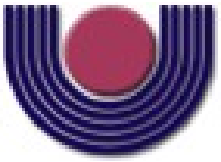

UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Informática

Curso de Bacharelado em Informática

Estrutura de Índices para Arquivos

Jhonata Rodrigo de Peder

Marcelo Schuck

CASCAVEL

2009

JHONATA RODRIGO DE PEDER

MARCELO SCHUCK

ESTRUTURA DE ÍNDICES PARA ARQUIVOS

Monografia apresentada a disciplina de Banco de Dados I, referente a nota parcial para disciplina.

Orientador: Carlos J.M. Olguín

CASCAVEL

2009

JHONATA RODRIGO DE PEDER

MARCELO SCHUCK

ESTRUTURA DE ÍNDICES PARA ARQUIVOS

Monografia apresentada à disciplina de Banco de Dados, referente a nota parcial da disciplina.

Prof. Carlos J. M. Olguin

Colegiado de Informática, UNIOESTE

Cascavel, 20 de setembro de 2009.

Lista de Figuras

Lista de Abreviaturas e Siglas

SGBDs	Sistema de Gerenciamento de Banco de dados
IBM	International Business Machines Corporation
log	logaritmo
CEP	Código de Endereço Postal
NUD	Numero de Departamento

Sumário

Resumo

O **Índice** é um arquivo auxiliar associado a uma tabela. Sua função é acelerar o tempo de acesso às linhas de uma tabela, criando ponteiros para os dados armazenados em colunas especificas. O Banco de Dados usa o Índice de maneira semelhante ao índice remissivo de um livro[7], verifica um determinado assunto no índice e depois localiza a sua posição em uma determinada página. De acordo com a estrutura de dados utilizada no banco de dados, determinada estrutura de indexação torna-se mais eficiente para realização de consultas, tornando o retorno de uma solicitação mais rápida, precisa e eficiente.

Palavras-chave: Árvore, Banco de Dados, Índice.

Capítulo 1

Introdução

Os índices utilizados para acelerar a recuperação de registros em resposta a determinadas condições de pesquisas, constitui-se, estruturas de acessos auxiliares[5]. A estrutura de índices geralmente oferece caminhos de acesso secundário que fornecem meios alternativos para acessar os registros sem afetar o seu posicionamento físico no disco. Eles possibilitam acesso eficiente a registros, com base nos campos de indexação que são utilizados para construir índices.

Basicamente, qualquer campo do arquivo pode ser utilizado para criar um índice e múltiplos índices em diferentes campos podem ser construídos no mesmo arquivo. É possível existir uma variedade de índices, com cada um deles usando uma determinada estrutura de dados para acelerar a pesquisa. Para encontrar um registro ou registros num arquivo, com base em um determinado critério de seleção num campo de indexação, é necessário inicialmente acessar um índice, que aponta para um ou mais blocos no arquivo onde os registros solicitados estão localizados. Os tipos mais predominantes de índices são baseados em arquivos ordenados, ou seja, índices de nível único, e estruturas de dados do tipo árvore, índices multinível e árvores–B^+, os índices também podem ser construídos com base em hashing e outras estruturas de pesquisa de dados.

Os índices ordenados de nível único podem ser classificados em: principal; secundário; e clustering. Visualizando um índice de nível único, como um arquivo ordenado, pode-se adicionar índices adicionais para o mesmo, fazemos surgir o conceito de índices multinível.

Dentre as estruturas de índice multinível, as árvores – B^+ se tornaram uma estrutura padrão na maioria dos SGBDs relacionais, para gerar índices mediante demanda.

No capitulo 2 será apresentado os conceitos referentes aos tipos de índices ordenados de nível único, como índices principais, índices por clustering e índices secundários. No capitulo 3 apresentam-se os conceitos de índices multiníveis e os conceitos de índices multinível dinâmicos utilizando árvores-B e árvores-B^+. Índices em chaves com mais de um atributo serão apresentado no capitulo 4, e demais conceitos serão explanados no capitulo 5, sendo a conclusão obtidas apresentada no capitulo 6.

Capítulo 2

Índices Ordenados de Nível Único

A grande quantidade de dados presente num banco de dados encontram-se dispostas de forma seqüencial, numa memoria secundária. Para um acesso eficiente a algum dado requerido é necessário a utilização de algum método que utiliza as consultas a serem realizadas. Como exemplo, tem-se o índice remissivo para encontrar determinada palavra presente em final de livros, sendo este conceito aplicado a banco de dados através de estrutura de índices.

Para um arquivo com uma dada estrutura de registros consistindo de diversos campos, ou atributos, uma estrutura de acesso de índice é geralmente definida num único campo do arquivo, chamado campo de indexação, ou atributo de indexação. O índice geralmente armazena cada valor do campo índice juntamente com uma lista de ponteiros para todos os blocos de disco que contém registros com aquele valor de campo. Os valores no índice são ordenados de forma que possa ser realizada uma pesquisa binária no índice.

O arquivo índice é bem menor que o arquivo de dados de modo que pesquisar um índice com uma pesquisa binária é razoavelmente eficiente. Existem diversos tipos de índices ordenados. Um índice principal é especificado no campo chave de ordenação de um arquivo de registros ordenados, uma vez que os registros de arquivo em disco possuem valor exclusivo para aquele campo. Se inúmeros registros no arquivo podem possuir o mesmo valor para o campo de ordenação, não sendo este um campo chave, este índice passa a ser chamado de índice clustering. Por possuir no máximo um campo de ordenação física, apenas um índice principal ou um índice clustering pode ser utilizado.

