sequencial
RESET	Fecha arquivos, libera bloqueios e aborta transações pendentes
SAVE	Salva o conteúdo de um arquivo BTRIEVE em um arquivo sequencial padrão (inverso do comando LOAD)
SINDEX	Cria um índice suplementar para um arquivo existente
STAT	Emite informações sobre um arquivo
STOP	Remove o gerenciador de registro da memória e devolve a parte da memória alocada para o sistema operacional
VER	Emite a versão do gerenciador de registro

Operações para Manipular Arquivos Usando o BTRV

Operações	Descrição
ABORT TRANSACTION	Termina uma transação lógica interrompida, eliminando todas as operações executadas por esta transação
BEGIN TRANSACTION	Define o início de uma transação lógica
END TRANSACTION	Define o fim de uma transação lógica
CLEAR OWNER	Remove uma senha atribuída a um arquivo do BTRIEVE
SET OWNER	Atribui uma senha a um arquivo do BTRIEVE
CLOSE	Fecha um arquivo do BTRIEVE
OPEN	Abre um arquivo do BTRIEVE tornando-o disponível
CREATE	Cria um arquivo no sistema com características específicas
CREATE SUP. INDEX	Cria um índice suplementar
DROP SUP. INDEX	Remove um índice suplementar
DELETE	Remove o registro corrente de um arquivo
INSERT	Insere um novo registro em um arquivo
UPDATE	Atualiza o registro corrente
EXTEND	Divide o conteúdo de um arquivo em 2 discos
GET EQUAL	Recupera o registro com valor do índice igual ao valor procurado
GET GREATER	Idem, maior
GET GREATER OR EQUAL	Idem, menor ou igual
GET LESS THAN	Idem, menor
GET LESS THAN OR	Idem, menor ou igual
GET FIRST	Recupera o registro correspondente ao primeiro valor do índice
GET LAST	Idem, último
GET NEXT	Recupera o registro seguinte ao registro corrente
GET PREVIOUS	Idem, anterior
GET DIRECT	Recupera o registro localizado em um determinado endereço
GET POSITION	Retorna o endereço físico do registro corrente
GET DIRECTORY	Devolve o diretório corrente
SET DIRECTORY	Modifica o diretório corrente
LOCK	Bloqueia um registro
UNLOCK	Libera um ou mais registros
RESET	Libera todos os recursos do BTRIEVE utilizados por uma aplicação
STAT	Devolve as características de um determinado arquivo

Continuação:

STEP FIRST	Recupera o registro localizado na primeira posição física em um arquivo
STEP LAST	Idem, última
STEP NEXT	Recupera o registro localizado no endereço físico seguinte ao registro corrente
STEP PREVIOUS	Idem, anterior
STOP	Finaliza o BTRIEVE removendo-o da memória
VERSION	Devolve a versão atual do sistema

5.2 Sistema VSAM

5.2.1 Introdução

O sistema VSAM (Virtual Storage Access Method) [37, 38, 39, 41] foi desenvolvido pela IBM, para ser usado com sistemas operacionais de memória virtual, como DOS/VSE, OS/VS1 e OS/VS2 MVS. Ele foi criado para substituir o sistema ISAM (Indexed Sequential Access Method) da IBM.

De acordo com Keehn e Lacy [41], antes do surgimento do VSAM os métodos de acesso sequenciais-indexados existentes utilizavam a técnica de encadeamento de páginas para inserir novos dados no arquivo, degradando o desempenho do sistema.

VSAM permite 4 tipos diferentes de organização de arquivos, dependendo do método de armazenamento e acesso aos registros. Estes tipos são: *sequenciado pelo índice* (KSDS), *sequenciado pela entrada* (ESDS), *registros relativos* (RRDS) e *linear*. Todas estas organizações permitem tanto o acesso direto como sequencial. Abordaremos basicamente os aspectos relativos aos arquivos sequenciados pelo índice.

As informações sobre os arquivos de dados e do espaço do disco são armazenadas em estruturas denominadas de *catálogos*. As definições dos arquivos e dos catálogos são realizadas pelo *serviço de método de acesso*, dirigido por comandos. Para processar os arquivos utilizam-se 2 macros: *macro de bloco de controle* e *macro de pedidos* que tem a função de recuperar, inserir, remover e atualizar os registros.

5.2.2 Características Gerais dos Arquivos Sequenciados pelo Índice

De acordo com Comer [19], um arquivo do VSAM sequenciado pelo índice é representado como na figura 5.1. Chamaremos esta estrutura de *árvore-VSAM*. Note-se que essa árvore é semelhante às *árvores-B** (v. 2.4), com a diferença de que os ponteiros que encadeiam os nós folhas foram deslocados para o penúltimo nível, denominado de *conjunto de sequência*. Os nós folhas são chamados de *intervalos de controle*. Um intervalo de controle (*IC*) é uma área contígua de disco dentro de uma única trilha, contendo registros de dados lógicos de tamanho fixo ou variável, informações de controle dos registros e do próprio intervalo de controle e espaço livre para futuras inserções. Ele é a unidade de informação transferida nas operações de entrada/saída entre a memória virtual e a memória secundária, correspondendo a uma página. O tamanho desses intervalos pode variar entre diferentes arquivos do sistema, porém dentro do mesmo arquivo eles têm tamanhos iguais (geralmente um múltiplo de 512 bytes).

Figura 5.1: Estrutura da *árvore-VSAM*.

A figura 5.2 ilustra a estrutura de um *IC*. Na figura $R_1, R_2, \dots, R_m$ representam os registros de dados lógicos armazenados. $ICR_1, ICR_2, \dots, ICR_m$ são informações de controle de cada registro lógico R_j ($1 \leq j \leq m$), como por exemplo o tamanho ou o número de registros adjacentes com o mesmo tamanho. Cada ICR_j ocupa 3 bytes. O campo *ICIC* contém informações de controle do próprio *IC*, incluindo a quantidade de espaço livre do intervalo. Ocupa 4 bytes. Os intervalos de controle referenciados pelo mesmo nó do conjunto de sequência são agrupados em uma área contínua do disco, de tamanho fixo, denominada de *área de controle (AC)*. O tamanho desta área varia de uma trilha a um cilindro do disco.

R_1	R_2	$\dots$	R_n	Espaço livre	ICR_n	$\dots$	ICR_2	ICR_1	$ICIC$
-------	-------	---------	-------	--------------	---------	---------	---------	---------	--------

Figura 5.2: Forma geral de um *IC*.

O conjunto de índices e o conjunto de sequência são formados por registros de índices, cujo formato geral está mostrado na figura 5.3. O campo *cabeçalho* contém 24 bytes de informações de controle sobre o registro. Um item em um registro do conjunto de índice (conjunto de sequência), consiste de um valor do índice e um ponteiro para um registro do próximo nível (para um intervalo de controle). Os valores do índice dentro do campo *entradas do índice* são agrupados em seções e armazenados em formato comprimido. Utilizam-se as técnicas de compressão frontal e posterior, conforme mencionado no capítulo 3 (v. 3.1), eliminando os caracteres não necessários para distingui-los dos valores adjacentes. A figura 5.4 ilustra o formato geral de uma entrada do índice, onde *F* contém o número de caracteres eliminados pela compressão frontal, *L* o número de caracteres que permanecerão no valor do índice e *P* é o ponteiro para o próximo nível, com tamanho de 1 byte.

<i>cabeçalho</i>	<i>lista de IC's livres</i>	<i>espaço não utilizado</i>	<i>entradas de índices</i>
------------------	-----------------------------	-----------------------------	----------------------------

Figura 5.3: Formato geral de um registro de índice.

Valor comprimido	<i>F</i>	<i>L</i>	<i>P</i>
------------------	----------	----------	----------

Figura 5.4: Formato geral de uma entrada do índice.

Quando um registro de dados tem um tamanho tal que não cabe, sozinho, em um *IC*, o sistema utiliza *registros espalhados* ("spanned records"), isto é, o registro é alocado em mais do que um *IC* dentro da mesma *AC*. O *ICIC* desse *IC* recebe uma indicação que o registro é espalhado. O valor do índice de um registro espalhado deve estar obrigatoriamente no primeiro *IC*. Se o último *IC* usado pelo registro espalhado contiver espaço livre, este *IC* não pode ser usado para outros registros. O espaço é reservado para futuras extensões do próprio registro.

No caso de sobrecarga de um *IC* em função da inserção de um registro, o *IC* é desdobrado da seguinte maneira. Considere o intervalo de controle *ic'*, sobrecarregado, pertencente à área de controle *ac'*. Se houver intervalos de controles livres em *ac'*, o procedimento divide as informações de *ic'* com um destes intervalos livres e atualiza o nó do conjunto de sequência para refletir este desdobramento. Se não houver espaço livre em *ac'*,

então realiza-se uma divisão em ac' , alocando-se uma nova área na memória secundária e distribuindo-se os intervalos de controle de ac' nestas duas áreas.

Um arquivo do VSAM sequenciado pelo índice sempre contém um arquivo de índice primário, constituindo do *agrupamento básico* (“basic cluster”). Os valores do índice utilizados nestes arquivos são chamados de *valores primários* ou *valores básicos* e determinam univocamente um registro, isto é, o índice é uma chave.

O sistema possibilita a existência de índices secundários (*índices alternativos* na nomenclatura do VSAM), organizados como uma *árvore-VSAM*, com a diferença de que os intervalos de controle contêm um cabeçalho e registros de tamanho variável. Os valores dos índices alternativos são chamados de *valores alternativos*. Os registros dos intervalos de controle são formados por valores alternativos e por valores primários associados aos registros do índice primário. O cabeçalho possui informações do tipo: tamanho do valor primário, número de valores primários, tamanho do valor alternativo e se este é ou não valor único. O acesso a um registro através de um valor alternativo do índice é feito percorrendo-se o índice alternativo para localizar este valor. Quando encontrado, utiliza(m)-se o(s) valor(es) do índice primário associado(s) com o valor alternativo para determinar o(s) registro(s) desejado(s), percorrendo-se o agrupamento básico para este(s) valor(es).

5.2.3 Análise de Alguns Aspectos

Vamos analisar alguns aspectos que devem ser considerados no sistema.

1. Tamanho do Intervalo de Controle

Grandes intervalos de controle requerem menos operações de entrada/saída, possibilitando mais espaço livre e consequentemente uma menor probabilidade de desdobramentos dos intervalos. Porém, o tamanho dos *IC's* pode afetar o espaço do buffer da memória virtual, além de gastar mais tempo na transferência de e para a memória virtual.

2. Tamanho da Área de Controle

Quando as áreas de controle têm tamanho grande, há uma redução no número de nós no conjunto de índices e uma diminuição na altura da árvore. Entretanto, no caso de desdobramentos em *AC's* extensos, ocorre a movimentação de um maior número de dados.

3. Espaço Livre

A determinação da quantidade de espaço livre é importante, pois uma quantidade

grande resulta no aumento do número de níveis do conjunto de índices (altura da árvore), enquanto que uma quantidade limitada acarreta excessivas divisões dos intervalos e áreas de controle.

Capítulo 6

PROPOSTA DE IMPLEMENTAÇÃO

6.1 Introdução

Com base em todo o estudo realizado, faremos uma proposta de implementação de um “gerenciador de arquivos” que utiliza a estrutura das *árvores-B* para implementação de índices. Levaremos em conta o baixo custo atual das memórias RAM, bem como o fato dos computadores terem grande capacidade de memória. Esse fato não foi considerado pelos autores da área até o momento, a menos da proposta de Souza e Carvalho [89] (v. 4.8). Consideraremos também a alta velocidade de processamento da CPU, tornando praticamente desprezível o seu tempo em relação ao tempo de acesso aos discos.

O gerenciador a ser proposto destina-se a usuários programadores. Alguns dos recursos do gerenciador apresentam diferentes opções, ficando a critério do usuário a escolha da opção que melhor se adapte à aplicação ou situação em que o sistema será utilizado.

Cada seção é composta de várias propostas indicadas pela letra **P**, seguidas de justificativas (letra **J**) quando não as considerarmos óbvias, em que fazemos menção a outras propostas da literatura. Em alguns casos haverá definições de novos termos (letra **D**), importantes para o entendimento do texto.

6.2 Características de Índices

P 6.2.1 Restrições de Integridade.

Um índice pode ser declarado como: a) chave ou não chave; b) de valores modificáveis ou não modificáveis e c) podendo ou não conter um valor vazio. Os padrões são: não chave, modificável e com valor vazio. Um índice declarado como chave é automaticamente considerado como não modificável e sem conter o valor vazio.

J 6.2.1 Um índice não modificável exige para alteração de um de seus valores a eliminação prévia de todos os registros desse valor, seguida da inserção dos mesmos com o novo valor. Essa restrição pode ser muito útil para se distinguir dados permanentes (por exemplo, o nome de um funcionário) de valores voláteis (salário no Brasil). Valores vazios permitem o adiamento das entradas de dados não essenciais a representação de dados ou não aplicáveis (um exemplo destes últimos é o nome do cônjuge de um funcionário legalmente solteiro). Os padrões foram escolhidos levando-se em conta os casos de índices que consideramos mais comuns. Os padrões estabelecidos para as chaves levam em conta que elas servem exclusivamente para identificar univocamente registros (“restrição de identidade”). O ideal de seu projeto é que tenham valores fictícios, de acordo com Setzer [87] e assim sendo, a sua modificação deve ser um caso excepcional. Uma chave não pode assumir o valor vazio, pois este teria que ocorrer apenas em um único registro de dados, perdendo a sua caracterização de vazio (valor desconhecido, inaplicável, etc.).

