Unidade II

5 CONSULTA DE DADOS

O resultado de muitas consultas é uma pequena proporção dos registros. Não é produtivo para o sistema ler cada registro e verificar se o campo ou os campos de condições são válidos. O ideal é que o sistema consiga localizar os registros rapidamente.

5.1 Conceitos básicos

Um índice em arquivo funciona da mesma forma que o índice em um livro. Caso você precise localizar determinado assunto, procurando por uma palavra ou frase, pode buscar o tema no índice, verificar a página em que consta e depois ler as páginas indicadas para encontrar o conteúdo que você procura.

Existem dois tipos básicos de índices:

- índices ordenados, que utilizam uma ordem classificada dos valores;
- índices hash, que utilizam uma distribuição uniforme de valores por um intervalo de buckets (balde) – o bucket usa uma função de hash para distribuir os valores.

O uso de índices pode melhorar significativamente o desempenho do banco de dados. Com isso em mente, primeiro precisamos entender como funcionam os processos internos. Recomenda-se usar índices em colunas altamente seletivas, por exemplo, além de uma chave primária, que muitas vezes pode ser usada como identificador exclusivo de uma entidade em sua aplicação. Também se pode ter um índice em uma coluna para auxiliar nas consultas que têm uma cláusula WHERE se precisarem usar AND, OR ou NOT, que geralmente alteram o desempenho da consulta em determinados casos.

5.2 Índices ordenados

Existem diversos tipos de índices ordenados. Um índice primário é definido no campo de chave de ordenação de um arquivo ordenado de registros. Elmasri e Navathe (2011, p. 425) o definem com detalhes:

> Um índice primário é um arquivo ordenado cujos registros são de tamanho fixo com dois campos, e ele atua como uma estrutura de acesso para procurar e acessar de modo eficiente os registros de dados em um arquivo. O primeiro campo é do mesmo tipo de dado do campo de chave de ordenação – chamado de chave primária – do arquivo de dados e o segundo campo é um ponteiro para um bloco de disco (um endereço de bloco). Existe uma entrada de índice (ou registro de índice) no arquivo de

índice para cada bloco no arquivo de dados. Cada entrada de índice tem o valor do campo chave primária para o primeiro registro em um bloco e um ponteiro para esse bloco como seus dois valores de campo.

A figura 10 mostra a criação de um índice primário no arquivo ordenado. O atributo Nome é utilizado como chave primária (considerando que o valor de Nome seja único, isto é, que não haja valores duplicados). As três primeiras entradas de índice têm um valor Nome e um ponteiro P:

<K(1) = (Aaron, Eduardo), P(1) = endereço de bloco 1>

<K(2) = (Adams, João), P(2) = endereço de bloco 2>

<K(3) = (Alexandre, Eduardo), P(3) = endereço de bloco 3>

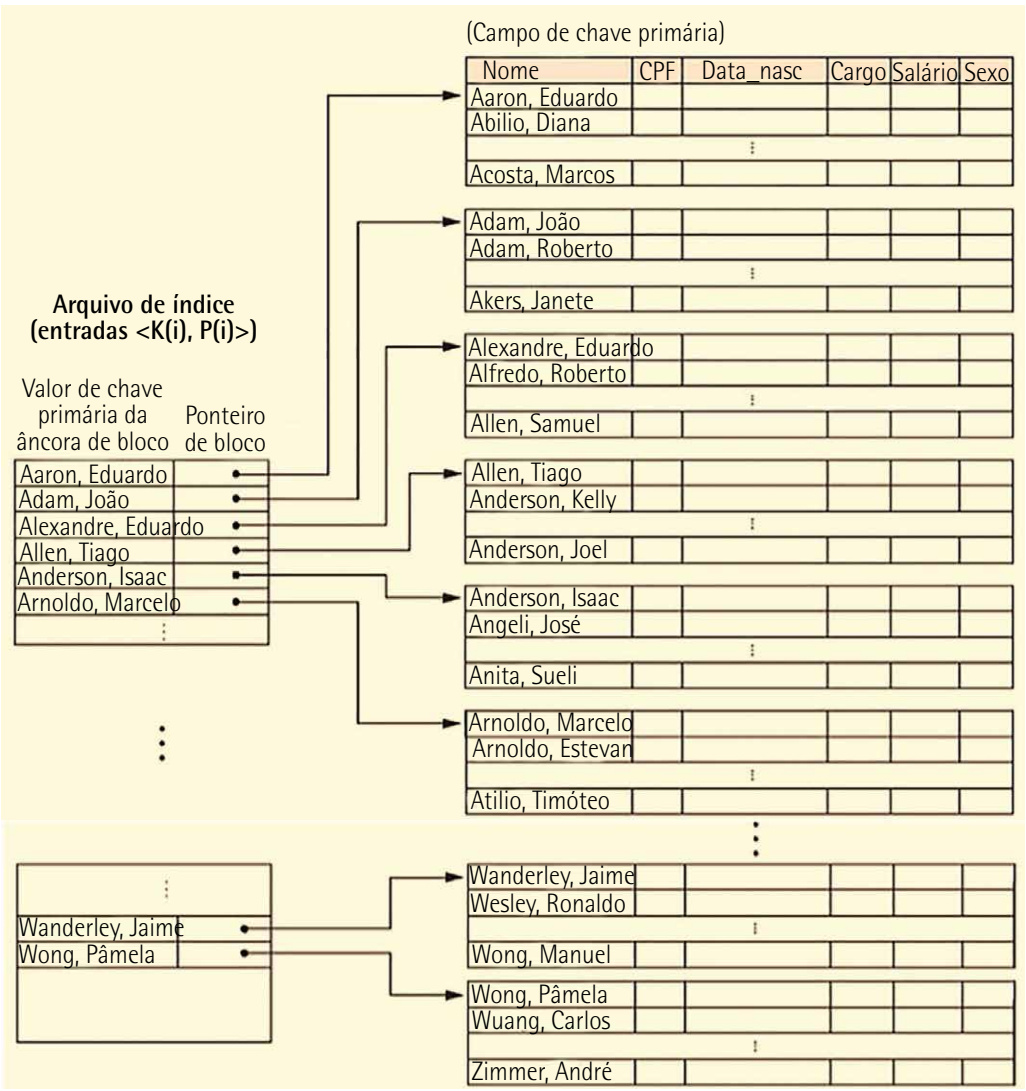

Figura 10 – Índice primário no campo de chave de ordenação do arquivo

Fonte: Elmasri e Navathe (2011, p. 426).

Os índices ordenados podem ser classificados como densos ou esparsos. Um índice denso tem uma entrada de índice para cada valor de chave no arquivo de dados. Já o esparso tem entradas de índice considerando apenas alguns valores de pesquisa, tendo, assim, menos entradas do que o registro de arquivos, como o índice primário.

Quando criamos uma tabela, geralmente não declaramos NONCLUSTERED em uma chave primária, que assume o valor de CLUSTERED, criando automaticamente os índices agrupados. Cada tabela deverá ter um e somente um índice de agrupamento (uma primary key). Quando criamos um índice de agrupamento, será necessário espaço em disco (aproximadamente 1,2 vez o tamanho atual da tabela). Depois desta operação, o espaço em disco é restaurado automaticamente.

Os índices sem agrupamento ou secundários são utilizados quando os usuários necessitam de diferentes maneiras de consultar dados. Por exemplo, um leitor precisa consultar livros sobre animais e pesquisar utilizando o nome popular e o nome científico dos animais. O programador poderá criar um índice sem agrupamento para recuperar os nomes comuns e um índice de agrupamento para recuperar os nomes científicos.