Um terceiro tipo de índice, chamado índice secundário, pode ser especificado em qualquer campo não ordenado de um arquivo. Um arquivo pode possuir diversos índices secundários alem de seu método de acesso principal. A diferença entre índice secundário e índice primário está no fato de que o índice secundário não determina a colocação de registros no banco de dados.

2.1 Índices Principais

Um índice principal é um arquivo ordenado cujos registros são de tamanho fixo com dois campos. O primeiro campo é do mesmo tipo de dado que o campo chave de ordenação, chamada de chave primária, do arquivo de dados e o segundo campo é um ponteiro para um bloco do disco, ou seja, um endereço de bloco. Existe uma entrada de índice no arquivo índice para cada bloco no arquivo de dados, possuindo o valor do campo-chave primário para o primeiro registro num bloco e um ponteiro para aquele bloco com seus dois valores de campo. O número total de entradas no índice é o mesmo que o número de blocos de disco no arquivo ordenado de dados.

Considerando uma estrutura cuja chave-primária é o nome de determinada pessoa, e este nome não podendo ser repetido, e aplicando a estrutura para indexação para um nome i com a seguinte formulação <K(i), P(i)>, e tendo os nomes Aaron, Adams e Alexandre, um processo de busca, através de índices principais, para esses nomes, se comportaria da seguinte forma:

<K(1) = (Aaron, Ed),P(1) = endereço do bloco 1>

<K(2) = (Adams, John),P(2) = endereço do bloco 2>

<K(3) = (Alexander, Ed),P(3) = endereço do bloco 3>

Os índices também podem ser caracterizados como densos, possuindo uma entrada de índice para cada valor deixado de pesquisa, e portanto para cada registro, no arquivo de dados, ou índices esparsos que por outro lado possui entradas de índice para somente alguns dos valores de pesquisa. Índice principal é portanto um índice não denso, ou esparso, uma vez que inclui uma entrada para cada bloco de disco do arquivo de dados em vez de para cada valor de pesquisa.

O arquivo índice para um índice principal precisa substancialmente de menos blocos que um arquivo de dados, por duas razões. Primeiramente, existem menos entradas de índice do que registros no arquivo de dados. Em segundo lugar, cada entrada de índice é geralmente

menor em tamanho do que um registro de dados, porque possui somente dois campos, conseqüentemente, mais entradas de índice do que registro de dados podem caber em um bloco. Uma pesquisa binária no arquivo índice requer portanto menos acessos a blocos do que uma pesquisa binária no arquivo de dados.

Dependendo do tipo de chave primária a ser adotado, pode ser realizada um agrupamento de dados em blocos. Estes blocos encontram-se dispostos de forma seqüencial. A utilização de índice principal pode prover um acesso direto a cabeça deste bloco evitando assim a quantia excessiva de acessos ao banco, como apresentada na Figura 2.1, justificando as vantagens citadas anteriormente. Este primeiro registro em cada bloco do arquivo de dados é chamado de registro âncora do bloco ou simplesmente âncora do bloco.

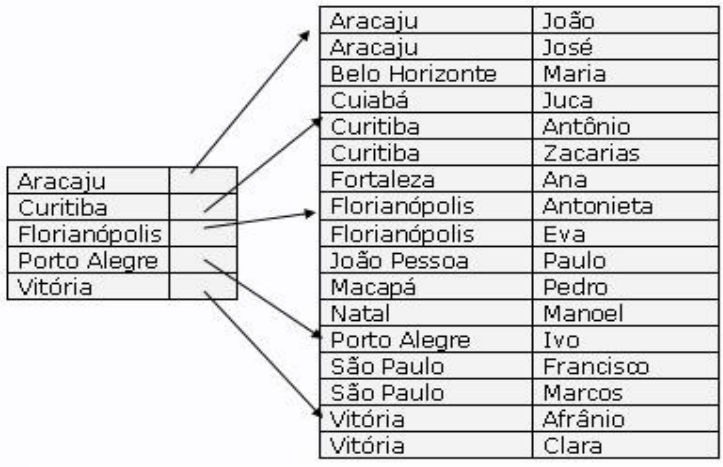

Figura 2.1: Exemplo de aplicação de índice principal para uma consulta.

Esta estruturação de índices apresenta grandes problemas como qualquer arquivo ordenado referente ao processo da inclusão ou exclusão. Com um índice principal o problema é aumentado porque o processo de inserir um registro em sua posição correta no arquivo de dados ocasiona um movimento que poderá alterar os registros âncora de um bloco, dissipando um grande custo computacional para garantira a efetiva ordenação de todo o bloco. Para minimizar este problema, a utilização de um arquivo de *overflow* torna-se eficiente

2.2 Índices Clustering

Se os registros de um arquivo estiverem fisicamente ordenados por um campo que não seja chave, que não possui um valor distinto para cada registro, esse campo é chamado de campo

clustering. Podemos criar um tipo diferente de índice, chamado índice *clustering*, para acelerar a recuperação de registros que possuem o mesmo valor para o campo *clustering*. Isso difere de um índice principal que requer que o campo de ordenação do arquivo de dados possua um valor distinto para cada registro.

Um índice *clustering* também é um arquivo ordenado com dois campos; o primeiro campo é do mesmo tipo do campo *clustering* do arquivo de dados e o segundo campo é um ponteiro para o bloco. Existe uma entrada no índice *clustering* para cada valor distinto do campo *clustering*, que contém o valor e um ponteiro para o primeiro bloco no arquivo de dados que possua um registro com aquele valor para seu campo *clustering*. Um exemplo de índices *clustering* é apresentado na Figura 2.2.