P 6.2.2 Tipos.

O sistema deve admitir índices dos seguintes tipos: inteiro (2 bytes), inteiro-longo (4 bytes), flutuante (4 bytes), precisão-dupla (flutuante de 8 bytes), data (2 bytes para o ano, 1 para o mês e 1 para o dia), tempo (1 byte para hora, 1 para minuto e 1 para segundo), decimal-compactado (tipo comp-3 do COBOL padrão ANSI-74, com sinal à direita), lógico (1 byte), cadeia de caracteres (normal, padrão C com \0 à direita ou com 1 byte de contagem de bytes à esquerda). Qualquer índice pode ser composto com campos eventualmente não contíguos ou superpostos.

P 6.2.3 Representação do valor vazio.

Um índice do tipo inteiro, inteiro-longo, flutuante e precisão-dupla assume valor interno vazio correspondente ao maior número negativo em valor absoluto. O de tipo cadeia de caracteres é representado com 1 byte onde todos os bits têm valor 0. Os tipos data, hora e decimal-compactado são representados com 1 byte com todos os bits iguais a 1. O campo correspondente no registro pode não conter nenhuma informação, se o registro é de tamanho variável. O usuário pode convencionar valores explícitos externos para representar diversos tipos diferentes de vazio (desconhecido, inaplicável, etc.), sendo tal fato ignorado pelo sistema.

D 6.2.4 Denomina-se de *arquivo-índice* um arquivo de dados composto apenas por valores de um índice, conforme proposto no sistema BTRIEVE [74] (v. 5.1). Neste índice permite-se apenas consultas do tipo “existe ou não um certo valor do índice”.

P 6.2.4 Organização dos arquivos quanto aos índices.

Um índice pode ser declarado com uma das seguintes características: *primário*, *secundário* ou *arquivo-índice*. Os 2 primeiros podem ou não ser chave e o terceiro é necessariamente uma chave.

J 6.2.4 Tendo sido declarado algum índice como primário para um determinado arquivo de dados, os registros e as páginas deste último arquivo deverão ser mantidos na forma mais ordenada possível, segundo os valores crescentes desse índice. Com isso aumenta-se muito a eficiência nas buscas ordenadas por esse índice, como por exemplo busca de todos os valores maiores do que um valor dado, busca de intervalos de valores (v. 2.11), etc. À medida que o arquivo de dados tornar-se desorganizado devido a inserções não ordenadas, o arquivo pode ser reorganizado (v. 6.8). A condição de um índice ser ou não chave é essencialmente uma restrição de integridade, devendo portanto aplicar-se tanto a índices primários como secundários. Um arquivo-índice pode servir como tabela de consistência para verificação eficiente do fato de um valor pertencer ou não a um conjunto de dados.

6.3 Estrutura da Árvore de Índices

P 6.3.1 Implementar a estrutura da *Árvore-B⁺*.

Na escolha da estrutura da árvore de índices vamos considerar 3 alternativas: *árvore-B* (Bayer e McCreight [8], v. capítulo 1), *árvore-B** (Knuth [44], v. 2.4) e *árvore-B⁺* (v. 2.5).

Deixamos de lado a *árvore-2-3* (Hopcroft, v. 2.1) e a *árvore-BB* (Bayer [7], v. 2.2), em virtude destas se aplicarem à memória primária. Por outro lado a *árvore-B com multitabelas* (Lomet [57], v. 2.7) pode requerer a visita de todas as tabelas, dificultando as operações de busca, inserção e remoção. Já a *árvore-B histórica* (Maier e Salveter [62], v. 2.9) contém as mesmas características da *árvore-B*, a menos do número mínimo de valores permitidos em cada nó, para garantir linearidade no número de operações de concatenação e desdobramento. No entanto, o critério de armazenamento na estrutura sendo proposta baseia-se no espaço ocupado e não no número de valores. A *árvore-BD* (Lomet [58], v. 2.10) e a organização *Bounded Disorder* (Litwin e Lomet [56], v. 2.11) foram descartadas, pois necessitam do uso de uma técnica de hashing para garantir a recuperação de uma única página, complicando os algoritmos de busca, inserção e remoção. Não escolhemos a *árvore-Bit* (Ferguson [29], v. 2.12), pois esta também apresenta algoritmos mais

complicados e a compactação nos nós folhas que a mesma proporciona pode ser obtida utilizando-se técnicas de compressão (v. P 6.3.5). A *árvore-B^L* (Lehman e Yao [53], v. 2.8) não foi mencionada, pois contém as mesmas características da *árvore-B**. Utilizaremos ainda as idéias das *árvores-B^O* (Knuth [44], v. 2.3) e das *árvores-B pré-fixadas* (Bayer e Unterauer [12], v. 2.6) conforme será visto em P 6.6.7.3 e P 6.3.5 respectivamente.

Descreveremos inicialmente as vantagens da escolha, representadas por VA, seguidas da desvantagens (DE), com critérios sobre as mesmas e finalmente a justificativa global, comparando as vantagens com as desvantagens. Os índices secundários terão também a forma de uma *árvore-B⁺* e não como VSAM [37, 38, 39, 41] (v. 5.2).

VA 6.3.1.1 Diferentemente das *árvores-B** as *árvores-B⁺* não contém registros de dados nas folhas e sim ponteiros para as páginas destes, possibilitando a construção de um índice sobre qualquer arquivo de dados, desde que seja possível localizar os valores do índice nos registros desse arquivo. Isso representa a não interferência da árvore na representação dos registros de dados. Notemos que as *árvores-B* também possuem esta vantagem.

VA 6.3.1.2 Conforme demonstrado em 2.5, em determinadas situações, as *árvores-B⁺* quando comparadas com as *árvores-B**, podem executar um menor número de acesso para atingir um registro de dados. Como mostrado em 2.4, a altura de uma *árvore-B*, armazenando um número grande de valores de índice, tende a ser bem maior do que as outras duas.

VA 6.3.1.3 Nas *árvores-B⁺* os registros de dados não seguem as regras de formação dos nós internos. Com isso não existe a restrição do fator de ocupação médio de 69,3% das páginas das folhas das *árvores-B**, conforme demonstrado por Küspert [47], Chu e Knott [18] e Baeza-Yates [3], podendo-se tender a 100% nas *árvores-B⁺*. Esta mesma vantagem aplica-se às *árvores-B*.

VA 6.3.1.4 As *árvores-B⁺* possibilitam que as consultas do tipo “existe ou não o valor de um índice primário” sejam realizadas sem acesso aos registros de dados. Essa característica também se aplica para as *árvores-B*.

VA 6.3.1.5 Nas *árvores-B⁺* existe uniformidade de tratamento para acesso aleatório nos índices primários e secundários, o que não ocorre nas *árvores-B**.

VA 6.3.1.6 O uso das *árvores-B⁺* permite a existência de 2 funções diferentes e independentes que são a compactação da árvore de índices (v. 6.9) e a reorganização do arquivo de dados (v. 6.8). Em particular, a compactação é mais rápida do que nas *árvores-B**, uma vez que não é necessário varrer os dados.

DE 6.3.1.1 Na *árvore-B* o valor de um índice pode ser encontrado em algum nó interno, o que produz menos acessos. No entanto, a probabilidade de se encontrar um tal valor nas

folhas é muito maior. Por exemplo, para uma *árvore-B* de ordem $K = 50$ e portanto um grau médio por nó de cerca de 70, com altura $h = 2$, as probabilidades de encontrar um valor no nível 0 é de 0,02%, no nível 1 é de 1,4% e no 2 é de 98,58%. Assim, a diminuição da altura produz um efeito muitíssimo maior do que a possibilidade de encontrar um valor em um nó interno.

DE 6.3.1.2 Em determinadas situações, como demonstrado em 2.5, as *árvores-B⁺* podem ter um acesso a mais do que as *árvores-B**.

DE 6.3.1.3 As *árvores-B** provocam a ordenação dos registros de dados segundo os valores do índice primário. Não consideramos esse fato muito relevante, pois uma entrada de dados ordenada pelos valores do índice primário produz a mesma organização do arquivo de dados. Uma entrada aleatória dá uma certa vantagem à *árvore-B** na busca sequencial por valores desse índice, pois tende a agrupar registros com valores contíguos do índice primário em uma mesma página. Isso pode ser remediado na *árvore-B⁺* por meio de reorganizações periódicas do arquivo de dados, como especificado em P 6.8.2.

J 6.3.1 Consideramos que o grande problema das *árvores-B⁺* consiste na possibilidade de existir um acesso a mais como foi abordado em DE 6.3.1.2. Aparentemente os casos práticos encaixam-se, em boa parte, nos casos em que a *árvore-B⁺* exige o mesmo número de acessos. Para confirmar esse fato, procedemos ao cálculo do número de acessos para alguns casos práticos do banco de dados do sistema de funcionários do CCE (Centro de Computação e Eletrônica) da USP, descritos a seguir. Nas tabelas que mostramos os cálculos e resultados, empregamos os termos abaixo, onde $h \in \{0, 1, \dots\}$.

1. *num-reg-dados*: número de registros de dados;
2. *tam-reg-dados*: tamanho do registro de dados;
3. *tam-nó*: tamanho do nó ;
4. *tam-val*: tamanho do valor do índice;
5. *tam-pont-int*: tamanho do ponteiro para um nó da árvore;
6. *tam-pont-dados*: tamanho do ponteiro para um registro de dados;
7. *tam-entr-h*: tamanho de um item (valor, ponteiro) em um nó do nível h ;
8. *num-val-h*: número total de valores no nível h ;
9. *num-val-nó-h*: número de valores em cada nó do nível h ;
10. *num-nós-h*: número de nós do nível h ;

Vamos calcular as alturas das *árvores-B⁺* e *árvores-B^{*}* para os seguintes casos.

Caso 1:

num-reg-dados = 50.000 registros;

tam-reg-dados = 330 bytes;

tam-nó = 3780 bytes;

tam-val = 6 bytes;

tam-pont-dados = 4 bytes¹;

tam-pont-int = 2 bytes;

Propriedades	<i>árvore-B⁺</i>	<i>árvore-B[*]</i>
<i>tam-entr-h</i>	<i>tam-val</i> + <i>tam-pont-dado</i> = 10	<i>tam-reg</i> = 330
<i>num-val-nó-h</i>	$\frac{\text{tam-no}}{\text{tam-entr-h}} = \frac{3780}{10} = 378$	$\frac{3780}{330} = 11,45 \Rightarrow 11$
<i>num-nó-h</i>	$\frac{\text{num-reg}}{\text{num-val-no-h}} = \frac{50.000}{378} = 132,27 \Rightarrow 133$	$\frac{50.000}{11} = 4545,45 \Rightarrow 4546$
<i>tam-entr-(h-1)</i>	<i>tam-val</i> + <i>tam-pont-int</i> = 8	8
<i>num-val-(h-1)</i>	(<i>num-nó-h</i>) - 1 = 132	4545
<i>num-val-nó-(h-1)</i>	$\frac{3780}{8} = 472,5 \Rightarrow 472$	472
<i>num-nó-(h-1)</i>	$\frac{\text{num-val-(h-1)}}{\text{num-val-no-(h-1)}} = \frac{132}{472} \Rightarrow 1$	$\frac{4545}{472} = 9,63 \Rightarrow 10$
<i>tam-entr-(h-2)</i>	_____	8
<i>num-val-(h-2)</i>	_____	9
<i>num-val-nó-(h-2)</i>	_____	472
<i>num-nó-(h-2)</i>	_____	1

Para este exemplo temos uma *árvore-B⁺* com 2 níveis ($h^{+min} = 1$) e uma *árvore-B^{*}* com 3 níveis ($h^{*min} = 2$).

Usando a idéia demonstrada na seção 2.5 temos que:

$$h_{min}^{*} = \log_{2K^{*}+1} \frac{I}{2K} = \log_{473} \frac{50.000}{11} \cong \log_{473} 4545,5 \cong 1,37 \Rightarrow \lceil \log_{473} \frac{50.000}{11} \rceil = 2$$

$$h_{min}^{+} = \log_{2K^{+}+1} \frac{I}{2K'} = \log_{473} \frac{50.000}{378} \cong \log_{473} 132,3 \cong 0,8 \Rightarrow \lceil \log_{473} \frac{50.000}{378} \rceil = 1$$

$$d_{min} = 1 \Rightarrow (473)^1 < 4545,5 \leq (473)^2 \text{ e } (473)^0 < 132,3 \leq (473)^1$$

$$h_{max}^{*} = \log_{K^{*}+1} \frac{I}{2K} + 1 = \log_{237} 4545,5 + 1 \cong 1,54 + 1 \Rightarrow \lceil \log_{237} \frac{50.000}{11} \rceil = 2$$

$$h_{max}^{+} = \log_{K^{+}+1} \frac{I}{2K'} + 1 = \log_{237} 132,3 + 1 \cong 0,9 + 1 \Rightarrow \lceil \log_{237} \frac{50.000}{378} \rceil = 1$$

$$d_{max} = 1 \Rightarrow (273)^1 < 4545,5 \leq (237)^2 \text{ e } (237)^0 < 132,3 \leq (237)^1$$

¹Na implementação do CCE, *tam-pont-dados* = *tam-pont-int* = 6 bytes, pois a palavra do computador utilizado é de 6 bytes. Fizemos a adaptação para um caso geral, considerando que os ponteiros para os nós internos são relativos a uma base e a árvore ocupa pouco espaço contíguo.