Saiba mais

Elmasri e Navathe (2011) apresentam os conceitos essenciais para a implementação de sistemas de banco de dados:

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. 6. ed. São Paulo: Pearson, 2011.

5.3 Arquivos de índice de árvore B+

Na ciência da computação, uma árvore B+ é uma estrutura de dados do tipo árvore. Essa estrutura de índice é a mais utilizada, pois mantém sua eficiência independentemente da inserção e exclusão dos dados. É ainda balanceada: o caminho de cada raiz até uma folha é o mesmo – essa propriedade assegura a boa performance para pesquisa, inserção e exclusão.

A figura 11 apresenta uma árvore B+ completa para o arquivo de instrutor de uma universidade (com n = 4, isto é, um nó não folha pode manter até n ponteiros e precisa manter pelo menos n/2 ponteiros). Os ponteiros nulos foram omitidos para facilitar o entendimento da figura; considere que todo campo de ponteiro que não possui uma seta deve ser interpretado como se tivesse um valor nulo.

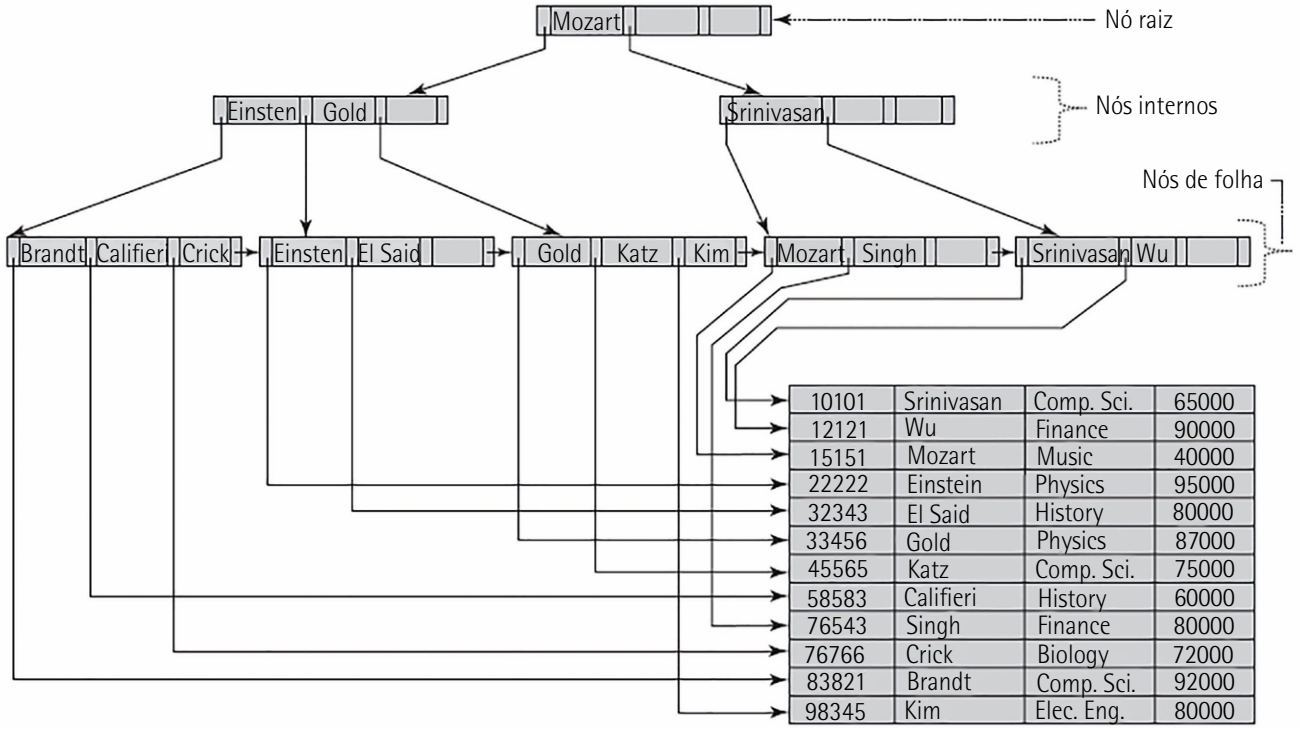

Figura 11 – Árvore B+ para o arquivo de instrutor (n = 4)

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 358).

5.3.1 Consulta em árvore B+

O pseudocódigo a seguir indica a função de busca (v), presumindo que não há duplicatas, ou seja, que há no máximo um registro com determinada chave de busca. A função começa na raiz da árvore e se move para baixo até atingir o nó folha que contém o valor especificado, se esse valor existir na árvore. Especificamente, começando da raiz como o nó atual, a função repete as etapas a seguir até que um nó folha seja alcançado (SILBERSCHATZ; KORTH; SUDARSHAN, 2020, p. 360).

```
function find(v)
/* Assumindo que não há duplicatas, retorna o ponteiro para o registro com o
 * valor de chave de busca v se tal registro existir; se não, retorna nulo */
    Defina C = nó raiz
    while (C não é um nó de folha) begin
        Seja i = menor número tal que v ≤ C · K_i
        if esse número i não existir then begin
            Seja P_m = último ponteiro não nulo no nó
            Atribua C = C · P_m
        end
        else if (v = C · K_i) then Defina C = C · P_{i+1}
        else Defina C = C · P_i /* v < C · K_i */
    end
    /* C é o nó de folha */
    if para algum i, K_i = v
        then return P_i
        else return null; /* Não existe um registro com valor de chave v*/
```

Primeiro, o nó atual é pesquisado procurando o menor i, tal que o valor da chave de pesquisa Ki seja maior ou igual a v. Suponha que um valor seja encontrado; então, se Ki é igual a v, o nó atual se torna o nó indicado por Pi 1; caso contrário, Ki > v e o nó atual se torna o nó indicado por Pi. Se Ki não for encontrado, então obviamente v > Km–1, em que Pm é o último ponteiro diferente de zero para o nó. Neste caso, o nó atual é o indicado por Pm. Esse procedimento itera na árvore até que um nó folha seja alcançado.

Se o nó folha tem o valor da chave de busca Ki = v, o ponteiro Pi nos aponta para a entrada com o valor da chave de busca Ki. A função então retorna um ponteiro para o registrador Pi. Se uma chave de pesquisa com valor v não for encontrada em um nó folha, não há registro na relação com o valor da chave v, e a função de pesquisa retornará nula para indicar falha.