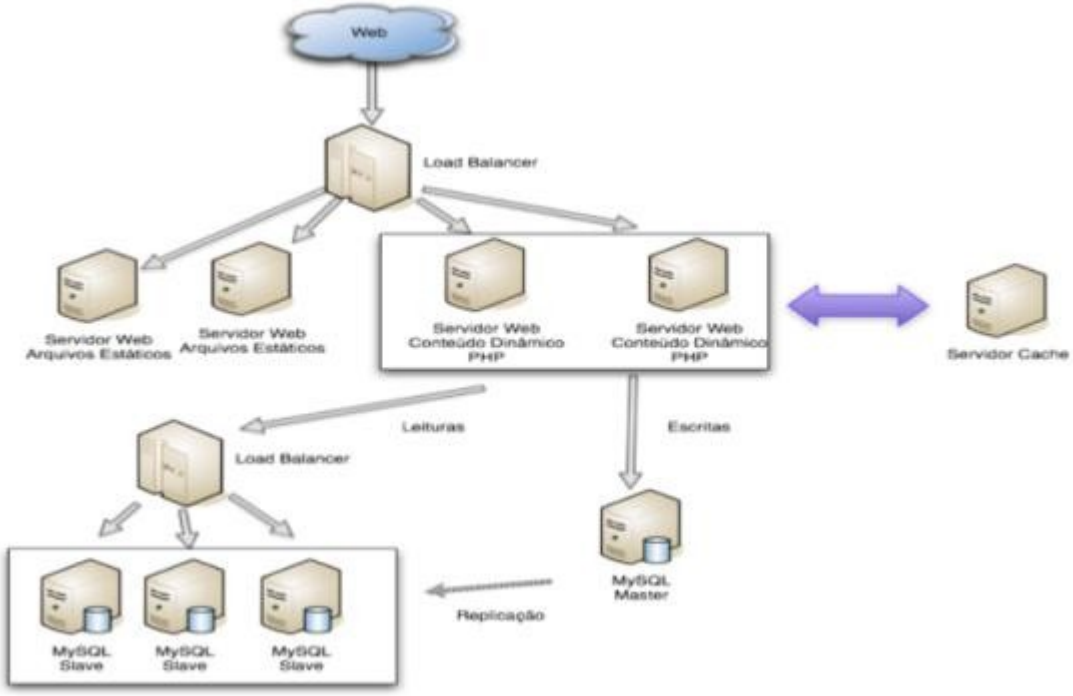

Figura 2.2: Exemplo de aplicação de índice por *clustering*.

Um índice *clustering* é um exemplo de índice não denso, porque possui uma entrada para cada valor distinto do campo de indexação, em vez de para cada registro no arquivo. A inclusão e a exclusão de registros, também causarão problemas neste método, porque os registros de dados estão fisicamente ordenados, repetindo-se os mesmos problemas observados para a indexação com índices principais.

2.3 Índices Secundários

Um índice secundário é também um arquivo ordenado com dois campos. O primeiro campo é do mesmo tipo de dados que um campo sem ordenação do arquivo de dados que seja um campo de indexação. O segundo campo é um ponteiro para um bloco ou para um ponteiro de registro. Podem existir muitos índices secundários para o mesmo arquivo.

Consideramos primeiramente uma estrutura de acesso de índice secundário num campo chave que possui um valor distinto para cada registro. Esse campo é as vezes chamado de chave secundária. Neste caso, existe uma entrada de índice para cada registro no arquivo de dados, que contém o valor da chave secundária para o registro e um ponteiro para o bloco no qual o registro está armazenado ou para o próprio registro, tornando-se um índice denso.

As entradas estão ordenadas de acordo com os valores apresentados por índices principais no seguinte esquema <K(i), P(i)>, com isso pode-se realizar uma pesquisa binária. Uma vez que os registros do arquivo de dados não estão fisicamente ordenados pelos valores do campo chave secundário, não podemos utilizar âncoras de bloco. É por isso que uma entrada de índice é criada para cada registro no arquivo de dados, em vez de para cada bloco, como é o caso de um índice principal. A Figura 2.3 exemplifica a ocorrência de uma estrutura de índice secundário.

Pode-se criar um índice secundário em um campo que não seja chave, podendo possuir, inúmeros registros no arquivo de dados, o mesmo valor para o campo de indexação. Existem algumas opções para implementar esse índice:

- Opção 1: incluir diversas entradas de índice com o mesmo valor K(i), uma para cada registro, sendo um índice denso.
- Opção 2: possuir registros de tamanho variável para as entradas de índices, com campo de repetição para os ponteiros. Mantendo uma lista de ponteiros <P(i,1),...,P(i,k)> na entrada de índices para K(i), ou seja, um ponteiro para cada bloco que contenha um registro cujo valor de campo de indexação seja igual a K(i). Ambas nas opções 1 e 2, o algoritmo de pesquisa binaria no índice deve ser modificado de maneira apropriada.

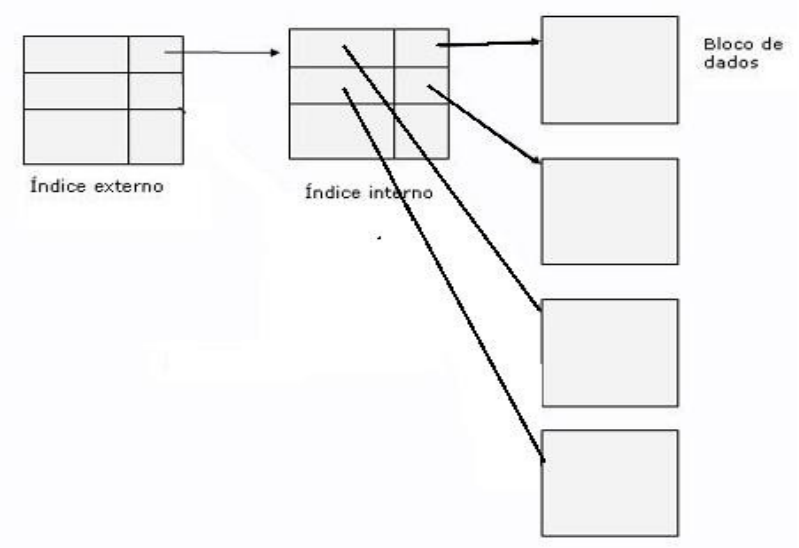

Figura 2.3: Exemplo de aplicação de índice secundário.