Caso2:

$num-reg-dados = 50.000$ registros;

$tam-reg-dados = 330$ bytes;

$tam-nó = 3780$ bytes;

$tam-val = 76$ bytes;

$tam-pont-dado = 4$ bytes;

$tam-pont-int = 2$ bytes;

Propriedades	$árvore-B^+$	$árvore-B^*$
$tam-entr-h$	$tam-val + tam-pont-dado = 80$	$tam-reg = 330$
$num-val-nó-h$	$\frac{tam-no}{tam-entr-h} = \frac{3780}{80} = 47,25 \Rightarrow 47$	$\frac{3780}{330} = 11,45 \Rightarrow 11$
$num-nós-h$	$\frac{num-reg}{num-val-no-h} = \frac{50.000}{47} = 1063,8 \Rightarrow 1064$	$\frac{50.000}{11} = 4545,45 \Rightarrow 4546$
$tam-entr-(h-1)$	$tam-val + tam-pont-int = 78$	78
$num-val-(h-1)$	$(num-nós-h) - 1 = 1063$	4545
$num-val-nó-(h-1)$	$\frac{3780}{78} = 48,46 \Rightarrow 48$	48
$num-nós-(h-1)$	$\frac{num-val-(h-1)}{num-val-no-(h-1)} = \frac{1063}{48} = 22,14 \Rightarrow 23$	$\frac{4545}{48} = 94,68 \Rightarrow 95$
$tam-entr-(h-2)$	78	78
$num-val-(h-2)$	22	94
$num-val-nó-(h-2)$	48	48
$num-nós-(h-2)$	$\frac{22}{48} \Rightarrow 1$	$\frac{94}{48} \Rightarrow 2$
$tam-entr-(h-3)$	_____	7
$num-val-(h-3)$	_____	1
$num-val-nó-(h-3)$	_____	48
$num-nós-(h-3)$	_____	1

Temos uma $árvore-B^+$ com 3 níveis e uma $árvore-B^*$ com 4 níveis.

De acordo com a demonstração vista em 2.5:

$$h_{min}^* = \log_{49} \frac{50.000}{11} \cong \log_{49} 4545,5 \cong 2,16 \Rightarrow \lceil \log_{49} \frac{50.000}{11} \rceil = 3$$

$$h_{min}^+ = \log_{49} \frac{50.000}{47} \cong \log_{49} 1063,8 \cong 1,78 \Rightarrow \lceil \log_{49} \frac{50.000}{47} \rceil = 2$$

$$d_{min} = 1 \Rightarrow (49)^2 < 4545,5 \leq (49)^3 \text{ e } (49)^1 < 1063,8 \leq (49)^2$$

$$h_{max}^* = \log_{25} 4545,5 + 1 \cong 2,6 + 1 \Rightarrow \lceil \log_{25} \frac{50.000}{11} \rceil = 3$$

$$h_{max}^+ = \log_{25} 1063,8 + 1 \cong 2,15 + 1 \Rightarrow \lceil \log_{25} \frac{50.000}{47} \rceil = 3$$

$$d_{max} = 1 \Rightarrow (25)^2 < 4545,5 \leq (25)^3 \text{ e } (25)^2 < 1063,8 \leq (25)^3$$

Caso3

 $num-reg-dados = 48.000$ registros; $tam-reg-dados = 1920$ bytes; $tam-nó = 3228$ bytes; $tam-val = 20$ bytes; $tam-pont-dado = 4$ bytes; $tam-pont-int = 2$ bytes;

Propriedades	<i>árvore-B⁺</i>	<i>árvore-B[*]</i>
$tam-entr-h$	$tam-val + tam-pont-dado = 24$	$tam-reg = 192$
$num-val-nó-h$	$\frac{tam-no}{tam-entr-h} = \frac{3228}{24} = 134,5 \Rightarrow 134$	$\frac{3228}{192} = 16,8 \Rightarrow 16$
$num-nos-h$	$\frac{num-reg}{num-val-no-h} = \frac{48.000}{134} = 358,2 \Rightarrow 359$	$\frac{48.000}{16} = 3000$
$tam-entr-(h-1)$	$tam-val + tam-pont-int = 22$	22
$num-val-(h-1)$	$(num-nos-h) - 1 = 3583$	2999
$num-val-nó-(h-1)$	$\frac{3228}{22} = 146,72 \Rightarrow 146$	146
$num-nos-(h-1)$	$\frac{num-val-(h-1)}{num-val-no-(h-1)} = \frac{358}{146} = 2,45 \Rightarrow 3$	$\frac{2999}{146} = 20,54 \Rightarrow 21$
$tam-entr-(h-2)$	22	22
$num-val-(h-2)$	2	20
$num-val-nó-(h-2)$	146	146
$num-nos-(h-2)$	1	1

Neste caso temos uma situação em que a altura da *árvore-B⁺* é igual a da *árvore-B^{*}*.

Assim seguindo a idéia apresentada em 2.5:

$$h_{min}^* = \log_{147} \frac{48.000}{16} \cong \log_{147} 3000 \cong 1,6 \Rightarrow \lceil \log_{147} \frac{48.000}{16} \rceil = 2$$

$$h_{min}^+ = \log_{147} \frac{48.000}{134} \cong \log_{147} 358,2 \cong 1,17 \Rightarrow \lceil \log_{147} \frac{48.000}{134} \rceil = 2$$

$$d_{min} = 0 \Rightarrow (147)^1 < 3000 \leq (147)^2 \text{ e } (147)^1 < 358 \leq (147)^2$$

$$h_{max}^* = \log_{74} 3000 + 1 \cong 1,86 + 1 \Rightarrow \lceil \log_{74} \frac{48.000}{16} \rceil = 2$$

$$h_{max}^+ = \log_{74} 358,2 + 1 \cong 1,36 + 1 \Rightarrow \lceil \log_{74} \frac{48.000}{134} \rceil = 2$$

$$d_{max} = 0 \Rightarrow (74)^1 < 3000 \leq (74)^2 \text{ e } (74)^1 < 358,2 \leq (74)^2$$

Descartado o perigo de um acesso extra, às vezes necessário nas *árvores-B⁺*, e levando-se em conta a independência da organização do arquivo, a altura, a uniformidade dos índices primários e secundários, o fator de ocupação das páginas de dados e a rapidez do processo de compactação das árvores de índices, concluímos pela escolha das *árvores-B⁺*.

P 6.3.2 Ponteiros à direita.

Uso de um ponteiro a mais em cada nó folha, apontando para o vizinho imediatamente à direita, formando uma lista encadeada.

J 6.3.2 Aumentar a eficiência de buscas sequenciais ordenadas pelo índice e também para compactação da árvore (v. P 6.9.3).

P 6.3.3 Ponteiros à esquerda.

O usuário pode especificar o uso de um ponteiro extra em cada nó folha, apontando para o vizinho imediatamente à esquerda, formando com o ponteiro da direita uma lista duplamente encadeada.

J 6.3.3 Aumentar a eficiência de buscas em ordem decrescente dos valores do índice quando a frequência dessas consultas forem elevadas.

D 6.3.4 Um nó é considerado *subcarregado* quando sua ocupação é menor do que 50%.

P 6.3.4 Adiamento de Concatenações.

A eliminação de valores de índices pode produzir um nó subcarregado. Nessa situação propomos que não sejam feitas concatenações (v. 1.4.3) imediatas. A técnica a ser seguida, neste caso, será descrita em 6.6.

J 6.3.4 Segundo Lehman e Yao [53], na prática o número de remoções é menor do que de inserções, portanto o número de nós subcarregados deve ser pequeno. Além disso, as futuras inserções podem fazer com que os nós deixem de ser subcarregados. Em certas situações são realizadas concatenações como especificado em 6.6.

P 6.3.5 Compressão de valores de Índices.

Para os índices formados por valores do tipo cadeias de caracteres o gerenciador deve usar as técnicas de compressão frontal e posterior de caracteres. Para o tipo decimal-compactado propomos a eliminação de zeros à esquerda e o uso de compressão frontal. No caso de cadeias de caracteres, sugerimos ainda a eliminação dos caracteres brancos à direita dos valores do índice.

J 6.3.5 As técnicas de compressão de caracteres são as citadas por Wagner [93] (v. 3.1) e empregadas pelo sistema VSAM (v. 5.2) e MUMPS [65, 66] (v. 3.1). A compressão posterior corresponde à *árvore-B pré-fixada* de Bayer e Unterauer [12] (v. 2.6). Elas permitem um maior número de valores armazenados em cada nó e com isso possibilitam uma diminuição na altura da árvore. Notemos que com isso cada nó da árvore passa a ter um grau próprio; o critério de desdobramento é baseado no espaço ocupado e não no número de valores armazenados.

P 6.3.6 Nós sujeitos a compressão.

Nos índices primários e secundários que são chaves a compressão deve ser realizada nos valores localizados em todos os nós da árvore, a menos que o usuário opte pelo não uso da compressão posterior nos nós folhas. No caso de índices não chaves não há compressão posterior nos nós folhas. Nos arquivos-índices o gerenciador deve usar as duas técnicas de compressão nos nós internos e apenas a compressão frontal nas folhas.

J 6.3.6 Devido à compressão posterior nos índice primários e secundários que são chaves só é possível determinar se um valor encontrado corresponde exatamente ao valor sendo procurado através da recuperação de um registro. A opção pelo não uso da compressão posterior nos nós folhas dos índices primários e secundários chaves permite uma maior eficiência em consultas do tipo “existe (ou não) um certo valor do índice”. Este é o único tipo de consulta permitida nos arquivos-índices. No caso de índices não chaves não há compressão posterior nas folhas, pois isso poderia forçar a leitura de vários registros até que fosse encontrado o valor desejado.

P 6.3.7 Estrutura de dados de compressão.

Cada valor do índice comprimido é precedido por dois campos contendo: número de caracteres eliminados pela compressão frontal (NCF) e número de caracteres armazenados com o valor do índice (NCA). Cada um desses campos pode ter 1 ou 2 bytes. Se o primeiro byte tiver o bit mais a esquerda com o valor 0 (1), o campo tem 1 só byte (tem 2 bytes), indicando um comprimento de até 128 bytes (de até 32 Kbytes).

J 6.3.7 Esta técnica foi usada pelo sistema VSAM (v. 5.2) e MUMPS (v. 3.1), a menos da configuração de bits.

P 6.3.8 Compressão opcional.

O usuário pode optar pelo não uso das técnicas de compressão.

J 6.3.8 A existência de poucos caracteres repetidos nos valores de índices pode levar a quantidade total de bytes economizados pelos métodos de compressão a ser menor ou igual ao número total de bytes utilizados pelos campos NCF e NCA, não justificando o emprego das técnicas.

P 6.3.9 Nó filho subcarregado.

Deve-se reservar o primeiro bit de cada ponteiro para nós da árvore (nós descendentes) para determinar se o nó para o qual eles apontam está (valor 1) ou não (valor 0) subcarregado.

J 6.3.9 O uso dessa informação ficará claro quando abordarmos a técnica de concorrência (v. P 6.6.7.1).

6.4 Possíveis Estruturas para Os Nós da Árvore

Quando a velocidade da CPU for muito alta relativamente à de transferência de dados do disco, um valor pode ser buscado sequencialmente dentro de um nó; do contrário, recomendamos uma busca binária para os casos em que não se usam a técnica de compressão frontal, pois esta exige a varredura sequencial dos valores. Neste item apresentamos estruturas para esses dois tipos de busca, ficando a critério do implementador a escolha final.

As propostas seguintes usam campos como delimitadores, tamanho dos valores e ponteiros para os valores, e aplicam-se apenas para os tipos cadeias de caracteres e decimal-compactado. Para os outros tipos de dados em nós internos, propomos as mesmas estruturas sem o uso desses campos, uma vez que possuem tamanho fixo com poucos bytes.

Para as figuras relativas a essa seção consideraremos que cada valor comprimido i_j do índice ($1 \leq j \leq m$) contém os campos NCF e NCA embutidos (v P 6.3.7).

P 6.4.1 Nós internos com busca sequencial.

Os nós internos devem possuir delimitadores para separar os valores, dos ponteiros. Propomos delimitadores com tamanho igual a um byte, onde todos os bits têm valor 1. A figura 6.1 ilustra o formato de um nó com essas características.

Figura 6.1: Configuração de um nó interno com uso de delimitadores.

J 6.4.1 Baseando-se na proposta de Bayer e Unterauer [12] (v. 2.6) usamos delimitadores para identificar o final de um valor do índice (eles não especificam o formato destes delimitadores). Um byte formado por todos os bits 1 é suficiente, pois não existe nenhum caractere ou configuração decimal-compactada com essa representação. Não é possível realizar buscas binárias em nós com essa configuração, pois algum ponteiro p_j ($0 \leq j \leq m$) poderia conter um byte com todos os bits 1.

P 6.4.2 Nós internos com busca binária - I.

Para permitir buscas binárias em nós internos propomos o uso de delimitadores com tamanho igual ao tamanho dos ponteiros, que em geral devem ter tamanho de 2 bytes, com todos os bits iguais a 1.

J 6.4.2 Não há ponteiros para outros nós da árvore com essa configuração. A busca binária é realizada dividindo-se o espaço de buscas em metades sucessivas, procurando-se,

após cada divisão, o delimitador à direita (ou à esquerda).

P 6.4.3 Nós internos com busca binária - II.

Nesta configuração reserva-se uma área do nó contendo ponteiros de dois bytes para o início da posição de cada valor do índice no nó e um byte com o número de valores do nó, no final. A figura 6.2 ilustra o formato de um nó com essa configuração.

Figura 6.2: Configuração de um nó interno com uso de ponteiros para valores do índice.