5.3.2 Inserção em árvore B+

O próximo pseudocódigo mostra o algoritmo de inserção (SILBERSCHATZ; KORTH; SUDARSHAN, 2020, p. 363). Essa função insere um par (valor de chave, ponteiro) em um índice usando dois conjuntos de funções de inserção: insert_in_leaf e insert_in_parent. No pseudocódigo, L, N, P e T representam referências a nós, e L é usado para representar um nó folha. L · Ki e L · Pi representam o valor i e o ponteiro i no nó L, respectivamente; T · Ki e T · Pi são usados da mesma maneira. O pseudocódigo também usa a função parent(N) para encontrar o pai do nó N. Podemos calcular uma lista de nós do caminho para as folhas, primeiro encontrando o nó folha e, em seguida, usando esse caminho para encontrar o pai de qualquer nó.

```
procedure insert (valor K, ponteiro P)
    if (árvore está vazia) crie um nó de folha L vazio, que também é a raiz
    else Encontre o nó de folha L que deve conter o valor de chave K
    if (L tem menos que n – 1 valores de chave)
        then insert_in_leaf (L, K, P)
        else begin /* L já tem n – 1 valores de chave, divida-o */
            Crie o nó L'
            Copie L·P1 ... L.Kn–1 para um bloco de memória T que possa
                armazenar n pares (ponteiro, valor-chave)
            insert_in_leaf (T, K, P)
            Atribua L'·Pn = L·Pn; Atribua L·Pn = L'
            Apague L·P1 até L.Kn–1 de L
            Copie T·P1 até T·K⌊n/2⌋ de T em L começando em L·P1
            Copie T·K⌊n/2⌋+1 até T·Kn de T em L' começando em L'·P1
            Seja K' o menor valor de chave em L'
            insert_in_parent(L, K', L')
    end
procedure insert_in_leaf (nó L, valor K, ponteiro P)
    if (K < L·K1)
        then insira P, K em L imediatamente antes de L·P1
        else begin
            Seja Ki o valor mais alto em L menor ou igual a K
            Insira P, K em L imediatamente após L·Ki
        end
procedure insert_in_parent(nó N, valor K', nó K')
    if (N é a raiz da árvore)
        then begin
            Crie um novo nó R contendo N, K', N'    /* N e N' são ponteiros*/
            Torne R a raiz da árvore
            return
        end
    Seja P = parent (N)
    if (P tem menos de n ponteiros)
        then insira (K', N') em P imediatamente após N
        else begin /* Divida P */
            Copie P para um bloco de memória T que possa armazenar P e (K', N')
            Insira (K', N') em T imediatamente após N
            Apague todas as entradas de P; Crie o nó P'
            Copie T·P1 ... T·P⌊(n+1)/2⌋ para P
            Defina K'' = T·K⌊(n+1)/2⌋
            Copie T·P⌊(n+1)/2⌋+1 ... T·Pn+1 para P'
            insert_in_parent(P, X'', P')
    end
```

O procedimento insert_in_parent recebe os parâmetros N, K', N', em que o nó N é dividido em N e N', e K' é o menor valor de N'. O procedimento altera N pais para registrar a distribuição. Os procedimentos insert_in_leaf e insert_in_parent utilizam uma área de memória temporária T para armazenar o conteúdo

do nó compartilhado, podendo ser modificados para copiar dados diretamente do nó compartilhado para o novo nó, o que reduz o tempo necessário para copiar os dados. No entanto, o uso de um estado temporário T simplifica os procedimentos.

5.3.3 Exclusão em árvore B+

Veja a seguir o pseudocódigo para deletar uma entrada em árvore B+ (SILBERSCHATZ; KORTH; SUDARSHAN, 2020, p. 366).

```
procedure delete(valor K, ponteiro P)
    encontre o nó de folha L que contém (K, P)
    delete_entry(L, K, P)

procedure delete_entry(nó N, valor K, ponteiro P)
    delete (K, P) de N
    if (N é a raiz and N só tem um filho restante)
    then torne o filho de N a nova raiz da árvore e exclua N
    else if (N tem número insuficiente de valores/ponteiros) then begin
        Seja N' o filho anterior ou seguinte de parent(N)
        Seja K' o valor entre os ponteiros N e N' em parent(N)
        if (as entradas em N e N' couberem em um único nó)
            then begin  /* Una os nós */
                if (N é antecessor de N') then swap_variables(N, N')
                if (N não é uma folha)
                    then anexe K' e todos os ponteiros e valores existentes em N a N'
                    else anexe todos os pares (K_i, P_i) existentes em N a N'; atribua N' · P_n = N · P_n
                delete_entry(parent(N), K', N); exclua o nó N
            end
        else begin /* A redistribuição toma emprestada uma entrada de N' */
            if (N' é antecessor de N) then begin
                if (N é um nó não folha) then begin
                    seja m tal que N'.P_m seja o último ponteiro em N'
                    remova (N' · K_m-1, N' · P_m) de N'
                    insira (N' · P_m, K') como o primeiro ponteiro e valor em N,
                        deslocando outros ponteiros e valores para a direita
                    substitua K' em parent(N) por N' · K_m-1
                end
                else begin
                    seja m tal que (N' · P_m, N' · K_m) seja o último par de ponteiro/valor em N'
                    remova (N' · P_m, N' · K_m) de N'
                    insira (N' · P_m, N' · K_m) como o primeiro ponteiro e valor em N,
                    deslocando outros ponteiros e valores para a direita
                    substitua K' em parent(N) por N' · K_m
                end
            end
            else ... simétrico ao caso then ...
        end
    end
```

O procedimento swap_variables(N, N') simplesmente troca os valores das variáveis (ponteiros) N e N'; esta mudança não muda a própria árvore. O pseudocódigo usa a condição "número insuficiente de valores/indicadores". Para nós não folha, este critério significa menos de [n/2] ponteiros; para nós folha, isso significa valores menores que [(n/1)/2]. O pseudocódigo redistribui registros tomando emprestado

um registro de um nó vizinho. Também podemos redistribuir entradas dividindo-as igualmente entre dois nós. O pseudocódigo refere-se à exclusão do registro (K, P) do nó. Para nós folha, o registro realmente vem antes do valor da chave, então o ponteiro P precede o valor da chave K. Em nós não folha, P vem depois do valor da chave K.

5.4 Arquivos de índice de árvore B

Na ciência da computação, uma árvore B é uma estrutura de dados semelhante a uma árvore balanceada que armazena dados ordenados e permite a pesquisa. Uma árvore B é adequada para sistemas de armazenamento que leem e gravam blocos de dados relativamente grandes. A ideia básica é trabalhar com dispositivos de armazenamento secundários; quanto menos espaço em disco uma estrutura de dados fornecer, melhor será o desempenho do sistema na recuperação de dados manipulados.

Um índice de árvore B é semelhante a um índice de árvore B+. A principal diferença entre as duas técnicas é que a árvore B elimina o armazenamento extra de valores de chave de pesquisa. Na árvore B+ da figura 11, as chaves de busca Einstein, Gold, Mozart e Srinivasan aparecem em nós não folha, bem como em nós folha. O valor de cada chave de pesquisa aparece em algum nó de página; diversos são replicados em nós não folha.

A árvore B permite que os valores da chave de pesquisa sejam exibidos apenas uma vez (se forem exclusivos), diferentemente da árvore B+, em que um valor pode aparecer tanto em um nó fora da página quanto em um nó na página. A figura 12 mostra uma árvore B que representa as mesmas chaves de pesquisa que a árvore B+ da figura 11. Como as chaves de busca não são repetidas na árvore B, podemos armazenar o índice em menos nós do que o índice da árvore B+ correspondente. No entanto, como as chaves de pesquisa que aparecem em nós que não são de página não aparecem em nenhum outro lugar da árvore B, somos forçados a adicionar um campo de ponteiro à chave de pesquisa de cada nó que não é de página. Essas referências adicionais apontam para arquivos ou grupos associados à chave de pesquisa.