- Opção 3: mais utilizada, consiste em manter as próprias entradas de índice num tamanho fixo e possuir uma única entrada para cada valor de campo de indexação criado, porém, cria um nível adicional de acesso indireto para lidar com os diversos ponteiros. Neste esquema não denso o ponteiro P(i) na entrada de índice <K(i), P(i)> aponta para um bloco de ponteiros de registros; cada ponteiro de registro daquele bloco aponta para um dos registros do arquivo de dados com valor K(i) para o campo de indexação. Se ocorrer algum valor K(i) em muitos registros, de forma que seus ponteiros de registros não possam caber em um único bloco de discos, é utilizado um *cluster* ou uma lista de blocos. Esta técnica é apresentada na Figura 2.3. A recuperação através de situações de índices requer um ou mais acessos a blocos devido ao nível adicional, porém os algoritmos para pesquisar um índice e para inserir novos registros no arquivo de dados são diretos. As recuperações em condições complexas de seleção podem ser executadas, através de referencias aos ponteiros de registros, sem necessariamente ter que recuperar muitos registros de arquivos desnecessários.

Capítulo 3

Índices Multinível

É possível construir-se índices multinível a partir de índices de apenas um nível. Por exemplo, um arquivo de índice ordenado de chaves distintas pode ser chamado de primeiro nível, e este apontar para um outro arquivo de índice ordenado. Como o índice do primeiro nível é um índice primário, suas chaves podem ser ponteiros para cada bloco do índice do segundo nível. Esse processo pode ser aplicado indefinidamente[6].

Uma organização comum de arquivos utilizada no processamento de dados comerciais é um arquivo ordenado com um índice principal multinível em seu campo chave de ordenação. Esta organização, de arquivo seqüencial indexada, foi utilizada em grande escala pelos primeiros sistemas IBM[4]. A inserção é manipulada por alguma forma de arquivo de *overflow* que é incorporado periodicamente ao arquivo de dados, sendo, este índice, recriado durante a reorganização de arquivos.

3.1 Índices Dinâmicos Multinível

Um índice multinível reduz o numero de blocos acessados quando se pesquisa um registro, dado seu valor de campo de indexação. Apesar deste beneficio, os processos de inclusão e exclusão, continuam ocasionando problemas no momento da manipulação de dados, uma vez que todos os níveis de índices são arquivos fisicamente ordenados. Para manter os benefícios de utilizar a indexação multinível, reduzindo ao mesmo tempo os problemas relativos à

inclusão e exclusão, a utilização de um índice multinível que deixa algum espaço em cada um de seus blocos para inserir novas entradas torna-se necessário.

Esta estruturação tornou-se conhecida como índice multinível dinâmica. Enquanto índices de um ou dois níveis auxiliam em acelerar o processo de recuperação de informação, existem estruturas de dados mais genéricas que são comumente implementadas em SGBDs comerciais por serem mais flexíveis e escaláveis. A estrutura de dados mais comum para esse propósito é a Arvore-B[1]. A Arvore-B, em essência:

- Automaticamente mantém os níveis balanceados para a quantidade de dados que está sendo indexada, e;
- Gerencia o espaço usado por seus blocos para que ele sempre esteja ocupado com pelo menos a metade de sua capacidade.

3.2 Arvore-B

Arvore-B[2] são árvores de busca desenvolvidas para trabalharem em discos magnéticos ou qualquer outro dispositivo de armazenamento de acesso direto em memória secundária. Em uma aplicação comum de uma Arvore-B, a quantidade de dados é tão grande que provavelmente não caberia na memória principal. A Arvore-B copia blocos específicos para a memória principal quando necessário e os grava no disco se os blocos tiverem sido alterados. A Figura 3.1 exemplifica um modelo de Arvore-B.

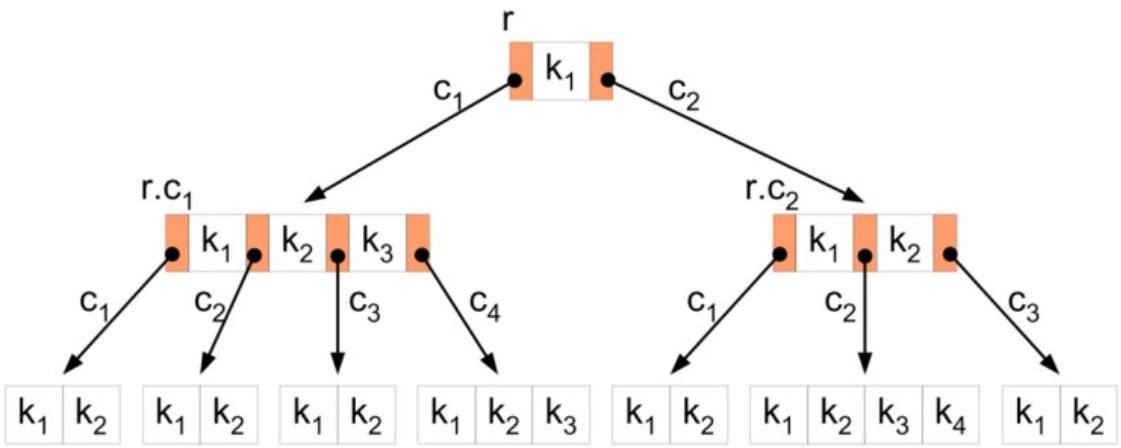

Figura 3.1: Arvore-B.