J 6.4.3 O número de valores do nó serve para determinar o espaço ocupado pela “área de ponteiros”. A busca binária é realizada nessa área comparando-se o valor referenciado por um $p(i_j)$ ($1 \leq j \leq m$) com o valor sendo procurado. Notemos que é possível determinar o final de um valor i_j através do conteúdo de $p(i_{j+1})$, pois os ponteiros p_j 's têm tamanho fixo. Observemos que essa configuração requer o mesmo número de bytes extras que a apresentada em P 6.4.2, devendo ser mais rápida do que aquela.

P 6.4.4 Nós folhas de índices não chaves com busca sequencial.

P 6.4.4.1 Valores comprimidos ou com tamanho fixo.

Uma configuração possível para este caso está mostrada na figura 6.3. Para cada valor i_j do índice ($1 \leq j \leq m$) existe um campo cp_j de tamanho de 1 byte, que armazena a quantidade de ponteiros associados com i_j . No caso de existir mais do que 256 ponteiros associados com i_j , repete-se esse valor quantas vezes for necessário, colocando-se em seguida de cada um os próximos ponteiros.

Figura 6.3: Configuração de um nó folha de um índice não chave com valores comprimidos ou com tamanho fixo.

J 6.4.4.1 O campo cp_j serve para indicar o início do valor i_{j+1} , pois os ponteiros para os registros de dados têm tamanho fixo. Notemos que para os valores com compressão

frontal não há necessidade de se guardar o tamanho de i_j em um campo especial, uma vez que este tamanho está contido no campo NCA. Por outro lado, não há compressão posterior nas folhas.

P 6.4.4.2 Valores não comprimidos ou com tamanho variável.

Neste caso usa-se um campo T_j ($1 \leq j \leq m$), ocupando 1 ou 2 bytes, para armazenar o tamanho do valor i_j . O primeiro bit de T_j é reservado para indicar se há (valor 1) ou não (valor 0) mais do que um ponteiro para os registros de dados. Em caso de multiplicidade de ponteiros usa-se um campo cp_j , logo após i_j , contendo a quantidade destes ponteiros. A figura 6.4 ilustra essa configuração para os casos em que (a) há mais de um ponteiro e (b) há um só ponteiro.

Figura 6.4: Configurações de um nó folha de um índice não chave com valores não comprimidos ou com tamanho variável.

J 6.4.4.2 O uso do campo T_j ($1 \leq j \leq m$) serve para indicar o fim do valor i_j , uma vez que este não é comprimido (não contém o campo NCA) e pode ter tamanho variável. O contador de ponteiros cp_j é usado para determinar o início de T_{j+1} . Esse contador não é necessário quando existir apenas um único ponteiro associado.

P 6.4.5 Nós folhas índices não chaves com busca binária.

Propomos uma configuração semelhante à apresentada em P 6.4.3, acrescentando-se a cada campo p_j ($1 \leq j \leq m$) 1 byte contendo o número de ponteiros associados a i_j (cp_j).

J 6.4.5 O uso do byte adicional permite, juntamente com $p(i_{j+1})$, o cálculo do tamanho de i_j .

P 6.4.6 Nós folhas de índices chaves.

Sugerimos o uso de uma das configurações apresentadas em P 6.4.1, P 6.4.2 ou P 6.4.3, pois nesse caso também haverá um único ponteiro para cada valor do índice.

P 6.4.7 Nós folhas de arquivos-índices.

Deve-se usar delimitadores entre cada valor, com tamanho de 1 byte, composto por bits 1, tanto na busca binária quanto na busca sequencial.

6.5 Alocação Física

P 6.5.1 Categorias de arquivos de dados.

O sistema prevê a existência de 2 categorias de arquivos de dados: *arquivos externos*, criados por outros sistemas e *arquivos internos*, construídos pelo gerenciador. Estes últimos podem ser de *tamanho limitado* ou *expansíveis*. Um arquivo externo pode ser transformado em um arquivo interno através de uma *operação de carga inicial*, que permite a carga em bloco (isto é, com a execução de um único comando) produzindo, de uma vez, a construção de todos os arquivos de índice.

J 6.5.1 Desejamos dar ao gerenciador a capacidade de construir árvores de índices para qualquer arquivo externo. Pode-se usar uma rotina de conversão de externo para interno, possibilitando maior eficiência no uso dos índices. Para os arquivos internos de tamanho limitado o usuário deve prever o espaço máximo a ser ocupado; essa categoria permite maior eficiência nas buscas pelo índice.

P 6.5.2 Rotina de cálculo de endereçamento.

Quando os valores de um índice estão posicionados em campo fixo do registro de dados, o gerenciador deve ser capaz de identificar a posição desses valores, a partir dos parâmetros fornecidos pelo usuário para cada índice. Quando os valores de um índice não ocupam um campo fixo no registro de dados é necessário usar uma rotina de cálculo do endereço da posição do valor do índice, bem como o comprimento deste, para cada registro de dados. Esta rotina é fornecida pelo usuário para cada índice e para cada arquivo.

J 6.5.2 Com isso permite-se que o arquivo de dados tenha qualquer estrutura em seus registros. Por exemplo, a compressão de dados produz campos de tamanho variável. Uma proposta para registro de tamanho variável com campos compostos e múltiplos (multi-valorados) pode ser encontrada em Setzer [87]. Nesse caso, a localização de cada campo é feita pela ordem do mesmo dentro do registro, formando uma lista encadeada. A rotina fornecida pelo usuário percorre a lista até chegar ao campo de número de ordem correspondente, concluindo seu endereço e comprimento, os quais são passados para o gerenciador.

P 6.5.3 Reserva de espaço para arquivos de tamanho limitado.

Quando o sistema operacional permitir, o gerenciador deve reservar uma área contígua do disco para armazenar o arquivo de dados e todos os arquivos de índices associados ao mesmo, cujo tamanho é deduzido a partir de informações fornecidas pelo usuário (v. P 6.10.7). Neste caso, deve-se usar nas folhas das *árvores-B⁺*, um número de página relativo à página 0 inicial do arquivo para referenciar as páginas do arquivo de dados, formando um ponteiro relativo.

J 6.5.3 A vizinhança do arquivo de dados com as *árvores- B^+* diminui a movimentação do braço do disco (“seek-time”). cremos que para microcomputadores ou super-micros (estações de trabalho), um número de páginas ocupando 2 bytes é suficiente. Neste caso, em um exemplo com páginas com tamanho de 2 Kbytes e registros de dados de 500 bytes, ter-se-ia cerca de 250.000 registros.

P 6.5.4 Sobrecarga do espaço reservado.

Quando o espaço reservado para os arquivos de tamanho limitado atingir, por exemplo, 90% de sua ocupação (parâmetro a ser fornecido pelo usuário), o gerenciador insere uma mensagem no “quadro de avisos” (v. 6.10) para informar ao usuário. O usuário deve observar o quadro de avisos e executar as rotinas de reorganização (v. 6.8), compactação (v. 6.9) ou utilizar uma rotina de cópia das informações para uma nova área maior do que a primeira, com eventual compactação e reorganização. Caso o espaço torne-se 100% ocupado as funções de inserção de registros retornam código de erro apropriado.

J 6.5.4 A alocação de novas páginas do disco não se adapta eventualmente ao endereçamento relativo.

P 6.5.5 Referências a arquivos externos e expansíveis.

Os ponteiros das *árvores- B^+* para os arquivos externos e expansíveis devem ser capazes de referenciar todo o disco, não sendo utilizados ponteiros relativos (v. P. 6.5.3) para estes.

J 6.5.5 Não se reservam áreas do disco para esses arquivos, sendo alocadas páginas quando necessário.

P 6.5.6 Árvore de espaço livre.

Para cada arquivo de dados existirá uma *árvore de espaço livre*, organizada como uma *árvore- B^+* , contendo informações sobre os espaços não ocupados dentro de cada página do arquivo de dados. Cada item de um nó folha da árvore de espaço livre é constituído pelo par (endereço da página, espaço). Cada item de um nó não folha é formado pelo endereço de uma página e um ponteiro para um nó descendente. Os nós folhas formam uma lista encadeada com ponteiros para os vizinhos à direita. O usuário deve fornecer como parâmetro o tamanho mínimo para os registros de dados (v. P 6.10.7), afim de evitar que exista na árvore uma entrada para uma página que não contenha espaço livre suficiente para armazenar um registro de dados.

J 6.5.6 Garantir uma melhor ocupação do arquivo de dados, tentando aproveitar áreas livres dentro de páginas não totalmente ocupadas. Notemos que os nós não folhas servem apenas para guiar a busca. O uso da mesma estrutura (*árvore- B^+*) possibilita o reaproveitamento das operações com índices. O funcionamento desta estrutura ficará claro em P 6.7.1 e P 6.7.2.

P 6.5.7 Arquivos residentes.

O sistema deve permitir que o usuário solicite que arquivos de índices e/ou de dados e/ou árvores de espaço livre fiquem residentes na memória primária, sem ocasionar alterações nas funções utilizadas.

J 6.5.7 Permitir maior eficiência de acesso sem alterações dos programas de aplicação.

P 6.5.8 Raiz residente.

As raízes das *árvores-B⁺* não residentes devem passar a residir na memória primária, após seu primeiro uso, desde que o sistema operacional o permita.

J 6.5.8 Como por hipótese inicial a memória principal é grande, o armazenamento dessas raízes diminui um acesso no percurso das árvores. O primeiro acesso indica que o índice é usado pela aplicação.

6.6 Concorrência

Apresentamos um novo método para permitir o uso concorrente de *árvores-B* e suas variações, também válido para as *árvores-B⁺* sugeridas.

P 6.6.1 Princípios do método.

O método satisfaz 3 aspectos simples para as operações de busca, inserção, remoção e compactação concorrentes da árvore: cada processo em execução bloqueia um número pequeno de nós em cada instante; manutenção de um elevado fator de ocupação na árvore e consequente garantia de árvores com alturas baixas.

J 6.6.1 Permitir que um grande número de processos utilizem simultaneamente os índices e os arquivos de dados com pouca interferência entre eles, devido aos bloqueios.

D 6.6.2 Denominamos de *vetor de fatores de ocupação* uma área da memória principal reservada para conter o fator de ocupação de cada nível de uma árvore. Este fator é calculado empregando-se somente os nós do nível.

P 6.6.2 Fator de ocupação por nível.

Para cada arquivo de índice aberto existe um vetor de fatores de ocupação.

D 6.2.3 Denominamos de *pilha do caminho hierárquico* uma área da memória principal reservada para armazenar os endereços dos nós de um caminho de acesso de um processo em execução.

P 6.3.2 Caminho de acesso.

Para cada processo em execução o gerenciador guarda uma pilha do caminho hierárquico.

P 6.6.4 Tipos de bloqueios.

Utilizam-se 2 tipos de bloqueios: bloqueio-*f* e bloqueio-*c*. O primeiro consiste dos bloqueios realizados nas operações de busca, inserção e remoção. O segundo é utilizado quando o gerenciador faz uma compactação total da árvore. Os nós bloqueados por um determinado processo podem ser lidos por outros, mas nunca atualizados.

J 6.6.4 A redução no número de tipos de bloqueios serve para simplificar o processo. Conforme vimos no capítulo 4, existem soluções que utilizam vários tipos de bloqueios como as de Bayer e Schkolnick [11] (v. 4.3), Miller e Snyder [67] (v. 4.4), Kwong e Wood [50] (v. 4.7) e Souza e Carvalho [89] (v. 4.8). O bloqueio-*c* serve para permitir que seja realizada a operação de compactação da árvore em concorrência com as demais operações. Ele é diferente do bloqueio-*f*, pois possibilita que os nós à direita de um nó com bloqueio-*c* sejam utilizados por outros processos (v. P 6.9.3).

D 6.6.5 Um nó é considerado *completo* (comparar com a definição correspondente na seção 1.2) quando não possui espaço livre ou não contém espaço suficiente para armazenar um novo valor do índice, isto é, espaço igual ao tamanho máximo estipulado para os valores do índice.

P 6.6.5 Pré-modificações.

À medida que um processo percorre a *árvore-B⁺* de cima para baixo, em qualquer operação, e encontra um nó completo, o gerenciador tenta fazer uma redistribuição, através do uso da técnica de overflow (v. 1.4.2) ou, se isso não for possível, um desdobramento no nó (v. 1.4.2), afim de deixar um espaço livre suficiente para armazenar um novo valor do índice. Por outro lado, se forem encontrados nós subcarregados (v. 6.3.4), tenta-se executar concatenações (v. 1.4.3) em nós vizinhos (*compactação local*) a ser explicada em P 6.6.7.1.

J 6.6.5 O método que sugerimos baseia-se na idéia de pré-desdobramentos de nós de Guibas e Sedgewick [32] e nas soluções de concorrência apresentadas por Mond e Raz [70] (v. 4.8). Diferentemente desses métodos o nosso executa, além dos pré-desdobramentos, pré-redistribuições e pré-concatenações. Sagiv [84] (v. 4.6) também propôs reorganizações em nós subcarregados executadas concorrentemente com as operações de busca, inserção e remoção. No entanto, elas não são executadas quando a árvore é percorrida de cima para baixo como no nosso caso. Os 2 primeiros evitam que modificações posteriores em um nó, ou eventualmente em seu irmão, produzam alterações nos ascendentes dos mesmos. Com isso diminui-se o número de bloqueios executados por cada processo. As pré-concatenações também permitem um aumento do fator de ocupação e diminuição da altura da árvore.

P 6.6.6 Mecanismo de bloqueio.