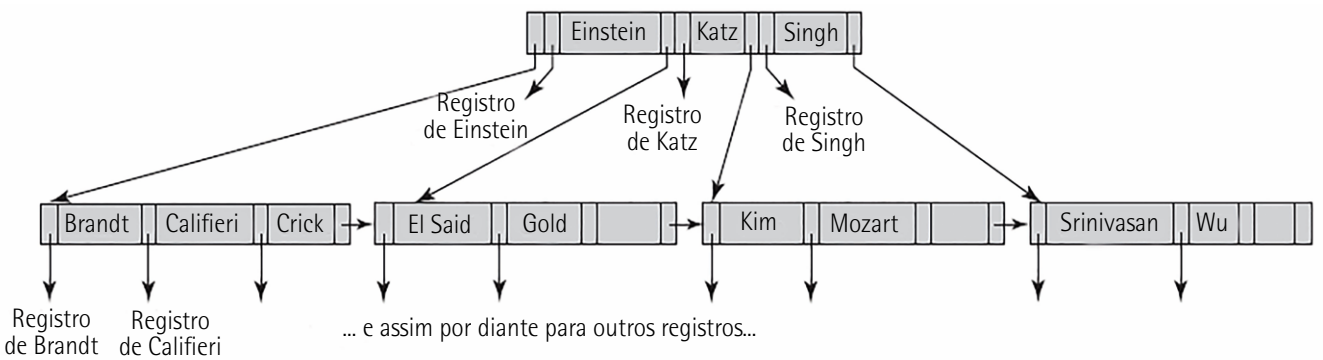

Figura 12 – Árvore B equivalente à árvore B+ da figura 11

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 370).

Observação

Muitos manuais, artigos e profissionais de sistemas de banco de dados usam o termo árvore B para se referir à estrutura de dados que chamamos de árvore B+. Na verdade, seria justo dizer que o termo árvore B agora é considerado sinônimo de árvore B+. No entanto, para evitar confusão entre as duas estruturas de dados, convém utilizar os termos corretamente.

5.5 Hashing estático

Um diagrama de hashing estático é mostrado na figura 13. As páginas que contêm dados podem ser observadas como uma coleção de buckets, com uma página primária e páginas de overflow adicionadas. O arquivo consiste em 0 até N-1 buckets, e inicialmente uma página principal por bucket.

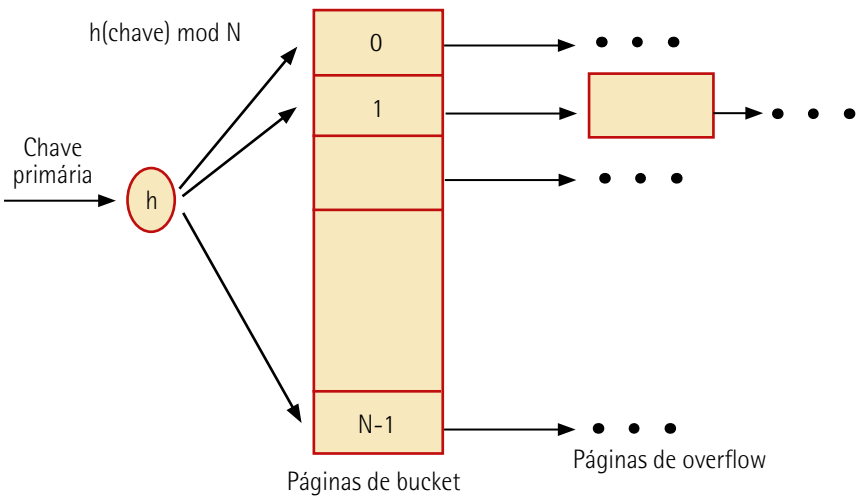

Figura 13 – Hashing estático

Fonte: Ramakrishnan e Gehrke (2008, p. 335).

Para consultar uma entrada de dados, utilizamos uma função hash h para localizar o bucket ao qual ela pertence, e em seguida pesquisamos este bucket. Para melhorar a consulta de um bucket, podemos utilizar entradas de dados ordenadas pelo valor da chave de pesquisa.

Lembrete

Função hash é um algoritmo que transforma dados de comprimento variável em dados de comprimento fixo.

Para inserir uma entrada de dados, utilizamos a função hash para localizar o bucket certo e depois adicionamos a entrada. Caso o espaço não seja suficiente, alocamos uma nova página de overflow, adicionamos a entrada de dados à nova página e a página à cadeia de overflow do bucket.

Para excluir uma entrada de dados, utilizamos a função de hashing para localizar o bucket correto, identificamos a entrada, consultando o bucket, e a removemos. Caso essa entrada de dados seja a última em uma página de overflow, esta página é removida da cadeia de overflow do bucket e inserida em uma lista de páginas livres.

A função hash é muito importante para a abordagem hashing. Ela deve distribuir valores uniformemente no domínio do campo de pesquisa sobre uma coleção de buckets. Se tivermos N buckets, numerados de 0 até N-1, uma função hash h da forma h(valor) = (a * valor + b) funciona bem na prática (o bucket identificado é h(valor) mod N). As constantes a e b podem ser selecionadas para corrigir a função hash. Consultando todas as páginas na sua cadeia de overflow, é fácil prever como o desempenho pode se deteriorar. Ao manter primeiro 80% das páginas cheias, podemos evitar páginas de overflow caso o arquivo não cresça muito, mas geralmente a única maneira de se libertar de cadeias de overflow é criando um novo arquivo com mais buckets.

O maior problema com o hashing estático é que o número de buckets é fixo. Caso um arquivo diminua bastante, teremos muito espaço inútil. Mas se um arquivo aumentar bastante, longas cadeias de overflow se desenvolvem, resultando em baixo desempenho.

5.6 Hashing dinâmico

A maioria dos bancos de dados utilizados cresce muito ao longo do tempo. Se pretendemos usar hashing estático nesses bancos, temos três opções:

- escolher uma função hash baseada no tamanho atual do arquivo, o que diminui o desempenho à medida que o banco de dados cresce;
- escolher uma função de hash com base no tamanho esperado do arquivo no futuro, o que, embora evite a degradação do desempenho, pode desperdiçar uma quantidade significativa de espaço inicialmente;
- reorganizar periodicamente a estrutura de hash à medida que o arquivo cresce, o que envolve escolher uma nova função de hash, recalculá-la para cada registro no arquivo e criar definições de grupo – essa é uma operação em larga escala e demorada, e demanda negar o acesso ao arquivo durante a reorganização.

Várias técnicas de hash dinâmico permitem que a função de hash seja alterada agilmente para acomodar a expansão ou contração do banco de dados.

Saiba mais

Sugerimos a leitura dos artigos que introduziram dois tipos de hashing dinâmico – para hashing extensível, Fagin *et al.* (1979), e para hashing linear, Litwin (1978, 1980):

FAGIN, R. *et al.* Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, Nova York, v. 4, n. 3, p. 315-344, 1979. Disponível em: https://bit.ly/3yPSAGG. Acesso em: 18 out. 2022.

LITWIN, W. Linear hashing: a new tool for file and table addressing. *In:* INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 6., 1980, Montreal. *Proceedings* [...]. [*S.l.*]: VLDB, 1980. p. 212-223.

LITWIN, W. Virtual hashing: a dynamically changing hashing. *In:* INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 4., 1978, Berlim. *Proceedings* [...]. [*S.l.*]: VLDB, 1978. p. 517-523.

5.6.1 Hashing extensível

Primeiramente, é preciso analisar alguns pontos de um arquivo de hash estático para entender melhor o hash extensível. Caso seja necessário adicionar uma entrada de dados em um bucket carregado, deve-se inserir uma página de overflow. Caso não se deseje adicionar mais páginas de overflow, será necessário reorganizar o arquivo neste ponto, dobrando o número de buckets e redistribuindo as entradas no novo conjunto de buckets. Todo o arquivo precisará ser lido e duas vezes mais páginas terão que ser gravadas para se conseguir a reorganização, causando assim uma deficiência. Este problema pode ser resolvido de forma simples: podemos usar um diretório de ponteiros para buckets e dobrar o número de buckets, duplicando assim apenas o diretório e dividindo apenas o bucket que sofreu o overflow.