As Arvore-B são similares as demais arvores de busca otimizadas, mas sua estrutura minimiza operações de entrada e saída. A diferença mais significante entre a Arvore-B e as demais arvores está no fato de que cada nó da Arvore-B pode ter vários filhos, ou seja, o *branching factor* da Arvore-B pode ser bastante grande. Arvore-B apresentam sua altura mais reduzida que as demais arvores de busca, podendo ser usadas para implementar operações com tempo computacional $O(\log n)$, que é considerado um tempo excelente para estas operações numa grande quantidade de arquivos armazenados.

3.2.1 Árvore-B: Definições

A Arvore-B é uma árvore baseada nas seguintes propriedades:

- Cada nó x de uma Arvore-B possui:
  - $n[x]$ chaves, armazenadas em ordem crescente: $chave1[x] \leq chave2[x] \leq ... \leq chaven[x]$.
  - $d[x]$ dados associados[1] às $n[x]$ chaves.
  - Um campo booleano que define se o nó é *folha* ou se é um *nó interno*.
- Se x é um nó interno, então x terá $n + 1$ ponteiros para seus filhos. As folhas não têm filhos.
- Duas chaves adjacentes $chavea[x] \leq chaveb[x]$ em um nó x definem um intervalo onde todas as chaves ki em que $chavea[x] \leq ki \leq chaveb[x]$ se encontrarão na sub-árvore com raiz em x acessível a partir do ponteiro $filhoa+1[x]$.
- Toda folha possui a mesma profundidade, que é a altura da árvore h.
- Cada nó da Arvore-B, por definição, possui restrições quanto à quantidade de chaves.
  - Cada nó, exceto a raiz, precisa ter pelo menos t-1 chaves.
  - Cada nó *interno*, exceto a raiz, precisa ter t ponteiros para seus filhos.

Cada nó possui no máximo $2t$-1 chaves, para que assim cada nó interno tenha no máximo $2t$ filhos.

3.2.2 Características da Arvore-B

[1]

A maioria das operações em uma Arvore-B é proporcional ao número de acessos a disco executados. Apesar de algumas outras arvores, além da Arvore-B terem o crescimento da altura na ordem de ***O*(log *n*)**, a base do logaritmo para a Arvore-B pode ser muito maior. Desta forma, a Arvore-B economiza aproximadamente log *t* em relação a demais arvores otimizadas quanto ao crescimento da altura da árvore.

3.2.2.1 Busca

A busca na Arvore-B é similar a uma busca em uma árvore binária, exceto pelo fato que ao invés de apenas fazer uma decisão binária para qual ramo da árvore prosseguir a cada nó, a decisão passa a ser em relação ao número de chaves que o nó possui. Ou seja, no pior caso, é feito $n[x] + 1$ decisões, sendo $n[x]$ o número de chave em um nó.

O número de acessos a disco realizados pela *busca* é da ordem da altura *h* da árvore, portanto, $O(h) = O(\log_t n)$ onde n é número de chaves na Arvore-B e t é o mínimo de chaves por nó. Lembrando que cada nó comporta no máximo $2t$ chaves, o tempo computacional de busca da Arvore-B é dado por $O(th) = O(t\log_t n)$.

3.2.2.2 Inserção & Splitting

Inserir uma chave na Arvore-B não é tão simples quanto inserir uma chave em uma árvore binária. Quando uma chave deve ser inserida em um nó cheio ($2t$-1 chaves), o nó é dividido pela chave *mediana* em dois nós de t-1 chaves. A chave mediana é, então, promovida para seu *pai* (que deve ser um nó não cheio, senão também deverá ser dividido), para assim identificar o ponto de divisão entre os dois novos nós.

3.3 Arvore-B$^+$

A maioria das implementações de índices multiníveis dinâmicos usa a variante Arvore-B$^+$ da Arvore-B. Na Arvore-B, uma chave somente é entrada uma vez em algum nível da árvore, em conjunto com o dado associado. Já na Arvore-B$^+$, todos os dados só são armazenados nas *folhas*. Desta maneira, a estrutura conceitual das folhas difere da estrutura dos nós internos. As folhas da Arvore-B$^+$ estão ligadas em seqüência, tornando possível o acesso ordenado a seus campos. A Figura 3.2 aprezenta um modelo de Arvore-B$^+$

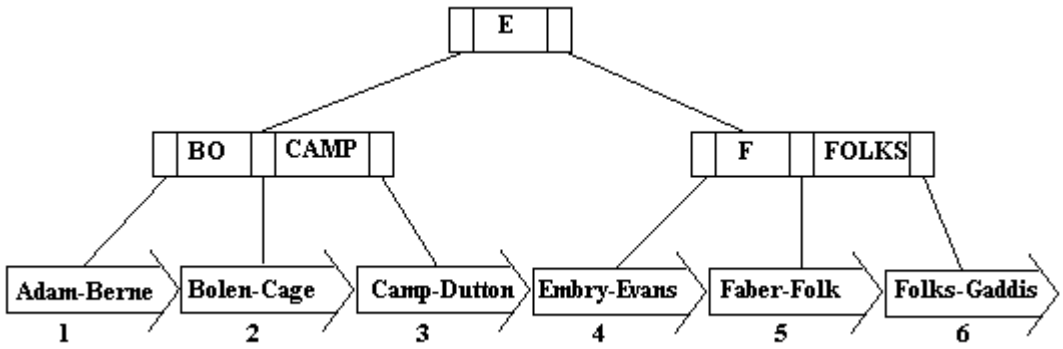

Figura 3.2: Exemplo de Arvore-B$^+$.