Durante a busca de um processo leitor-puro ou da fase de busca de um processo atualizador (fase 1 definida em 4.1) bloqueiam-se, a partir da raiz, o nó corrente n e seu respectivo pai. Antes de se bloquear o filho apropriado de n libera-se o bloqueio do nó pai de n .

J 6.6.6 Evitar um número excessivo de nós bloqueados por cada processo (no máximo 2 são bloqueados por cada processo em execução em cada instante) e uma situação de deadlock. Essa técnica baseia-se na apresentada por Mond e Raz.

P 6.6.7 O Método.

Ao encontrar um nó n completo o gerenciador executa concatenações em nós filhos de n adjacentes subcarregados, ou uma redistribuição entre n e alguns dos seus irmãos, ou um desdobramento de n , afim de permitir que n contenha espaço livre para futuras inserções e possa provocar desbloqueio de seu pai. A escolha de um desses 2 últimos depende do fator de ocupação do nível do pai de n , guardado no vetor de fatores de ocupação. Para todo o método, vamos considerar P_1 o processo em execução e n o nó corrente de P_1 .

P 6.6.7.1 Concatenação.

O procedimento consiste em examinar n verificando se existem 2 filhos subcarregados de n , a uma “distância” de no máximo, por exemplo, 3 ponteiros adjacentes de n . Essa informação é facilmente obtida observando-se o primeiro bit de cada ponteiro de n que aponta para um filho (v. P 6.3.9). No caso em que existem nós filhos subcarregados e estes não estão bloqueados (por outro processo), concatenam-se esses filhos através de redistribuições, eliminando algum separador de n e desligando-se o primeiro bit do ponteiro deste filho. Entretanto, se os nós filhos subcarregados estiverem bloqueados por outro processo P_2 , P_1 fica em estado de espera até que esses bloqueios sejam liberados. Denominaremos essa técnica de *compactação local*. Estas compactações locais devem ser realizadas quando for possível obter, em cada nó, espaço suficiente para inserção de um novo valor. Se após esse procedimento n continuar completo, então, dependendo do caso, sugerimos a aplicação de P 6.6.7.2 ou de P 6.6.7.3.

J 6.6.7.1 A concatenação de filhos de n elimina um separador em n , aproveitando para realizar uma compactação local na árvore e com isso melhorar o fator de ocupação. O uso do bit que indica subcarga evita a leitura do nó apontado pelo respectivo ponteiro. Escolhemos uma distância máxima de 3 ponteiros pois acreditamos que uma distância maior acarretaria grandes gastos com redistribuições. Notemos que não há necessidade de se bloquear os filhos de n antes de se fazer as concatenações, pois neste instante nenhum outro processo que ainda não passou por n poderá bloqueá-los, uma vez que n encontra-se bloqueado. Com isso o processo bloqueia no máximo 2 nós a cada instante (n e seu pai). Se existir um processo P_2 bloqueando o nó n' , onde n' é um filho de n , então n não é o nó corrente em P_2 , pois caso contrário n estaria bloqueado por P_2 . Seja n'' , filho de n' , o nó corrente em P_2 . Afirmamos que as alterações executadas por P_2 em n'' e conseqüentemente em n' , não se propagam para n , pois por construção n' contém espaço livre para futuras

inserções (P_2 já visitou n'). A figura 6.5 ilustra um trecho de uma *árvore-B⁺* com essa situação, onde $f(P_j)$ indica um bloqueio- f executado pelo processo P_j .

Figura 6.5: Exemplo dos processos P_1 e P_2 em execução do método.

P 6.6.7.2 Redistribuição.

Quando o nó n está completo e o fator de ocupação do nível de seu pai for maior do que 85%², por exemplo, o sistema tenta reservar espaço livre em n através da técnica de overflow da seguinte forma. Se n tem pelo menos 2 irmãos à esquerda e/ou à direita, o procedimento tentará fazer redistribuição entre os elementos de n e irmãos, olhando primeiro os irmãos da esquerda e depois os da direita; ou seja, a redistribuição é executada examinando-se no máximo 4 irmãos de n . Se algum desses irmãos estiver bloqueado por um processo P_2 , por exemplo, P_1 fica esperando até que seja liberado. No caso em que n não possui 2 irmãos para algum dos lados, então examina-se para este lado a quantidade existente. Quando não for possível empregar a técnica de overflow para algum desses irmãos de n , faz-se um desdobramento em n , como explicado em P 6.6.7.3.

J 6.6.7.2 Não há necessidade de P_1 bloquear os irmãos de n , pois o pai de n encontra-se bloqueado, impedindo o alcance desses nós por qualquer outro processo que ainda não passou pelo pai de n . Dessa forma garantimos, a cada instante, o bloqueio de 2 nós por cada processo em execução. Se um dos irmãos de n , por exemplo n_1 , encontra-se bloqueado por um processo P_2 , no momento em que P_1 deseja executar a redistribuição, garantimos que o nó corrente em P_2 é um filho n'_1 de n_1 , pois caso contrário, o pai de n não estaria bloqueado por P_1 . A figura 6.6 ilustra essa situação. Nesse caso, P_1 deve esperar até que P_2 libere o bloqueio de n_1 . Notemos que não há problema de deadlock, uma vez que n_1 contém espaço livre para inserção de um novo elemento (P_2 já visitou n_1). Reduzimos a tentativa de redistribuição apenas para os nós irmãos de n e não para os vizinhos, isto é, para os “primos” de n , afim de evitar problemas de deadlock. Cremos

²O limite de 85% para se executar um desdobramento ou redistribuição foi escolhido intuitivamente, uma vez que não foi realizado qualquer tipo de simulação.

que a tentativa de redistribuição dos elementos entre 4 nós é suficiente, pois um número maior acarretaria muitos gastos com o tempo de verificação.

Figura 6.6: Exemplo dos processos P_1 e P_2 em execução do método.

P 6.6.7.3 Desdobramento.

Quando n está completo e o fator de ocupação do nível do pai de n for menor ou igual a por exemplo 85%, ou for maior do que 85%, mas não foi possível realizar redistribuição entre seus irmãos, ou a aplicação de concatenação não reservou um espaço suficiente em n , então n é desdobrado. Faz-se um desdobramento 2 para 3 (v. 2.3), isto é, unem-se os elementos de n e de um de seus irmãos adjacentes n_1 , dividindo-os em 3 nós. Se este irmão encontra-se bloqueado por outro processo P_2 (como na figura 6.6), então P_1 aguarda até que n_1 seja liberado. Após o desdobramento e a atualização do nó pai de n , executa-se uma das seguintes ações, dependendo da localização do ponteiro p_j para o próximo nó filho a ser visitado: a) se p_j está no nó que continha originalmente n , libera-se o bloqueio de seu pai e realiza-se um bloqueio no nó apontado por p_j ; b) se p_j está no novo nó que contém parte dos elementos de n , libera-se o bloqueio de n , bloqueia-se este novo nó e libera-se o bloqueio do pai de n nesta ordem. Em seguida bloqueia-se o nó referenciado por p_j , dando continuidade ao procedimento. Quando n for um nó folha deve-se atualizar os ponteiros para os vizinhos à direita.

J 6.6.7.3 Com a técnica de desdobramento de 2 para 3 nós o fator de ocupação em cada nó é de pelo menos 67% e em média de 81% (v. 2.3). As justificativas dadas em J 6.6.7.2 para o problema de deadlock, para o não bloqueio dos irmãos de n , para a espera do processo P_1 quando um dos irmãos estiverem bloqueados e para o número máximo de 2 nós bloqueados em cada instante por um processo em execução aplicam-se também nesse caso.

P 6.6.8 Deadlock.

O método apresentado está livre de daeadlock.

J 6.6.8 Além das explicações dadas em J 6.6.7.1, J 6.6.7.2 e J 6.6.7.3 o método apresentado está livre de deadlock, pois os bloqueios são sempre executados seguindo-se uma ordem correspondente à estrutura da árvore, isto é, de cima para baixo.

6.7 Operações

Nesta seção abordaremos com certo detalhe algumas operações especiais. As demais operações são aquelas típicas para gerenciadores de arquivos com *árvores-B*, ou com alguma variação, como propostas por exemplo pelo sistema BTRIEVE (v. 5.1).

P 6.7.1 Inserção.

Devemos distinguir entre as operações de inserção e de carga inicial. A primeira consiste na inclusão de um único registro de dados. A segunda diz respeito à carga em bloco como visto em P 6.5.1. Para a operação de inserção propomos o uso da árvore de espaço livre (v. P 6.5.6), tanto quando existir ou não arquivo de índice primário.

P 6.7.1.1 Sem índice primário.

Percorre-se a árvore de espaço livre para atingir o seu nó folha mais à esquerda. A partir desse ponto visitam-se os nós folhas, da esquerda para direita, pelos ponteiros para os vizinhos (lista encadeada), tentando localizar o endereço de uma página que contém uma quantidade de espaço livre suficiente para armazenar o novo registro de dados. Se este endereço existir, faz-se a inserção do registro nessa página do arquivo de dados e atualizam-se os arquivos de índice e a árvore de espaço livre. Esta última atualização é executada subtraindo do item contendo o endereço da página o espaço utilizado pelo novo registro, ou retirando este item da árvore, quando o espaço tornar-se menor do que o tamanho mínimo do registro estipulado pelo usuário (v. P 6.10.3). No caso de não existir nenhum endereço livre para se efetuar o armazenamento, usa-se uma nova página do disco.

P 6.7.1.2 Com índice primário.

A inserção de um registro de dados em um arquivo que contém um índice primário é realizada percorrendo-se, inicialmente, a árvore do índice primário, para localizar a posição de inserção j do valor do índice do novo registro, em um nó n ($1 \leq j \leq m$). Ao se determinar a posição j , examinam-se os endereços das páginas que contém os registros com os valores vizinhos à esquerda e à direita de i_j em n , afim de verificar a existência de espaço livre nestas páginas. Isso é realizado buscando-se na árvore de espaço livre os endereços desses vizinhos na seguinte ordem: endereço de $i_{j-1}, i_{j+1}, i_{j-2}, i_{j+2}, \dots, i_1, i_m$,

até que se encontre uma página com espaço suficiente, ou se atinja o endereço da página que contém i_m e esta não possuir espaço. Nesse caso, procura-se o endereço de uma possível página como em P 6.7.1.1. Ao se determinar a página, caso exista, onde o novo registro será incluído, executa-se a inserção, atualizam-se os arquivos de índice e a árvore de espaço livre. Quando não for encontrado nenhum endereço de página na árvore, utiliza-se uma nova página do disco.

J 6.7.1 O uso da árvore de espaço livre tenta manter um elevado fator de ocupação do arquivo de dados. No caso do arquivo com índice primário verificam-se os endereços das páginas dos vizinhos afim de preservar a contiguidade física. No entanto, para os arquivos sem índice primário não há necessidade de se examinar os vizinhos.

P 6.7.2 Remoção.

Na remoção de um registro de dados localizado em uma página $\mathcal{P}$ do arquivo, atualizam-se os arquivos de índices e a árvore de espaço livre associados. Quando existir um endereço para $\mathcal{P}$ na árvore de espaço livre, soma-se ao campo espaço o tamanho da área ocupada pelo registro. No caso de não existir entrada para $\mathcal{P}$, insere-se uma nova.

P 6.7.3 Desdobramento Marginal.

Quando forem inseridos registros de dados com valores ordenados do índice primário, o gerenciador deve identificar este fato. Neste caso é utilizada a técnica de desdobramento marginal (v. 1.4.2) para incluir aqueles valores na árvore do índice primário. No entanto, cada nó deve conservar um fator de ocupação de desdobramento marginal dado pelo usuário (v. P 6.10.5).

J 6.7.3 A técnica de desdobramento marginal introduzida por Küspert [47] serve para aumentar o fator de ocupação da árvore e diminuir o número de desdobramentos. Notemos que com um fator de ocupação de desdobramento marginal dado pelo usuário, maior ou igual a 20%, não se justifica o uso dessa técnica, pois no caso médio temos uma ocupação de 81% (v. 2.3). A existência do uso de um fator de ocupação de desdobramento marginal em lugar de um valor proposto por Küspert, deve-se ao fato de estarmos trabalhando com valores de tamanho variável. Para detectar a ordenação dos valores sendo inseridos, o gerenciador compara-os sucessivamente. Quando a ordem for perturbada encerra-se o uso da técnica de desdobramento marginal até que seja detectada uma nova sequência ordenada. Essas sequências devem ter no mínimo 3 elementos, isto é, esse tipo de desdobramento é apenas ativado durante a inserção do terceiro elemento.

6.8 Reorganização do Arquivo de Dados

D 6.8.1 Dado um arquivo de dados $\mathcal{D}$, com um índice primário, considere $p_1, p_2, \dots, p_n$ a

sequência de ponteiros das folhas da *árvore-B⁺* desse índice. Dizemos que dois ponteiros adjacentes p_j e p_{j+1} ($1 \leq j \leq n$) estão *ordenados*, se p_j e p_{j+1} apontam para uma mesma página ou para páginas fisicamente consecutivas do arquivo de dados. O *fator de ordenação* de $\mathcal{D}$ é dado pela razão entre o número de pares de ponteiros adjacentes ordenados e o número total de ponteiros menos 1 ($n - 1$). O *fator de ocupação de dados* de $\mathcal{D}$ é dado pela razão entre o espaço ocupado pelos registros de dados e o espaço total de $\mathcal{D}$.

P 6.8.1 Motivos para reorganizar.