A figura 14 exemplifica um arquivo de hashing extensível. O diretório é composto de uma matriz de tamanho 4, com cada elemento sendo um ponteiro para um bucket. Para localizar uma entrada de dados, aplicamos uma função hash ao campo de pesquisa e pegamos os dois últimos bits da sua representação binária para obter um número entre 0 e 3. O ponteiro nesta posição da matriz nos dá o bucket desejado, supondo que cada bucket possa guardar quatro entradas de dados. Portanto, para localizar uma entrada de dados com valor hash igual a 5 (binário 101), examinamos o elemento 1 do diretório e seguimos o ponteiro até a página de dados (bucket B na figura).

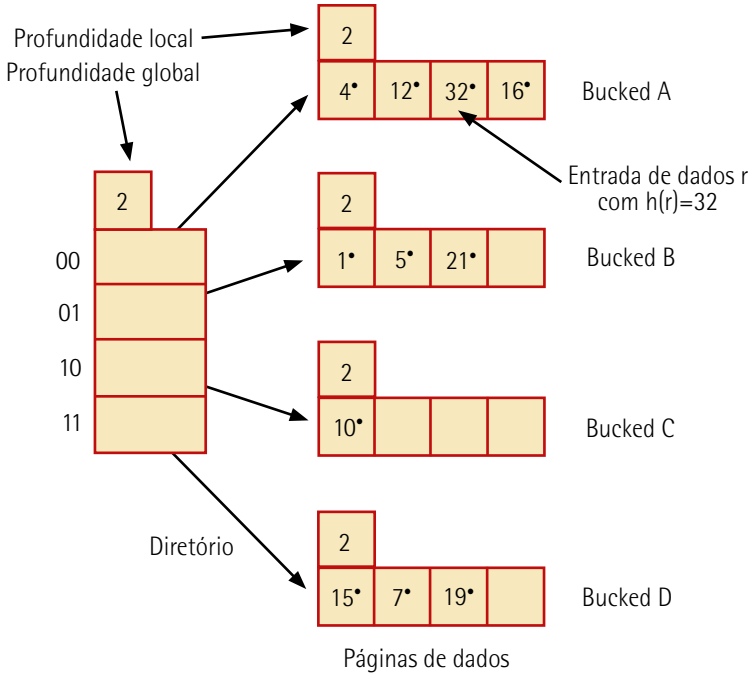

Figura 14 – Exemplo de um arquivo de hashing extensível

Fonte: Ramakrishnan e Gehrke (2008, p. 336).

5.7 Comparação de indexação ordenada e hashing

Conseguimos estruturar arquivos de registros como arquivos ordenados utilizando a organização sequencial indexada ou organização por árvore B+. Como alternativa, conseguimos estruturar os arquivos utilizando o hashing. Por último, conseguimos também estruturá-los como arquivos de heap, em que os registros não estão ordenados de qualquer forma em particular.

Cada esquema tem suas vantagens em determinadas situações. Um implementador de sistema de banco de dados pode fornecer muitos esquemas, permitindo que a decisão final sobre quais utilizar seja do projetista. Essa técnica necessita que o implementador escreva mais códigos, o que aumenta o custo e o espaço do sistema.

Os sistemas de banco de dados suportam árvores B+ para indexação de dados fundamentados em disco. Contudo, a maioria dos bancos de dados não suporta matrizes de arquivos hash ou índices de hash para dados baseados em disco. Um dos motivos é que muitos aplicativos se beneficiam do suporte para consultas de intervalo. Outra razão é que os índices de árvore B+ lidam com o aumento do tamanho da relação de maneira controlada por meio da distribuição de vários nós, cada divisão com baixo custo, ao contrário do hash extensível, que requer o custo relativamente alto de copiar tabelas de endereços de bucket. Ainda outro motivo é que, ao contrário dos índices de hash, as árvores B+ fornecem bons limites de pior caso para operações de exclusão com chaves duplicadas.

Os índices de hash são utilizados na indexação na memória, se consultas de intervalo não forem frequentes. Eles são geralmente usados para criar índices temporários na memória durante o processamento de operações de junção, utilizando a técnica de junção hash.

6 OTIMIZAÇÃO DA CONSULTA

Otimização da consulta é o método de selecionar o melhor plano de avaliação para pesquisas mais eficientes, considerando as diversas estratégias geralmente possíveis para o processamento de uma consulta, especialmente se ela for complexa. Não queremos que os usuários digitem suas buscas para processá-las com eficiência. Entretanto, contamos com o sistema para produzir um plano de avaliação da pesquisa que minimize seu custo. Nesse momento é que a otimização de consultas se torna necessária.

Uma característica da otimização acontece no nível da álgebra relacional, onde o sistema tenta identificar uma expressão que seja equivalente a outra expressão dada, mas cuja execução seja mais eficiente. Outra característica é a escolha de uma estratégia detalhada para processar a consulta, como o algoritmo e os índices específicos a serem utilizados.

A diferença de custo entre uma boa e uma má estratégia (considerando o tempo de avaliação) costuma ser significativa e pode atingir várias ordens de grandeza. Portanto, o sistema deve gastar muito tempo para escolher uma boa estratégia para lidar com a consulta, mesmo que seja realizada apenas uma vez.

6.1 Processamento da consulta

O processamento de consultas refere-se ao conjunto de operações envolvidas na recuperação de dados de um banco. Os recursos incluem a tradução de consultas em linguagens de banco de dados de alto nível em expressões que podem ser usadas no nível físico do sistema de arquivos, além de transformações de otimização de pesquisa e avaliação de consultas em tempo real.

As fases no processamento de uma consulta são:

- análise e tradução;
- otimização;
- avaliação.

Antes de processar a solicitação, o sistema deve traduzi-la a um formato útil. A linguagem SQL é boa para uso humano, mas não para consulta no sistema. Uma representação interna mais útil é aquela baseada em álgebra relacional.

Então o primeiro passo do sistema é traduzir a solicitação em seu formato interno. Este processo de tradução é semelhante à tarefa de um analisador de compilador. Ao criar a forma interna da consulta, o analisador verifica a sintaxe da pesquisa, averigua se os nomes associados que aparecem são nomes

associados no banco de dados, e assim segue. O sistema cria uma representação em árvore analítica da consulta, que por sua vez é convertida em expressões de álgebra relacional. Se a consulta foi expressa como uma view, a etapa de compilação também substitui todas as funcionalidades da view pela expressão de álgebra de relação que a define.

Saiba mais

Para conhecer mais sobre compiladores, recomendamos a leitura do livro *Compiladores: princípios e práticas*, de Louden:

LOUDEN, K. C. *Compiladores*: princípios e práticas. São Paulo: Cengage Learning, 2004.

Depois que uma pesquisa é realizada, geralmente há vários métodos para calcular a resposta. Por exemplo, uma consulta pode ser expressa de diferentes formas no SQL, e pode ser traduzida em expressões de álgebra relacional de várias maneiras. Além disso, a representação algébrica relacional de uma seleção determina apenas parcialmente como a seleção é avaliada; geralmente há várias maneiras de avaliar expressões de álgebra de relação. Por exemplo, considere o comando SELECT:

```
SELECT salario
FROM Funcionario
WHERE salario < 7000
```

Podemos traduzir essa consulta em uma das seguintes expressões da álgebra relacional:

$\sigma_{salario < 7000}(\pi_{salario}(funcionario))$

$\pi_{salario}(\sigma_{salario < 7000}(funcionario))$

Além disso, podemos realizar qualquer operação de álgebra relacional com um dos vários algoritmos diferentes. Para implementar o exemplo anterior, podemos procurar em cada tupla da relação funcionario para encontrar tuplas com salario inferior a 7 mil. Se o atributo salario tiver um índice de árvore B+, podemos usar o índice em vez de pesquisar a tupla.