3.3.1 Arvore-B+: Definições

Cada nó interno da Arvore-B$^+$ é constituído de:

- $n[x]$ chaves, armazenadas em ordem crescente: $chave1[x] \leq chave2[x] \leq ... \leq chaven[x]$.
- $p[x]$ ponteiros para nós descendentes associados às $n[x]$ chaves.
- Cada nó interno tem, no máximo, $2t$ ponteiros.
- Cada nó interno, exceto a raiz, tem pelo menos t ponteiros. A raiz tem pelo menos 2 ponteiros, caso seja um nó interno.

A estrutura de um nó folha:

- $n[x]$ chaves, armazenadas em ordem crescente: $chave1[x] \leq chave2[x] \leq ... \leq chaven[x]$.
- $d[x]$ dados (ou ponteiros para outra estrutura onde está o dado em si, acrescentando mais um nível de *indireção*) associados às $n[x]$ chaves.
- P, ponteiro para o próximo nó folha.
- Cada folha possui pelo menos t valores.
- Todas as folhas se encontram no mesmo nível de profundidade.

3.3.2 Características da Arvore-B+

Arvore-B$^+$ são mais usadas por SGBDs comerciais por oferecer consultas por intervalos. Se em uma consulta deseja-se obter dados referentes às chaves no intervalo $[a,b]$, deve-se executar uma busca por a até a folha onde a deveria se encontrar (mesmo que a não exista na

árvore). Se esta folha também não contiver *b*, deve-se seguir para a próxima folha através do ponteiro *P*, verificando todas as chaves até que:

- Encontre-se *b*.
- Alcance-se o fim do nó e tenha-se que prosseguir para a próxima folha.

3.3.2.1 Busca

A busca na Arvore-B^+ procede da mesma maneira da inserção na Arvore-B comentada na sessão 3.2.2.1, exceto pelo fato de que o dado na Arvore-B^+ sempre estará em uma folha. Na Arvore-B o pior caso é descer até uma folha para encontrar um dado, uma vez que o dado pode estar em um nó interno. Por tanto, na Arvore-B^+ o número de acessos a disco é sempre igual à altura *h* da árvore.

3.3.2.2 Inserção & Splitting

O princípio básico da inserção na Arvore-B^+ é o mesmo da Arvore-B. Primeiro procura-se o nó, depois se insere a nova chave. Embora o princípio seja o mesmo, o procedimento não é exatamente o mesmo. Na Arvore-$B^{+,}$ deve-se encontrar a *folha* a qual a nova chave pertence. Se houver espaço na folha, a nova chave é inserida. Senão, a folha deve ser dividida (*splitting*) em duas e suas chaves distribuídas entre elas. A divisão de um nó em um determinado nível da árvore implica em uma nova entrada (chave) no nível superior àquele nó. Por praticidade, aplica-se esse procedimento recursivamente caso a nova chave inserida no nível superior (nó pai da folha que foi dividida) também não tenha espaço (causando um novo *split*).

Capítulo 4

Índices em Chaves com Mais de um Atributo

Em muitas solicitações de recuperação e atualização, muitos atributos estão envolvidos, ao contrario dos métodos que utilizam chaves primarias ou secundarias, nas quais os arquivos são acessados em atributos únicos. Caso uma certa combinação de atributos seja utilizada muito frequentemente, é vantajoso montar uma estrutura de acesso eficaz através de um valor chave que seja a combinação de vários atributos.

Considerando um arquivo que contenha vários atributos, sendo pelo menos dois constituíveis de valores inteiros, e não sendo chaves primarias pode ser feita uma consulta com a seguinte forma <item 1, item 2>. A formulação desta pesquisa, atribuindo valores limites a item 1 e item 2, irá gerar um espaço de busca reduzido se comparado a uma busca com um atributo, índice principal ou secundário

4.1 Índices Dinâmicos Baseados em *Hashing*

Uma alternativa à Arvore-B que estende algoritmos de busca e os aplica em busca externa é chamada de *Extensible Hashtable*. Esta estrutura é uma implementação de busca que normalmente requer apenas um ou dois acessos a disco, inclusive para a inserção (na maioria dos casos).[6]

A *Extensible Hashtable* combina características de *hashing*, *multiway-tries* [3][8]e métodos de acesso sequencial. O primeiro passo é definir uma função de *hashing* que transforme a *chave* em um inteiro. Da mesma maneira como se comportam as *multiway-tries*, a *Extensible Hashtable* começa usando apenas os primeiros *bits* do código *hash* da chave para indexar através de uma tabela cujo tamanho é uma potência de 2. Semelhante aos métodos de acesso sequencial, a *Extensible Hashtable* mantém um estrutura que mapeia os códigos *hash* às páginas onde estão armazenados os dados. E as páginas de dados, ou *buckets*, se dividem

em duas quando sua capacidade chega a um limite pré-determinado de maneira similar às Arvore-B.

4.1.1 Definições

É definido, a priori, o tamanho máximo da código *hash*. De acordo com o aumento da quantidade de itens armazenados na *hashtable*, aumenta-se o número de *bits d* utilizados da código *hash*.