Existem 2 motivos para se fazer uma reorganização em um arquivo de dados $\mathcal{D}$: quando $\mathcal{D}$ possui um baixo fator de ordenação e/ou quando $\mathcal{D}$ possui um baixo fator de ocupação.

J 6.8.1 A existência de um índice primário implica em certa frequência de busca ordenada pelo mesmo. A desorganização do arquivo de dados segundo esse índice certamente produz ineficiência. Por outro lado, um baixo fator de ocupação em $\mathcal{D}$ implica em desperdício de espaço do disco, podendo gerar uma sobrecarga do espaço reservado para arquivos de tamanho limitado (v. P 6.5.3 e P 6.5.4).

P 6.8.2 Tipos de reorganização.

O gerenciador deve efetuar uma reorganização do arquivo de dados de forma *manual* ou *automática*, ordenando-o segundo os valores do índice primário. A reorganização manual é requisitada pelo usuário, quando este, consultando o “quadro de avisos” (v. 6.11) julgar necessário fazê-la, podendo ser *imediata* ou *programada* para certa data e hora. A automática pode ser *condicionada* ou *periódica*. A condicionada é executada quando o fator de ocupação e/ou de ordenação do arquivo atingir valor inferior a algum parâmetro dado pelo usuário (v. P 6.10.1), e em períodos pré-fixados, como por exemplo todos os domingos às 2:30 horas da manhã; todo dia primeiro de cada mês às 4:00 horas. A periódica é efetuada também em “datas regulares”, mas independentemente do fator de ordenação ou de ocupação. As reorganizações são executadas de maneira que cada página de dados tenha um fator de ocupação inicial igual a um parâmetro fornecido pelo usuário (v. P 6.10.2).

J 6.8.2 Permitir que o arquivo de dados seja reorganizado quando o usuário achar necessário ou nos momentos em que, em geral, há baixa utilização do sistema.

6.9 Compactação Total

Como visto em 3.2, a compactação reorganiza uma árvore eliminando alguns espaços não utilizados. Em P 6.6.7.1 vimos um caso de compactação local. Nesta seção trataremos da compactação total, isto é, englobando toda a árvore.

P 6.9.1 Formas de compactação.

De forma análoga à reorganização de um arquivo de dados (v. P 6.8.2) deve haver compactações manuais, subdivididas em imediatas ou programadas e automáticas, subdivididas em periódicas e condicionadas. Esta última é ativada quando o fator de ocupação da árvore (v. 1.6) for menor do que um parâmetro fornecido pelo usuário (v. P 6.10.3).

J 6.9.1 A existência de *árvores- B^+* com baixo fator de ocupação acarreta um número desnecessário de nós e conseqüentemente um aumento na altura da árvore.

P 6.9.2 Método - Parte I.

O procedimento de compactação constrói a *árvore- B^+* compactada em uma nova área do disco, isto é, a árvore não é montada em cima da original. Após a construção, a nova árvore é copiada para a área que contém a antiga. No final, libera-se o espaço onde a árvore foi construída.

J 6.9.2 Baseando-se na idéia do algoritmo de compactação de Miller et al. [68], optamos pela construção em uma nova área para permitir que a compactação seja executada de forma concorrente com as operações de busca, inserção e remoção, na árvore original, além de possibilitar que os nós de um mesmo nível fiquem fisicamente contíguos.

P 6.9.3 Método - Parte II.

O procedimento consiste em se visitar o nó folha n_1 mais à esquerda, fazendo um bloqueio-*c* (v. P 6.6.4) neste nó. Em seguida percorre-se n_1 colocando-se seus elementos em uma nova página P_1 do disco. Através do ponteiro para a direita (v. P 6.3.2) atinge-se o vizinho imediato n_2 de n_1 , executa-se um bloqueio-*c* em n_2 e insere-se elementos de n_2 em P_1 , quando couber, ou em uma nova página P_2 , fazendo P_1 apontar para P_2 . Esse processo continua até que seja atingido o último nó folha, produzindo a última página P_m . As páginas $P_1, \dots, P_m$ passam a ser os nós folhas da nova árvore sendo construída. A partir desse momento o procedimento constrói os nós do nível dos pais de $P_1, \dots, P_m$, percorrendo cada uma dessas páginas e colocando o último elemento de P_j como separador de P_j e P_{j+1} ($1 \leq j \leq m-1$). O processo continua da mesma forma para os demais níveis até que se construa a raiz da nova árvore. Após a montagem dessa raiz a nova árvore é copiada para a área da antiga e todos os processos passam a utilizá-la. O fator de ocupação inicial de cada página construída pela compactação é estipulado pelo usuário (v. P 6.10.4). Observemos que os nós folhas ficam fisicamente à esquerda dos seus pais, e assim sucessivamente. O uso do bloqueio-*c* permite que outros processos leiam os nós bloqueados e possibilita que processos de atualização insiram e removam elementos nos nós à direita dos nós com bloqueio-*c*. Notemos que durante a construção dos nós internos da nova árvore só são executados procedimentos de leitura na antiga, pois todos os nós folhas contêm bloqueios-*c*.

J 6.9.3 Garantir a contiguidade de nós de um mesmo nível na árvore (busca em largura).

Ao se percorrer uma árvore, estando em um nó de um nível l , o próximo nó a ser visitado estará no nível $l + 1$. Se todos os nós do nível $l + 1$ estiverem fisicamente contíguos aos nós do nível l , o tempo de movimentação do braço do disco deverá ser menor.

P 6.9.4 Método - Parte III.

Associada a cada árvore sendo compactada deve-se usar um *indicador de compactação* para informar se a árvore está (valor 1) ou não (valor 0) em processo de compactação. Quando um processo de atualização detectar um nó folha contendo um bloqueio- c , ele deve ser interrompido e ficar examinando o indicador de compactação. No momento em que o valor deste indicador for 0, o processo deve recomeçar na raiz da nova árvore, tentando efetuar a operação desejada.

J 6.9.4 Contrariamente ao bloqueio- f (v. P 6.6.4), os procedimentos que encontram um nó n com bloqueio- c , não podem ficar em estado de espera até que n seja liberado, pois após a compactação a árvore deverá provavelmente ter sido alterada e o nó a ser usado pode nem ser o n .

6.10 Parâmetros Fornecidos pelo Usuário

Esses parâmetros devem ser fornecidos pelo usuário na operação de criação dos índices e de um arquivo de dados ou, se este último já existe, deve-se fornecer as suas especificações. A ausência de qualquer parâmetro descrito abaixo produz a utilização de um valor padrão. Notemos que deve haver possibilidade dos parâmetros serem fornecidos para cada arquivo ou globalmente para todos.

P 6.10.1 Fatores de ocupação de dados e de ordenação.

J 6.10.1 v. P 6.8.2.

P 6.10.2 Fator de ocupação inicial de dados para reorganização e carga inicial.

J 6.10.2 Impedir que logo após uma reorganização ou carga inicial, uma operação de inserção acarrete uma desorganização do arquivo.

P 6.10.3 Fator de ocupação da árvore para compactação.

J 6.10.3 v. P 6.9.1.

P 6.10.4 Fator de ocupação inicial para compactação.

J 6.10.4 Evitar que após uma compactação da árvore cada nó tenha 100% de ocupação,

e com isso qualquer atualização acarrete desdobramentos nos nós da árvore, destruindo a contiguidade física.

P 6.10.5 Fator de ocupação de desdobramento marginal.

J 6.10.5 v. P 6.7.3.

P 6.10.6 Tipo de reorganização (compactação), período, data e hora.

J 6.10.6 Necessário para efetuar reorganizações (compactações) manuais programadas e automáticas (v. P 6.8.2 (P 6.9.1)).

P 6.10.7 Parâmetros diversos.

1. Número esperado de registros de dados;
2. Tamanho médio e mínimo dos registros de dados;
3. Para cada índice:
 - (a) Tipo de índice (primário, secundário, arquivo-índice);
 - (b) Características como: unicidade, modificável, valor nulo, tipo(s) de compressão utilizada(s);
 - (c) Tamanho máximo do valor;
 - (d) Tamanho médio do valor;
 - (e) Multiplicidade média dos valores (caso de índices que não são chaves);
 - (f) Tamanho de cada nó;

J 6.10.7 Permitir que o gerenciador: (1, 2) reserve espaço para os arquivos de dados de tamanho limitado junto com os arquivos de índices associados (v. P 6.5.2); (3c) determine a quantidade de espaço livre a ser deixado em cada nó da árvore após uma concatenação, redistribuição ou desdobramento (v. D 6.6.5); (1, 3d, 3e, 3f) estipule o tamanho dos ponteiros para os nós da árvore, tamanho dos delimitadores usados em P 6.4.2 e do campo T (v. P 6.4.4.2). O tamanho mínimo dos registros de dados serve para controlar a presença de um endereço de página na árvore de espaço livre, como visto em P 6.5.5.

P 6.10.8 Tipo de arquivo (interno, externo ou expansível).

J 6.10.8 Permitir a determinação de endereçamentos relativos ou absolutos para os registros de dados e tamanho dos ponteiros.

P 6.10.9 Arquivos residentes.

J 6.10.9 v. P 6.5.6.

6.11 Quadro de Avisos (Informações para o Usuário)

Para cada arquivo de dados ou arquivo-índice deve existir um *quadro de avisos* para que o usuário possa tomar conhecimentos, em geral estatisticamente, do estado do arquivo e de suas árvores de índices.

P 6.11.1 Mensagens de sobrecaraga de espaço.

J 6.11.1 Informar que o espaço do disco reservado para os arquivos de tamanho limitado atingiu 90% de ocupação (v. P 6.5.3).

P 6.11.2 Estatísticas.

O sistema mantém rotinas estatísticas para indicar o fator de ocupação da árvore, o fator de ocupação e ordenação do arquivo de dados, a altura da árvore, o percentual de valores removidos e o percentual de ganho de espaço com o uso das técnicas de compressão (v. P 6.3.5).

J 6.11.2 Orientar o usuário em sua decisão de disparar manualmente compactações totais nas árvores e reorganizações nos arquivos de dados, eventualmente com ativação/desativação de compressão (v. P 6.9.1 e P 6.8.2).

Conclusão

O trabalho contém uma resenha completa sobre *árvores-B* mencionando provavelmente todos os trabalhos sobre a área até o momento, discorrendo sobre cada um com razoável detalhe. Ele pode, portanto, servir como ponto de partida para qualquer pesquisa na área e na implementação de gerenciadores de arquivos e de bancos de dados. Pode também servir de referência para cursos de estruturas de dados, gerenciamento de arquivos e de bancos de dados em nível de graduação e pós-graduação.

Pelo fato da *árvore-B* ser usada em provavelmente todos os gerenciadores de bancos de dados modernos esta estrutura tem uma grande importância prática. Dessa forma, o trabalho permite a realização de um estudo aprofundado nessa área e possibilita aos profissionais a obtenção de informações detalhadas, em uma abordagem homogênea sobre o assunto.

O trabalho apresenta contribuições originais expostas a seguir. 1) Um novo método de solução de concorrência para *árvores-B* e suas variações, utilizando apenas um único tipo de bloqueio na operações usuais de acesso (leitura, inserção, remoção e atualização), inclusive com o uso de valores de índice com tamanho variável, que permite realizar as operações usuais de acesso e de compactação da árvore de forma simultânea. 2) Um novo algoritmo de compactação da árvore, o qual garante um bom fator de ocupação e uma contigüidade física dos nós por nível e de níveis consecutivos. 3) Uma nova proposta de implementação de um gerenciador de arquivos destinado a usuários programadores, abrangendo inúmeros aspectos essenciais. 4) Nesse gerenciador temos: a) uso de *árvore-B⁺* que permite independência entre os arquivos de índice e de dados; b) um tipo de índice, denominado de *arquivo-índice*, o qual possibilita consultas do tipo “existe ou não um certo valor do índice”; c) diferentes estruturas para os nós da árvore dependendo do tipo de valor e de busca a ser realizada, juntamente com especificações de como implementar os delimitadores que separam os valores de tamanho variável dos ponteiros dentro de um nó; d) utilização de uma nova *árvore de espaço livre* que contém informações sobre os espaços não ocupados dentro de cada página do arquivo de dados; e) possibilidade de reorganização, manual ou automática, de um arquivo de dados; f) fornecimento de informações aos usuários através do que denominamos “quadro de avisos”. 5) Comparação entre as alturas da *árvore-B** e *árvore-B⁺*. 6) Um trabalho de “engenharia reversa” em um sistema comercial de gerenciador de arquivo que utiliza a estrutura *árvore-B*.

Este trabalho abre as seguintes possibilidades para futuras pesquisas: 1) simulações do método para determinar fatores limites de desdobramento ideal, de ocupação dos nós após uma compactação e para disparo automático da compactação; 2) extensão do método

para evitar desdobramentos prévios e concatenações adiadas nos nós localizados próximos à raiz, uma vez que o número destas operações nestes nós superiores é muito baixa em relação aos nós próximos às folhas; 3) simulações para comparação com outros métodos; 4) uma pesquisa sobre interface com usuários, através de levantamento de campo do que estes gostariam de receber como informações no “quadro de avisos”, bem como a maneira deles fornecerem os parâmetros necessários para o funcionamento do sistema.

Bibliografia

- [1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1975.

Consultado em aspectos gerais sobre tipos de árvores de busca.

- [2] D. Arnow and A.M. Tanenbaum. An empirical comparasion of B-trees, Compact B-trees and Multiway trees. *SIGMOD'84*, 14(2):33–46, june 1984.

Faz uma comparação do desempenho entre estes 3 tipos de árvores.

- [3] R.A. Baeza-Yates. Expected behavior of B^+ -trees under random insertions. *Acta Informatica*, 26:439–471, 1989.