Para descrever completamente a avaliação de uma consulta, devemos não apenas fornecer uma expressão algébrica relacional, mas também dar instruções que especifiquem a avaliação de cada operação. As instruções podem indicar o(s) índice(s) ou o algoritmo usado em uma função específica.

As operações de álgebra relacional com instruções de avaliação são chamadas de primitivas de avaliação. Um conjunto de operações primitivas que podem ser usadas para avaliar uma consulta é um plano de execução de consulta ou um plano de avaliação de consulta. A figura 15 apresenta o plano

de avaliação para nosso exemplo, em que uma operação de seleção recebe um índice específico (índice 1 na figura). O mecanismo de execução da consulta pega o plano de avaliação, executa-o e retorna as respostas da consulta.

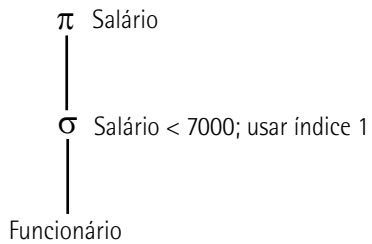

Figura 15 – Um plano de avaliação de consulta

Diferentes planos de avaliação para determinada seleção podem ter custos diferentes. Não esperamos que os usuários escrevam suas seleções de forma a propor o mais eficiente plano – essa é a tarefa do sistema, chamada de otimização de consulta. A consulta será avaliada de acordo com o plano escolhido, e então seu resultado será gerado.

Observação

Os conceitos descritos neste livro-texto para o processamento da consulta nos bancos de dados são representativos; nem todos os bancos seguem à risca a sequência de etapas. Alguns utilizam uma representação de árvore sintática com base na estrutura da SQL em vez da álgebra relacional.

O otimizador de consulta deve saber o custo de cada operação. Ainda que o valor exato seja difícil de calcular porque depende de muitos parâmetros – como a memória real disponível para a operação –, podemos estimar aproximadamente o custo de implementação de cada uma delas.

6.2 Medidas de custo da consulta

Há vários planos de avaliação possíveis, sendo importante comparar as opções de acordo com os custos previstos e escolher a melhor. Para isso, precisamos calcular os custos de cada atividade individualmente e adicioná-los aos do plano de avaliação da consulta, os quais podem ser estimados analisando vários recursos diferentes, como o uso de disco, o tempo de CPU necessário para executar a consulta e os custos de comunicação em um sistema de banco de dados distribuído ou paralelo.

Nos sistemas de banco de dados grandes alocados em disco magnético, os custos de E/S de disco geralmente são maiores que qualquer outro. Dessa forma, os primeiros modelos de custo se concentravam nos valores de E/S ao estimar os custos das operações de consulta. Como o armazenamento em memória flash se expandiu e se tornou mais barato, hoje a maioria dos dados de uma organização pode ser armazenada em unidades SSD (do inglês *solid-state drive*) de maneira mais econômica. O tamanho da memória principal aumentou significativamente e seu custo diminuiu o suficiente nos últimos anos

para que muitas organizações possam armazenar ali seus dados de negócios, mesmo que precisem utilizar memória magnética ou memória flash para assegurar a persistência.

Se os dados estiverem na memória ou em SSDs, os valores de E/S não afetam o custo total; portanto, precisamos considerar o custo da CPU ao calcular a avaliação da consulta. Não incluímos custos de CPU em nosso modelo para simplificar os exemplos, mas observe que podem ser calculados utilizando estimativas simples, como:

- custo de CPU por tupla;
- custo de CPU para processar cada entrada de índice (além do custo de E/S);
- custo de CPU por operador ou função (como operadores aritméticos, operadores de comparação e funções relacionadas).

O banco de dados possui valores determinados para cada custo, que são multiplicados pelo número de tuplas processadas, pelo número de itens de índice processados e pelo número de operadores e operações realizadas.

Ao avaliar o custo de um plano de implementação, usamos o número de transferências em bloco do local de armazenamento e o número de ocorrências de E/S aleatórias – cada um exigindo uma busca em disco do meio de armazenamento magnético – como dois fatores importantes. Se um subsistema de disco leva uma média de tT segundos para mover um bloco de dados e tem um tempo médio de acesso ao bloco (tempo de busca do disco mais latência rotacional) de tS segundos, então uma função que move b blocos e executa S ocorrências de E/S aleatórias utilizará $b \times tT + S \times tS$ segundos.

Observação

Os valores típicos para os discos magnéticos de alto nível, em 2018, considerando um tamanho de bloco de 4 kB e uma taxa de transferência de 40 MB por segundo, seriam:

tS = 4 milissegundos

tT = 0,1 milissegundo

Os valores de tT e tS precisam sempre ser calibrados conforme o sistema de disco utilizado.

A leitura de dados já na memória principal é feita em unidade de linhas de cache em vez de blocos de disco. No entanto, supondo que blocos inteiros de dados sejam lidos, o tempo de transferência tT de um bloco de 4 kB é menor que um microssegundo para dados na memória. A latência tS para buscar dados na memória é inferior a 100 nanossegundos.

Como diferentes dispositivos de armazenamento têm velocidades diferentes, os sistemas de banco de dados devem realizar pesquisas e testes de transferência de blocos para estimar tS e tT para sistemas/dispositivos de armazenamento específicos como parte do processo de instalação do software. Geralmente, os bancos de dados que não inferem esses números possibilitam que os usuários os definam como parte de seus arquivos de configuração.

As estimativas que apresentaremos não consideram o custo de gravar o resultado da operação de volta no disco. Eles serão considerados separadamente, se necessário. O custo de todos os algoritmos que consideramos depende do tamanho do buffer na memória principal. Na melhor das hipóteses, se os dados couberem no buffer, eles podem ser lidos nos buffers e não há necessidade de reutilizar o disco. Na pior das hipóteses, espera-se que o buffer contenha apenas alguns blocos de dados – cerca de um bloco por conexão. No entanto, devido à grande quantidade de memória principal disponível hoje, essas suposições de pior caso são muito pessimistas. Na verdade, geralmente há muita memória principal disponível para processar a solicitação, e nossas estimativas de custo usam a quantidade de memória disponível para o operador como parâmetro.

Embora assumamos que os dados devem primeiro ser lidos do disco, é possível que o buffer de memória já esteja utilizando um bloco. Para simplificar, ignoramos esse efeito e, consequentemente, para o resultado, o custo real de acesso ao disco na execução de um plano pode ser menor que o estimado.

O tempo necessário para executar um plano de avaliação de consultas, supondo que não haja outras operações no computador, refletiria todos esses custos e poderia ser usado como uma medida do custo do plano. Infelizmente, é muito difícil estimar o tempo de resposta de um plano sem executá-lo por dois motivos:

- o tempo de resposta depende do conteúdo do buffer quando a consulta é iniciada; esse dado não está disponível quando a consulta é otimizada e, mesmo que esteja, é complicado de ser considerado;

- em um sistema com diversos discos, o tempo de resposta é baseado em como os acessos são distribuídos entre os discos, e para isso precisamos conhecer o layout dos dados.

Curiosamente, o plano pode ter um tempo de resposta melhor se consumir mais recursos. Por exemplo, se houver vários discos no sistema, o plano A, que requer leituras de diversos discos mas lê vários ao mesmo tempo, pode ser mais rápido que o plano B, que tem menos leituras de disco, mas acessa apenas um disco por vez. No entanto, se várias instâncias de uma consulta usando o plano A forem executadas simultaneamente, o tempo total de resposta poderá ser maior do que se as mesmas instâncias fossem executadas usando o plano B, porque o plano A sobrecarrega mais os discos.