A estrutura da *Extensible Hashtable* é bem mais simples que a estrutura da Arvore-B e Arvore-$B^{+.}$ Ela consiste em:

- Uma tabela *dir* com referências para páginas (*buckets* ou *nodes*).
- *Buckets*, que são os repositórios dos dados cujo tamanho máximo é 2*M*.
- *N*, número de itens na tabela.
- *d*, o número de *bits* utilizados do código *hash* no momento.
- *D,* a quantidade de *buckets* da estrutura no momento.
- $D = 2^d$.

4.1.1.1 Busca

Através da função de *hashing* utilizada pela implementação, a chave do item é transformada em um código *hash*, que é um inteiro. A partir dos *d* primeiros *bits* é localizado o *bucket* onde estão todos os itens com aquele mesmo código *hash*, e realizada uma busca sequencial para localizar o item desejado.

4.1.1.2 Inserção e *Splitting*

Para se inserir um item em uma *Extensible Hashtable*, primeiro executa-se uma busca, e então a inserção. O modelo é similar ao seguido pela Arvore-B, exceto pelo algoritmo de busca e *split* utilizados. O método que realiza o *split* cria um novo nó vazio e examina o *k-ézimo bit* de *d*. Se este *bit* for 0, o item permanece no nó antigo. Se for 1, o item é movido para o novo nó. Então, $k + 1$ é atribuído à variável que armazena a quantidade de *bits* utilizados para aquele determinado nó (*k*). Se ainda assim este procedimento não resultar em pelo menos um item

em cada um dos nós envolvidos, o *split* é novamente executado. Ao final, o ponteiro para o novo nó é inserido na tabela *dir*.

O ponto crucial da inserção na *Extensible Hashtable* dá-se no momento em que o ponteiro para o novo nó é inserido na tabela *dir*. Se k for igual a d, apenas é inserido o novo ponteiro na tabela *dir*. Se k for maior que d, então o tamanho de D deve ser dobrado, ou seja, o número máximo de nós gerenciados pela tabela *dir* deve ser dobrado. Neste momento, mais um *bit* do código *hash* passa a ser utilizado para mapear os nós ($d = d + 1$). Este procedimento é realizado até que k volte a ser igual a d.

4.1.2 Características

De forma diferente da Arvore-B, a *Extensible Hashtable* não possui acesso ordenado aos dados armazenados uma vez que seus nós não têm ligações entre si, limitando assim seu campo de aplicação. *Extensible Hashtable* introduz um nível a mais de indireção ao método de *hashing* em memória primária. Contudo, o número de acessos a disco é da ordem de $O(1)$ para a busca caso a tabela *dir* seja mantida em memória primária.

Por outro lado, a *Extensible Hashtable* possui alguns defeitos. Quando se faz necessário dobrar o tamanho da tabela, os nós ficam inacessíveis durante o tempo gasto durante o processo, além de que a nova tabela pode não mais caber na memória principal. Como consequência, o número de acessos a disco aumentaria, denegrindo o desempenho da estrutura.

4.2 Arquivos de Grade

Considerando um arquivo empregado que contenha os atributos NUD (número do departamento), idade, logradouro, cidade, CEP, salário e abilidade com a chave de NSS, além das alternativas já citadas para inclusão no banco de dados utilizando tanto chaves primarias ou secundarias para pesquisas ou inserções ou demais operações, uma outra alternativa é organizar o arquivo empregado como arquivo de grade. Se uma operação necessitar acessar um arquivo por uma chave com dois atributos, por exemplo NUD e idade, pode ser construída uma disposição de grade com uma escala, ou dimensão, linear para cada um dos atributos da pesquisa.

Para realização de um arquivamento de grade neste modelo, considera-se a aplicação de uma escala, tanto para NUD e idade. Esta escala deve ser planejada a fim de atender de forma completa todos os dados possíveis de serem pesquisados, semelhante a uma estruturação *hash*. Baseando-se neste índice, e com os valores a serem pesquisados, relaciona-se em qual celula da grade encontra-se o item solicitado. Cada celula aponta para um endereço de *bucket* onde os registros que correspondem aquela celula estão armazenados. A Figura 4.1 mostra um exemplo da utilização dos conceitos de arquivos de grade e mostra também a designação de celulas para *buckets*.

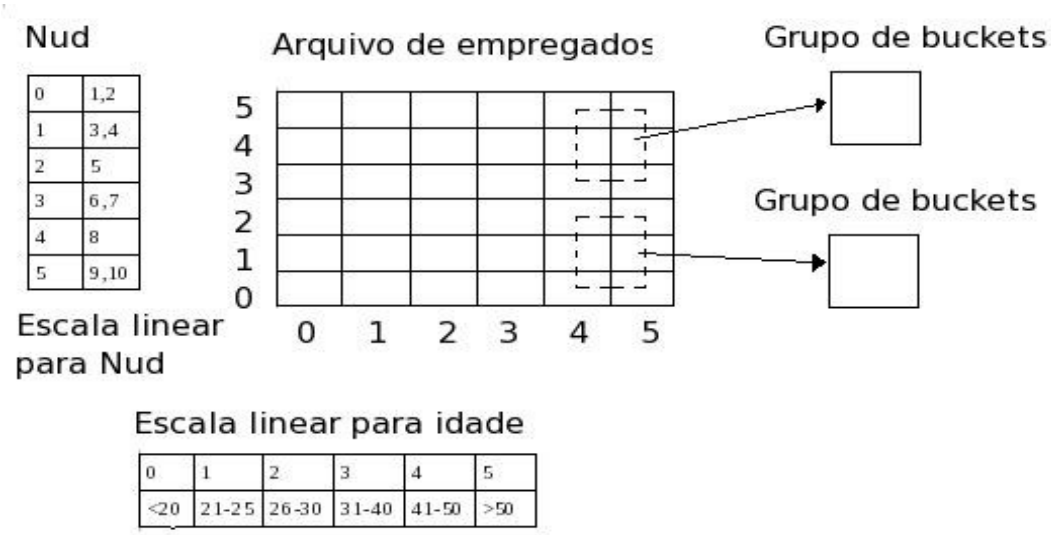

Figura 4.1: Exemplo de *array* de grade nos atributos NUD e idade.