Utiliza a técnica de *fringe analysis* para analisar o comportamento das *árvores- B^** sobre os seguintes aspectos: fator de ocupação de espaço, número de nós internos, altura e número de desdobramentos por inserção, com e sem o uso da técnica de overflow.

- [4] R.A. Baeza-Yates. Modeling splits in file structures. *Acta Informatica*, 26:349–362, 1989.

Analisa e compara o fator de ocupação de espaço em arquivos organizados com um índice em forma de *árvore- B* com os arquivos que utilizam o método de hashing, usando ou não a técnica de expansão antes de executar um desdobramento.

- [5] R.A. Baeza-Yates and P. Larson. Performance of B^+ -trees with partial expansions. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):248–257, june 1989.

Analisa o desempenho da *árvores- B^** que fazem uso da técnica de expansão dos nós ao invés de desdobrá-los, concluindo que o fator médio de ocupação de espaço é cerca de 81%.

- [6] R.A. Baeza-Yates. *Advances in Database Technology - EDBT 90*, volume 416 of

Lecture Notes in Computer Science, chapter An Adaptive Overflow Technique for B-Trees, pages 16–28. Springer, Veneza, march 1990.

Sugere uma nova técnica de overflow denominada de *técnica ajustável de overflow*, a qual opta por desdobrar ou redistribuir um nó folha de uma *árvore-B* com várias páginas. No final faz uma comparação entre todas as técnicas de overflow existentes.

- [7] R. Bayer. Binary B-trees for virtual memory. *ACM SIGFIDET Workshop*, pages 219–235, 1971.

Este artigo não foi localizado, no entanto seu conteúdo foi abordado por Wirth [96] e Ziviani [100].

- [8] R. Bayer. Symmetric Binary B-trees data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

Aborda uma classe de árvores binárias que permite fazer as operações de busca, inserção e remoção em tempo $O(\log n)$, onde n representa o número de nós da árvore.

- [9] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

Primeiro artigo a abordar sobre a estrutura de *árvore-B*, definindo-a, propondo e analisando os algoritmos de busca, inserção e remoção para um valor de índice e mostrando alguns resultados experimentais.

- [10] R. Bayer and J.K. Metzger. On the encipherment of search trees and random access files. *ACM Transactions on Database Systems*, 1(1):37–52, march 1976.

Propõem um método criptográfico para as *árvores-B* denominado de *chave de página* (page-key).

- [11] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

Sugere 4 soluções para o problema de concorrência nas *árvores-B**, as quais fazem uso de 3 tipos de bloqueios.

- [12] R. Bayer and K. Unterauer. Prefix B-tree. *ACM Transactions on Database Systems*, 2(1):12–26, march 1977.

Apresenta o conceito de *árvore-B* com pré-fixação simples e não simples, juntamente com os algoritmos para estas estruturas.

- [13] J.L. Bentley. Multidimensional bynary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, september 1975.

Aborda uma estrutura de dados que consiste de uma generalização da árvore de busca binária (*KD-Tree*). Esta estrutura armazena registros com K campos e permite buscas com campos múltiplos.

- [14] J.L. Bentley and J.H. Friedman. Data structures for range searching. *Computing Surveys*, 11(4):397–409, december 1979.

O artigo contém uma resenha sobre várias estruturas de dados que permitem consultas com campos múltiplos.

- [15] W. Boswell and A.L. Tharp. *Advances in Computing and Information*, volume 468 of *Lecture Notes in Computer Science*, chapter Alternatives to The B^+ -Tree, pages 266–274. Springer, may 1990.

Exemplifica 2 estruturas de arquivos para aplicações que requerem acesso sequencial e direto: *Bounded Disorder* e *Adaptive Hashing*. Faz uma análise comparativa entre estas estruturas e a *árvore- B^+* .

- [16] M.R. Brown. A storage for height balanced trees. *Inf. Processing Letters*, 7(5):231–232, august 1978.

O autor sugere uma representação do fator de balanço das árvores AVL, usando um bit em cada nó. Quando o nó não tem filhos, o fator é 0. Quando o nó tem um filho, o fator é (+,-)1. Quando o nó tem 2 filhos, então cada filho guarda seu fator de balanço e os dois juntos formam o fator do pai.

- [17] F.W. Burton, J.G. Kollias, D.G Matsakis, and V.G. Kollias. Implementation of overlapping B-trees for time and space efficient representation of collections of similar files. *The Computer Journal*, 33(3):279–280, 1990.

Apresenta, de forma resumida (short note), algoritmos para manipular *árvores-B* que possuem arquivos com conteúdos similares, isto é, arquivos com várias versões.

- [18] J.H Chu and G.D. Knott. An analysis of B-trees and their variants. *Information Systems*, 14(5):359–370, 1989.

Analisa o fator de ocupação médio e o comprimento do caminho médio nas *árvores-B* e em algumas de suas variações, através de equações de recorrência no número de nós com determinados tamanhos, apresentando verificações experimentais.

- [19] D. Comer. The Ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

Este artigo é uma resenha (survey) abordando de forma sucinta muitos dos tópicos relacionados com o tema *árvore-B*.

- [20] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw Hill Book Company, New York, 1990.

Consultados em aspectos gerais sobre tipos de árvores de busca.

- [21] J.O. Costa. Utilização das árvores-B na administração de arquivos com independência de dados. Master's thesis, Universidade Federal de Pernambuco, Recife, 1980.

Tese de dissertação de mestrado cujo objetivo é desenvolver um sistema gerenciador de arquivos que permite a independência de dados através do uso das *árvores-B*.

- [22] C.J. Date. *Bancos de Dados - Tópicos Avançados*, volume 2. Editora Campus, Rio de Janeiro, 1984.

Consultado em aspectos gerais de concorrência e criptografia em sistemas de banco de dados.

- [23] G. Diehr and B. Faaland. Optimal pagination of B-tree with variable-length items. *Communications of the ACM*, 27(3):241–247, march 1984.

Apresentam um algoritmo de paginação para *árvores-B* com elementos de tamanho variável que gasta tempo $O(n \log n)$ e um algoritmo para construir uma *árvore-B* de altura mínima em tempo $O(n^3 \log n)$, onde n é a quantidade de elementos.

- [24] E.W. Dijkstra. The structure of the multi-programming system. *Communications of the ACM*, pages 341–346, 1968.

Introduz a noção de *semáforos* para manipular os problemas de concorrência.

- [25] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structure persistent. *Journal of Computer and Systems Science*, 38(1):86–124, february 1989.

Abordam as estruturas de dados denominadas de *persistentes*, isto é, que mantêm várias versões dos dados, necessárias para determinadas aplicações.

- [26] B. Eisenbarth, N. Ziviani, G.H. Gonnet, K. Mehlhorn, and D. Wood. The theory of fringe analysis and its application to 2-3 trees and B-trees. *Information and Control*, 55:125–174, 1982.

Introduziram o método de *fringe analysis* para determinar estimativas para o fator de ocupação, número médio de nós, probabilidade de ocorrer um desdobramento e altura das *árvores-2-3* e *árvores-B*, com ou sem o uso da técnica de overflow.

- [27] C.S. Ellis. Concurrent search and insertion in 2-3-trees. *Acta Informatica*, 14(1):63–86, 1980.

Apresenta 2 algoritmos que permitem um alto grau de concorrência nas *árvores-2-3* sem ocasionar deadlock (Bloqueio-2-3 e Pipeline-2-3).

- [28] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, september 1979.

Introduz e analisa o método de hashing extensível.

- [29] D.E. Ferguson. Bit-tree: A data structure for fast file processing. *Communications of the ACM*, 35(6):114–120, june 1992.

Esta variação caracteriza-se por usar a técnica de pré-fixação nos nós não folhas e o método de *bit de distinção* nas folhas, aumentando a quantidade de elementos em cada nó.

- [30] R.A. Finkel and J.L. Bentley. Quad-trees - a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

O artigo trata sobre uma estrutura de dados que permite armazenar informações que serão recuperadas usando-se campos múltiplos. Os algoritmos de busca e inserção apresentados permitem recuperar dados com 2 dimensões.

- [31] I. Gagliardi and D. Merelli. Minimum boundary-length problem in B^* -tree: A case study. *Inf. Processing Letters*, 16(5):559–564, 1991.

Analysaram o uso dos algoritmos de Dierh e Faaland [23] em um sistema com *árvore-B* denominado de DOMINO.

- [32] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annual Symposium Foundation of Computer Science*, pages 8–21, 1978.

Abordam uma técnica de coloração para binarizar árvores e torná-las sempre balanceadas. Propõem também um novo algoritmo de atualização para *árvores-B*, o qual executa pré-desdobramentos nos nós durante a fase de busca.

- [33] G.K. Gupta and B. Srinivasan. Approximate storage utilization of B-trees. *Inf. Processing Letters*, 22:243–246, april 1986.

Aponta 2 falhas na demonstração de Leung [54] para determinar o fator de ocupação médio das *árvores-B*.

- [34] G. Held and M. Stonebraker. B-trees re-examined. *Communications of the ACM*, 21(2):139–143, february 1978.

O artigo propõe o uso de estruturas dinâmicas em sistemas de banco de dados e aborda problemas como concorrência, crescimento da altura e atualização dos índices.

- [35] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Rockville, Maryland, 1976.

Consultado em aspectos gerais sobre tipos de árvores de busca.

- [36] S. Huddleston and K. Melhorn. *Robust Balancing in B-Trees*, volume 104 of *Lecture Notes in Computer Science*, pages 234–244. Springer, Berling, 1981.

Propõem uma técnica para manter os custos de balanço assintoticamente linear.

- [37] IBM-SC26-4749-0. *IBM - MVS/DFP - Using Data Sets*. Version 3 - Release 3, 5th edition, march 1991.

Manual do sistema VSAM que aborda principalmente a estrutura de dados.

- [38] IBM-GC26-4151-5. *IBM - MVS/Extended Architecture - VSAM Administration Guide*. Version 2 - Release 4, 6th edition, september 1989.

Manual do sistema VSAM que trata basicamente da arquitetura.

- [39] IBM-GG24-1563. *IBM - VSAM Primer for the ICF Catalogue Facility in MVS Environment*.

Manual do sistema VSAM.

- [40] T. Johnson and D. Shasha. A framework for the performance analysis of concurrent B-trees algorithms. In *Proc. 9th Symposium on PODS*, 1990.

Não foi possível localizar este artigo.

- [41] D.G. Keehn and J.O. Lacy. VSAM data set design parameters. *IBM Systems Journal*, 30(3):186–212, 1974.

Descreve o sistema VSAM da IBM.

- [42] M.A. Keller and G. Wiederhold. Concurrent use of B-trees with variable-length entries. *Sigmod Records*, 17(2):89–90, june 1988.

Propõem uma solução para o problema de concorrência em *árvores-B* com valores de tamanho variável.

- [43] T.M. Klein, K.J. Parzignat, and A.L. Tharp. Optimal B-tree packing. *Inf. Processing Letters*, 16(2):239–243, 1991.

Apresenta um algoritmo para construir *árvores-B* ótimas em termos de ocupação de espaço.

- [44] D.E. Knuth. *The Art Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

Aborda várias das estruturas apresentadas.

- [45] H.F. Korth and A. Silberchatz. *Database System Concepts*. Mc. Graw Hill Inc., New York, 1986.

Consultado em aspectos de organização, implementação, projeto e seleção de arquivos utilizados em banco de dados, em particular para os arquivos indexados.

- [46] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, september 1980.

Trata de concorrência em *árvores* de busca binária.

- [47] K. Kuspert. Storage utilization in B^* -trees with a generalized overflow technique. *Acta Informatica*, 19(1):35–55, 1983.

Define a técnica de *desdobramento marginal* (marginal split) e analisa o fator de utilização de espaço nas *árvores-B**, construídas por inserções aleatórias e que fazem uso da técnica de overflow nas folhas.

- [48] Y.S. Kwong and D. Wood. In *Proc. MFCS*, volume 88 of *Lecture Notes in Computer Science*, chapter Approaches to Concurrency in B-Trees, pages 402–413. Springer, 1980.

Expõem os problemas de concorrência com *árvores-B* e analisam algumas soluções existentes.

- [49] Y.S. Kwong and D. Wood. In *Proc. 4th International Symposium Programming*, volume 83 of *Lecture Notes in Computer Science*, chapter Concurrent Operations in Large Ordered Indexes, pages 208–221. Springer, 1980.

Apresentam soluções para o problema de concorrência nas *árvores-B** e propõem uma nova estrutura denominada de *árvore-T* (uma árvore de árvores).

- [50] Y.S Kwong and D. Wood. A new method of concurrency in B-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, may 1982.

Propõem uma solução para o problema de concorrência em *árvores-B**, fazem uma comparação das várias soluções existentes e tratam o caso de concorrência dentro de um nó.

- [51] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, november 1977.

O autor exhibe vários algoritmos para o problema de concorrência. Todos estes algoritmos permitem o compartilhamento dos dados, pois fazem uso da técnica de ler os dados em um sentido e escrevê-los em outro.

- [52] S. Lanka and E. Mays. Fully persistent B^+ -tress. In *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, 20(2):426–435, 1991.

Apresenta variações da *árvore-B**, que permitem manter diferentes versões do banco de dados. Para isto utilizam as técnicas de *método do nó gordo*, *método do campo gordo* e *método do bloco de versão pura*.

- [53] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-tree. *ACM Transactions on Database Systems*, 6(4):650–670, december 1981.