Como resultado, os otimizadores geralmente procuram minimizar o custo total de recursos de um plano de consulta em vez de minimizar o tempo de resposta. Nosso modelo para estimar o tempo total de uso do disco (incluindo pesquisa e transferência de dados) é um exemplo de modelo de custo de solicitação com base nesse consumo de recursos.

6.3 Operação de seleção

Ao processar uma consulta, uma varredura de arquivo é o operador de processamento de dados de nível mais baixo. As varreduras de arquivos são algoritmos de pesquisa que procuram e recuperam registros que correspondem a uma condição de seleção. Em sistemas relacionais, esse operador permite que toda a relação seja lida se estiver armazenada em um arquivo separado.

6.3.1 Seleções usando varreduras de arquivos e índices

Considere uma operação de seleção para uma tabela cujos registros sejam armazenados em um único arquivo. O algoritmo de varredura mais óbvio para implementar a função de seleção é **A1 (busca linear)**. Em uma busca linear, o sistema varre cada bloco do arquivo e testa todos os registros para ver se eles correspondem aos critérios de seleção. A primeira busca é necessária para acessar o primeiro bloco do arquivo. Buscas adicionais podem ser necessárias se os blocos de arquivos não estiverem armazenados próximos uns dos outros, mas ignoramos esse efeito para facilitar.

Ainda que a seleção possa ser mais lenta do que outros algoritmos, a busca linear pode ser aplicada a qualquer arquivo, independentemente da classificação do arquivo, da disponibilidade de índices ou da natureza da operação de seleção.

As estimativas de custo para varredura linear e outros algoritmos de seleção são mostradas no quadro 4, em que foi utilizada uma árvore B+ de altura hi, assumindo que uma operação de E/S aleatória é necessária para cada nó no caminho da raiz até uma folha. A maioria dos otimizadores presume que os nós internos da árvore estão no buffer de memória porque são acessados com relativa frequência, e normalmente menos de 1% dos nós na árvore B+ correspondem a nós não folha. A fórmula de custo pode ser alterada corretamente cobrando apenas o custo de E/S aleatório por travessia da raiz à folha, definindo hi = 1.

Quadro 4 – Estimativas de custo para algoritmos de seleção

	Algoritmo	Custo	Motivo
A1	Busca linear	$t_S + b_r \times t_T$	Uma busca inicial mais b_r transferências de bloco, com b_r indicando o número de blocos no arquivo.
A1	Busca linear, igualdade sobre chave	Caso médio $t_S + (b_r / 2) \times t_T$	Como, no máximo, um registro satisfaz a condição, a varredura pode ser terminada assim que o registro requisitado for encontrado. No pior caso, b_r transferências de bloco ainda são necessárias.
A2	Índice agrupado de árvore B+, *i* gualdade sobre chave	$(h_i + 1) \times (t_T + t_S)$	(Em que h_i indica a altura do índice.) Pesquisa do índice atravessa a altura da árvore mais uma E/S para buscar o registro; cada uma dessas operações de E/S exige uma busca e uma transferência de bloco.
A3	Índice agrupado de árvore B+, igualdade sobre não chave	$h_i \times (t_T + t_S) + t_S + b \times t_T$	Uma busca para cada nível da árvore, uma busca para o primeiro bloco. Aqui, b é o número de blocos contendo registros com a chave de busca especificada, todos sendo lidos. Esses blocos são blocos de folha considerados como sendo armazenados sequencialmente (pois é um índice agrupado) e não exigem buscas adicionais.
A4	Índice secundário de árvore B+, igualdade sobre chave	$(h_i + 1) \times (t_T + t_S)$	Esse caso é semelhante ao índice agrupado.
A4	Índice secundário de árvore B+, igualdade sobre não chave	$(h_i + n) \times (t_T + t_S)$	(Em que n é o número de registros buscados.) Aqui, o custo da travessia do índice é o mesmo que o custo de A3, mas cada registro pode estar em um bloco diferente, exigindo uma busca por registro. O custo é potencialmente muito alto se n for grande.
A5	Índice agrupado de árvore B+, comparação	$h_i \times (t_T + t_S) + t_S + b \times t_T$	Idêntico ao caso de A3, igualdade sobre não chave.
A6	Índice secundário de árvore B+, comparação	$(h_i + n) \times (t_T + t_S)$	Idêntico ao caso de A4, igualdade sobre não chave.

Fonte: Silberschatz, Korth e Sudarshan (2020, p. 394).

As estruturas de índice são chamadas de caminhos de acesso porque fornecem uma via pela qual os dados podem ser encontrados e acessados. É eficiente ler registros de um arquivo em uma ordem que corresponda exatamente à ordem física.

Lembrete

Um índice clusterizado (também chamado de índice primário) permite ler registros em um arquivo em uma ordem que corresponda à ordem física do arquivo. Um índice não clusterizado é também chamado de índice secundário.

Os algoritmos de pesquisa que usam um índice são chamados de varreduras de índice. Usamos o predicado de seleção para nos orientar na escolha do melhor índice a ser utilizado ao processar a consulta. Os algoritmos de pesquisa que utilizam o índice são:

- **A2 (índice agrupado, igualdade sobre chave)**: para uma comparação de igualdade entre um atributo de chave e um índice primário, podemos utilizar o índice para pegar um único registro que atenda à condição de igualdade. As estimativas de custo estão no quadro 4. Para moldar a situação comum em que os nós internos do índice estão no buffer na memória, hi é definido como 1.

- **A3 (índice agrupado, igualdade sobre não chave)**: podemos separar diversos registros usando um índice primário quando a condição da seleção especifica uma validação de igualdade sobre um atributo não chave. Se diferencia de A2 apenas porque diversos registros podem ser separados. Contudo, os registros seriam guardados de forma sequencial no arquivo, que utiliza uma chave de busca para classificação. As estimativas de custo estão no quadro 4.

- **A4 (índice secundário, igualdade)**: as seleções que especificam uma condição de igualdade podem utilizar um índice secundário. Podemos separar um único registro se a condição de igualdade for sobre uma chave; diversas tuplas podem ser separadas se o campo de indexação não for uma chave. Para o primeiro caso, apenas um registro é separado, e o custo de tempo é o mesmo do caso A2 (índice agrupado). Para o segundo caso, cada registro pode residir em um bloco diferente, resultando em uma operação de E/S por registro separado, em que cada operação de E/S requer uma busca e uma transferência de bloco. O pior custo possível neste caso é (hi + n) × (tS + tT), em que n é o número de registros separados, se cada registro estiver em um bloco de disco diferente e as buscas de bloco forem feitas aleatoriamente. Na pior das hipóteses, o custo pode ser ainda maior do que o de uma pesquisa linear ao pesquisar muitos registros.

Se o buffer na memória for grande, o bloco que contém a entrada já pode estar no buffer, e essa probabilidade pode ser considerada para estimar o custo médio ou esperado de uma consulta. Com buffers grandes, a estimativa é muito menor do que a estimativa do pior caso.

Para alguns algoritmos, a organização de arquivo baseada em árvore B+ pode economizar um acesso, pois os registros estão armazenados no nível de folha da árvore.

Os índices secundários geralmente não armazenam referências aos registros mantidos em uma organização de arquivos usando uma árvore B+ ou outras organizações de arquivos que exigem a movimentação de registros. Os índices secundários armazenam os valores dos atributos usados como chaves de pesquisa na organização do arquivo por árvore B+. Acessar um registro por meio de um índice secundário é mais caro: primeiro o índice secundário é pesquisado pelos valores da chave de pesquisa na organização de arquivos da árvore B+, e em seguida a árvore B+ da organização de arquivos é pesquisada para encontrar os registros. A fórmula de custo descrita para indicadores secundários deve ser modificada de acordo quando tais indicadores secundários são usados.