Esse método é particularmente útil para consultas de intervalo que mapeia para um conjunto de celulas correspondentes a um grupo de valores ao longo das escalas lineares. Conceitualmente, o arquivo de grade pode ser aplicado a qualquer numero de chaves de pesquisa. Para *n* chaves de pesquisa, o *array* de grade possuirá *n* dimensões. O *array* de grade permite portanto um particionamento do arquivo ao longo das dimensões dos atributos chaves de pesquisa e fornece um acesso através das combinações de valores ao longo daquelas dimensões. Os arquivos de grade possuem um bom desempenho em termos de redução de tempo para o acesso de múltiplas chaves. Entretanto, eles representam um gasto adicional de espaço em termos da estrutura do *array* de grade, somado ao elevado custo de manutenção.

Capítulo 5

Outros Tipos de Índices - Índices Lógicos

Assumindo as entradas de índice <K(i), P(i)> sempre incluem um ponteiro físico que especifica o endereço do registro físico no disco como um número e *offset* de bloco. Isso é considerado um índice físico e possui a desvantagem de que o ponteiro deve ser alterado se o registro for movido para uma outra localização de disco. Por exemplo, suponha que uma organização principal de arquivos seja baseada no *hashing*, então, cada vez que um *bucket* é dividido, alguns registros são alocados nos novos *buckets* e portanto possuem novos endereços físicos. Caso houvesse um índice secundário no arquivo, os ponteiros para aqueles registros teriam que ser encontrados e atualizados, dificultando a operação.

Para remediar essa situação, podemos utilizar uma estrutura chamada índice lógico, cujas entradas de índice são da forma <K,K_P>. Cada entrada possui um valor K para o campo de indexação combinado com o valor K_P do campo utilizado para a organização principal do arquivo. Ao pesquisar o índice secundário do valor de K, um programa pode localizar o valor correspondente de K_P e fazer uso disso para acessar o registro através da organização principal do arquivo. Índices lógicos introduzem portanto um nível acional de acesso indireto entre a estrutura de acesso e os dados. Eles são utilizados quando endereços de registros físicos supostamente puderem se alterar com frequência. Os custos desse acesso indireto é a pesquisa adicional baseada na organização principal do arquivo.

Capítulo 6

Conclusões

Para atender a uma consulta em que se procura um ou mais registros, primeiramente o sistema acessa o índice, e neste há um ponteiro indicando a localização do bloco ou blocos na base de dados onde o(s) registro(s) se encontra(m). Há várias estruturas de dados que se propõem a resolver esse problema. Os tipos mais comuns de índices são os baseados em arquivos ordenados (índices de um nível), em árvores (índices multi nível) e índices baseados em *hashing*.

Índices baseados em arquivos ordenados usam a mesma ideia dos índices remissivos encontrados em livros, onde é fornecida uma lista de palavras importantes, em ordem alfabética, indicando uma lista de páginas onde há ocorrências daqueles termos. Sem o auxílio do índice remissivo, ter-se-ia que olhar página por página do livro a procura da palavra desejada.

As estruturas de dados que se baseiam nesta abordagem funcionam analogamente: o índice armazena cada valor do campo a ser indexado (*chave candidata* ou *campo indexado*) assim como uma lista de ponteiros para os endereços de todos os blocos do arquivo da base de dados onde aqueles atributos se localizam. O campo indexado se encontra ordenado, podendo assim ser feita uma *busca binária* no índice.

Devido a grande quantidade de dados presente no banco de dados, uma boa estruturação de índices deve ser adotada, tentando minimizar o máximo possível de acessos a dados de forma a encontrar um valor determinado. Decisões equivocadas neste método podem levar o sistema a desperdiçar elevado tempo para processos de busca, inserção e exclusão. Quanto mais

tempo uma dessas atividades necessitar para sua finalização, menos eficiente o banco tornar-se-a. Com isso, torna-se necessário um estudo avançado sobre o funcionamento e dados presente no banco para posterior manipulação do mesmo.

Referências Bibliográficas

[1] Bayer, R., McCreight, E.. “Organization and Maintenance of Large Ordered Indexes”. *Acta Informatica*, 1(3):173-189, Fevereiro, 1972.

[2] Casanova, M. A., Disponivel em:”*www.inf.puc-rio.br/~casanova/INF1731-BD/modulo15.pdf*“ Setembro, 2009.

[3] Frakes,W. B. e Baeza-Yates,R. "Information Retrieval: Data Structure & Algorithms". Prentice Hall, 1992.

[4] IBM. Disponível em:“http://www.ibm.com/br/pt/” Setembro, 2009.

[5] *Navarro*,P.L.K.G. Disponível em:“http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=67” Setembro, 2009.

[6] Neto, M. C. T., “Estrutura de Dados para Indexação de Grande Volume de Dados”, Disponivel em: “http://www.cin.ufpe.br/~tg/2001-1/ “, Setembro, 2009.

[7] Rissoli, V. R. V., “Arquitetura Interna de Banco de Dados”, Disponivel em: *“*www.ucb.br/prg/professores/vandor/aula_5_arquitetura.pps*”, Janeiro, 2005.*

[8] Sedgewick, R., Wyk, J. van , “Algorithms in C++, Part 1-4”. p. 560-564; 646-662. Third Edition. Addison-Wesley, 1998.