Propõem uma solução de concorrência para as *árvores-B** fazendo uma ligeira modificação na estrutura e garantindo o bloqueio de um único nó em cada instante de um processo em execução.

- [54] C.H.C Leung. Approximate storage utilization of B-trees: A simple derivation and generalization. *Inf. Processing Letters*, 19:199–201, 1984.

Demonstra de uma maneira simples que o fator de ocupação médio das *árvores-B* é de 69,3% e da *árvores-B^O* é 81%.

- [55] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. 6th Conference on Very Large Databases*, pages 212–223, Montreal, 1980.

Introduz o método de hashing linear.

- [56] W. Litwin and D.B. Lomet. A new method for fast data searches with keys. *IEEE Transactions on Software Engineering*, pages 16–24, march 1987.

Aborda uma variação da *árvore-B* que permite a existência de várias páginas em cada nó folha, reunindo as vantagens das árvores indexadas e do método de hashing.

- [57] D.B. Lomet. Multi-table search for B-tree files. *ACM SIGMOD Conference*, pages 35–42, 1979.

Consiste de um técnica para ser usada com as *árvores-B* que contêm valores comprimidos, na qual os dados são organizados em tabelas, pelo tamanho dos mesmos. Dessa forma, há uma redução na ocupação de espaço. O algoritmo de busca proposto é competitivo com os demais existentes no que diz respeito ao tempo de execução.

- [58] D.B. Lomet. Digital B-tree. In *Proc. of the 7th Conference on Very Large Databases*, pages 333–343, Cannes, September 1981.

Retrata uma variação da *árvore-B* que permite um aumento no número de informações nos nós através da existência de várias páginas em cada nó. Baseando-se no método de hashing extensível é possível recuperar uma única página para cada nó visitado, durante uma operação de busca.

- [59] D.B. Lomet. Partial expansions for files organizations with an index. *ACM Transactions on Database Systems*, 12(1):65–84, march 1987.

Sugere o uso de expansões dos nós para arquivos organizados como um índice, ao invés de desdobrá-los, permitindo com isso um aumento no fator de ocupação médio do espaço.

- [60] D.B. Lomet. A simple bounded disorder file organization with good performance. *ACM Transactions on Database Systems*, 13(4):525–551, december 1988.

Retrata uma organização dos dados em nós da *árvore-B* permitindo a existência de várias páginas em um único nó folha. Em cada operação de busca apenas uma página é recuperada através da técnica de hashing linear.

- [61] D.B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, december 1990.

Estrutura de dados que permite buscas com campos múltiplos possuindo algumas características das *árvores-B*.

- [62] D. Maier and C. Salveter. Histerical B-trees. *Inf. Processing Letters*, 12(4):199–202, august 1981.

Retrata uma variação da *árvore-B*, permitindo nós com um mínimo de $K - p$ valores do índice, onde K é a ordem da árvore e $p \leq \frac{K-1}{2}$. Com isso é possível reduzir o número de desdobramentos e concatenações na árvore, a um valor linear no número de inserções e remoções realizadas.

- [63] K. Maruyama and S.E. Smith. Analysis of design alternatives for virtual memory indexes. *Communications of the ACM*, 20(4):245–254, april 1977.

Propõem uma forma de organização de índices similar às *árvores-B*, analisando o desempenho da mesma para vários tipos de operações de busca com ou sem o uso de técnicas de compressão nos valores.

- [64] E.M. McCreight. Pagination of B^* -trees with variable length records. *Communications of the ACM*, 20(9):670–674, september 1977.

Apresenta um algoritmo de paginação para *árvores-B* com elementos de tamanho variável, isto é, para reorganizar a árvore em tempo $O(n \log n)$, onde n é o número de elementos a serem distribuídos.

- [65] MEDIDATA Informática Tecnologia S.A. *Manual do Usuário dos Sistema MUMPS com Árvore Balanceada*. Documento 41-02011-1, Rio de Janeiro, 1981.

Manual do sistema MUMPSda MEDIDATA.

- [66] MEDIDATA Informática Tecnologia S.A. *MUMPS Guia do Usuário*. Editora McGraw Hill LTDA, Rio de Janeiro, 1986.

Livro que aborda a linguagem MUMPS.

- [67] E.R. Miller and L. Snyder. Multiple access to B-trees. In *Proc. Conference Information Sciences and Systems*, pages 400–407, John Hopkins University - Baltimore, march 1978.

Tratam do problema de concorrência em *árvores-B* sugerindo uma solução na qual os nós bloqueados estão a uma distância pequena daquele que sofrerá modificação.

- [68] E.R. Miller, N. Pippenger, A.L. Rosenberg, and L. Snyder. Optimal 2-3 trees. *Siam Journal of Computing*, 8(1):42–59, february 1979.

Os autores propõem um algoritmo de tempo linear para construir *árvores-2-3* de espaço ótimo, isto é, baixas, a partir de um conjunto de itens e da determinação de um perfil para este conjunto. O algoritmo é facilmente adaptável para as *árvores-B*.

- [69] Y. Mond. Concurrency control in B^+ -trees databases based on preparation operations. Master's thesis, The Technion - IIT, Haifa, 1984.

Não tivemos acesso a esta tese.

- [70] Y. Mond and Y. Raz. Concurrency control in B^+ -trees databases using preparatory operations. In *Proc. of the 11th International Conference on Very Large Database*, pages 331–334, Stockholm, 1985.

Propõem um controle de concorrência para as *árvores-B** evitando um número elevado de nós bloqueados por cada processo em execução, através do uso de operações preparatórias que garantem a segurança dos nós para inserções e remoções.

- [71] S. Nagayama. Tabelas de decisão e implementação de gerador I-M-E. Master's thesis, Instituto de Matemática e Estatística - USP, São Paulo, 1990.

Contém extensa resenha sobre tabelas de decisão, bem como a implementação do gerador de aplicações I-M-E.

- [72] T. Nakamura and T. Mizoguchi. An analysis of storage utilization factor in block split data structuring schemes (SUF). In *Proc. 4th International Conference on Very Large Database Conference*, pages 489–495, Berlim, 1978.

Através de um diagrama de transição probabilística os autores determinam equações de recorrência no número de nós da *árvore-B* e provam que o fator de ocupação médio nesta estrutura é de $\ln 2 = 69,31\%$.

- [73] J. Navon. Concurrency and recovery for a simple database system. Master's thesis, The Technion - IIT, Haifa, 1984.

Não tivemos acesso a esta tese.

- [74] Novell, Inc. Development Products Division. *BTRIEVE Reference Manual / Operation Manual*.

Manual de referência e de operação do sistema BTRIEVE de organização de arquivos de índices usando *árvores-B*.

- [75] O. Nurmi, E. Soisalon-Soinien, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proc. ACM PODS*, pages 170–176, march 1987.

Não foi possível localizar este artigo.

- [76] Y. Ohsawa and M. Sakauchi. The BD-tree: A new n-dimensional data structure with highly efficient dynamic characteristics. *Inf. Processing Letters*, 83, 539-544 1983.

Propõem uma estrutura de dados que permite a realização de consultas com campos múltiplos possibilitando eficiência de tempo e de espaço. A estrutura é organizada por um método de particionamento com uma *zona discriminante*.

- [77] J.A. Orestein. Multidimensional tries used for associative searching. *Inf. Processing Letters*, 14(4):150-157, 1982.

Baseando-se na *árvore-KD*, foi proposta uma estrutura k-dimensional para realizar consultas com campos múltiplos. Esta estrutura possui um bom desempenho para as buscas e atualizações.

- [78] J.R. Parr. An access method for concurrently sharing a B-tree index. Technical Report 36, University of Western Ontario, Department of Computer Science, april 1977.

Artigo não localizado (citado em [84]).

- [79] K.H. Quitzow and M.R. Klopprogge. Space utilization and access path length in B-trees. *Information Systems*, 5:7-16, 1980.

O artigo descreve um modelo determinista para calcular o fator de ocupação de espaço e o comprimento dos caminhos nas *árvores-B*. O modelo pode ser adaptado para os casos em que se utilizam as técnicas de overflow, concatenação, redistribuição e desdobramento.

- [80] J.T. Robinson. The KDB-tree: A search structure for large multidimensional dynamic indexes. *Proc. ACM Sigmod Conference on Management of Data*, 14(2):10-18, june 1984.

A referência citada não está correta, impossibilitando a localização do artigo (citado em [60]).

- [81] A.L. Rosenberg and L. Snyder. Compact B-tree. *SIGMOD Conference Boston*, pages 43-51, 1979.

Definem a noção de *árvore-B compactada* e propõem um algoritmo para construir esta árvore a partir de uma não compactada.

- [82] A.L. Rosenberg and L. Snyder. Time and space optimality in B-tree. *ACM Transactions on Database Systems*, 6(1):174–183, march 1981.

Este artigo possui o mesmo conteúdo de [81].

- [83] F. Rubin. Experiments in text file compression. *Communications of the ACM*, 1976.

Aborda algumas técnicas de compressão de arquivos.

- [84] Y. Sagiv. Concurrent operations on B^* -trees with overtaking. *Journal of Computer and Systems Science*, 33:275–296, 1986.

Aperfeiçoou a solução proposta por Lehman e Yao [53] utilizando uma técnica de redistribuição denominada de *compressão* que pode ser utilizada concorrentemente com outras operações.

- [85] B. Samadi. B-trees in a system with multiple users. *Inf. Processing Letters*, 5(4):107–112, october 1976.

Sugere uma solução para o problema de concorrência nas *árvores-B* fazendo uso da técnica padrão de *semáforos*.

- [86] B. Seeger and P. Larson. Multi-disk B-trees. In *Proc. of the 1991 ACM SIGMOD International Conference on Management Data*, 20(2):436–445, 1991.

Os autores apresentam de maneira resumida, 3 formas de manusear com as *árvores-B* que utilizam vários discos, propondo um novo esquema que permite um bom balanço local.

- [87] V.W. Setzer. *Bancos de Dados - Conceitos, Modelos, Gerenciadores, Projeto Lógico e Projeto Físico*. Editora Edgar Blucher, São Paulo, 1986.

Consultado sobre projeto de chaves e estruturas internas do modelo relacional não normalizado.

- [88] R.F. Sincovec and R.S. Wiener. *Balanced Trees Using Secondary Storage: B-Trees Data Structure Using Modula-2*. John Wiley, 1986.

Define *árvore-B* e aborda implementações dos algoritmos de busca, inserção e remoção em Modula-2.

- [89] R.M.F. Souza and O.S.F. Carvalho. Controle de concorrência em *árvores-b*. *IV-Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho (IV SBAC PAD)*, pages 397–412, 1992.

Apresentam algoritmos de busca e inserção para o problema de concorrência nas *árvores-B* com elementos de tamanho variável. Estes algoritmos permitem o uso da técnica de overflow.

- [90] V. Srinivasan and M.J. Carey. Performance of B-tree concurrency control algorithms. In *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, 20(2):417-425, 1991.

O artigo fala de alguns algoritmos de concorrência para as *árvores-B* fazendo uma comparação do desempenho dos mesmos através de um modelo de simulação.

- [91] A.S. Tanenbaum. *Operating Systems, Design and Implementation*. Prentice-Hall Software Series, New Jersey, 1987.

Consultado em aspectos gerais sobre concorrência.

- [92] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, second edition, 1982.

Consultado em aspectos gerais sobre tipos de árvores de busca.

- [93] R.E. Wagner. Indexing design considerations. *IBM System Journal*, (4):351-367, 1973.

Define os tipos de índices, as operações de inserção, remoção e atualização em arquivos de índices e propõe técnicas de compressão de caracteres para os valores dos índices e de endereçamento relativo.

- [94] H. Wedekind. On the selection of access paths in a database system. *Database Management*, J.W. Klimbie and K.L. Koffeman, North Holland, Amsterdam, pages 385-397.

Define as *árvores-B** fazendo uma comparação qualitativa entre as alturas destas árvores e das *árvores-B*.

- [95] G. Wiederhold. *File Organization for Database Design*. Computer Science Series, New York, 1987.

Consultado por abordar vários aspectos de arquivos com índices.

- [96] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall Inc., New Jersey, 1976.

Consultado em aspectos gerais sobre tipos de árvores de busca e principalmente sobre as *árvores-B binárias e simétricas*.

- [97] W.E. Wright. Some average performance measures for the B-tree. *Acta Informatica*, pages 541–557, 1985.

Apresenta fórmulas e resultados de simulação para medir o desempenho médio de vários aspectos das *árvores-B* com uma grande quantidade de valores, tais como: número de nós folhas de cada tamanho, probabilidade de se fazer inserção em um nó de um determinado tamanho, probabilidade de ocorrer um desdobramento durante uma inserção, fator de ocupação e altura esperada.

- [98] A.C. Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978.

Analisa o número médio de nós e o fator de ocupação para as *árvores-2-3*, estendendo para as *árvores-B*, através de um método denominado por Eisenbarth de *fringe analysis*.

- [99] B. Zhang and M. Hsu. Unsafe operations in B-trees. *Acta Informatica*, 26:421–438, 1989.

Desenvolveram um modelo para calcular o número de operações *não seguras* nas *árvores-B*, isto é, operações do tipo desdobramento, concatenação e redistribuição. O modelo é caracterizado pelo número de operações de inserções e remoções que podem ocorrer em um nó antes de uma operação *não segura*.

- [100] N. Ziviani, H.J. Olivié, and G.H. Gonnet. The analysis of an improved symmetric binary B-tree algorithm. *The Computer Journal*, 28(4):417–425, 1985.

Apresenta uma mudança nos algoritmos de inserção e remoção para as *árvores-B* binárias simétricas.