6.3.2 Seleções envolvendo comparações

Vamos considerar uma seleção da forma $\sigma A \leq v(r)$. Podemos executá-la utilizando uma busca linear ou binária ou utilizando índices das seguintes maneiras:

- A5 (índice agrupado, comparação);
- A6 (índice secundário, comparação).

Para o caso A5, um índice agrupado de árvore B+ pode ser utilizado quando a condição de consulta é uma comparação. Para condições de comparação que estão na forma $A > v$ ou $A \geq v$, um índice agrupado conforme A poderá ser utilizado para direcionar a recuperação de tuplas. Para $A \geq v$, buscamos o valor v no índice para localizar a primeira tupla no arquivo que possua um valor $A \geq v$. Uma varredura de arquivo iniciando por essa tupla até o final do arquivo devolve todas as tuplas que atendem a condição. Para $A > v$, a varredura de arquivo inicia com a primeira tupla que apresente $A > v$. A estimativa de custo é idêntica à utilizada no caso A3.

Considerando as comparações na forma $A < v$ ou $A \leq v$, não precisamos fazer uma pesquisa de índice. Para $A < v$, podemos utilizar uma varredura de arquivo simples, desde o início do arquivo e continuando até a primeira tupla com atributo $A = v$ (mas sem considerá-la). O caso de $A \leq v$ é parecido, exceto que a varredura segue até a primeira tupla com atributo $A > v$ (mas sem considerá-la). O índice não é útil em nenhuma dessas circunstâncias.

No caso A6 (índice secundário, comparação), podemos utilizar um índice secundário ordenado para orientar a recuperação das condições de comparação que envolvem $<$, $\leq$, $\geq$ ou $>$. Os blocos de índice de menor nível são analisados, seja do menor valor até v (para $<$ e $\leq$), seja de v até o valor máximo (para $>$ e $\geq$).

O índice secundário fornece referências aos registros, mas para acessar os registros reais precisamos pegá-los usando referências. Essa etapa pode exigir uma operação de E/S para cada registro capturado, pois registros sucessivos podem residir em diferentes blocos de disco; toda operação de E/S requer uma busca de disco e uma transferência de bloco. Se o número de registros a serem pesquisados for grande, usar um índice secundário pode ser ainda mais caro do que uma pesquisa linear. Portanto, o segundo índice só deve ser usado quando poucos registros são selecionados.

Se o número de tuplas equivalentes for conhecido anteriormente, o otimizador de consulta pode optar entre um índice secundário e uma varredura linear baseando-se em previsões de custo. No entanto, se o número de tuplas correspondentes não for conhecido no momento da compilação, qualquer uma das opções pode apresentar um desempenho ruim, conforme o número real de tuplas correspondentes.

Nesse cenário, quando um índice secundário está livre mas o número de registros equivalentes não é conhecido com precisão, podemos trabalhar com um algoritmo híbrido que utiliza uma varredura de índice de mapa de bits, o qual primeiro cria um mapa com tantos bits quanto há blocos, e todos os bits são zerados. O algoritmo utiliza o índice secundário para encontrar índices para tuplas relacionadas, mas em vez de procurar as tuplas no mesmo instante, ele analisa à medida que cada item de índice é encontrado; o algoritmo obtém o número do bloco do item de índice e altera o bit correspondente no mapa para 1.

Após todas as entradas de índice terem sido processadas, o mapa de bits é analisado para encontrar todos os blocos com um bit definido como 1 – esses são justamente os blocos que contêm os registros correspondentes. A associação é tratada linearmente, mas os blocos sem um único bit são ignorados; apenas os blocos cujo bit correspondente indica 1 são pesquisados, então uma varredura interna de cada bloco é utilizada para recuperar todas as entradas associadas.

Saiba mais

Os índices de mapa de bits aumentam a velocidade de muitas estruturas de banco de dados. Para saber mais sobre o assunto, recomendamos ler o artigo de O'Neil e Quass (1997), que descreve o algoritmo:

O'NEIL, P.; QUASS, D. Improved query performance with variant indexes. *In:* SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1997, Tucson. *Proceedings* [...]. Nova York: ACM, 1997. p. 38-49.

Esse algoritmo pode ser um pouco mais custoso do que a varredura linear, mas na melhor das hipóteses é muito mais econômico. Da mesma maneira, no pior caso é apenas um pouco mais caro do que usar uma varredura de índice secundário para buscar tuplas diretamente, mas na melhor das hipóteses é muito mais econômico. Portanto, esse algoritmo híbrido assegura que o desempenho nunca seja muito pior do que o melhor plano para essa instância de banco de dados.

6.4 Operação de junção

O pseudocódigo apresentado a seguir (SILBERSCHATZ; KORTH; SUDARSHAN, 2020, p. 399) calcula a junção ($\mathbf{r} \bowtie_\theta \mathbf{s}$) de duas relações r e s. É denominado algoritmo de junção por loop aninhado pois consiste em um par de loops aninhados. A relação r representa a relação externa, e a relação s é a relação interna da junção, uma vez que, observando o algoritmo, o loop para a relação r encobre o loop para s.

O algoritmo usa a notação para as tuplas tr e ts, e para indicar a tupla resultante da concatenação dos atributos utiliza tr · ts:

for each tupla t_r **in** r **do begin**
  for each tupla t_s **in** s **do begin**
    testar par (t_r, t_s) para ver se satisfaz a condição da junção θ
    se satisfizer, acrescentar $t_r \cdot t_s$ ao resultado;
  end
end

Assim como o algoritmo de varredura de arquivo linear, o algoritmo de junção de loop aninhado não requer índices e pode ser usado independentemente das condições de junção. Estender o algoritmo para calcular a união natural é simples, pois pode ser expressa como uma união teta (*theta join*) seguida da eliminação de atributos duplicados por projeção. A única alteração necessária é uma etapa extra para remover atributos duplicados da tupla tr · ts antes de fornecer o resultado. Silberschatz, Korth e Sudarshan (2020, p. 399) explicam porque o algoritmo de junção por loop aninhado é custoso:

> O algoritmo de junção por loop aninhado é dispendioso, pois examina cada par de tuplas nas duas relações. Considere o custo do algoritmo de junção por loop aninhado. O número de pares de tuplas a serem consideradas é nr × ns, em que nr indica o número de tuplas em r e ns denota o número de tuplas em s. Para cada registro em r, temos de realizar uma varredura completa sobre s. Na pior das hipóteses, o buffer pode manter apenas um bloco de cada relação, e será necessário um total de nr × bs + br transferências de bloco (block transfers), com br e bs indicando o número de blocos contendo tuplas de r e s, respectivamente. Só precisamos de uma busca (seek) para cada varredura na relação interna s, pois ela é lida sequencialmente, e de br buscas para ler r, levando a um total de nr + br buscas. Na melhor das hipóteses, existe espaço suficiente para as duas relações caberem simultaneamente na memória, de modo que cada bloco terá de ser lido apenas uma vez; logo, somente br + bs transferências de bloco (block transfers) serão necessárias, juntamente com duas buscas (seeks).

Caso uma das relações esteja inteiramente na memória principal, é útil usá-la como uma relação interna, pois esta é lida apenas uma vez. Dessa forma, se s for pequena o suficiente para estar na memória principal, podemos requerer apenas um total de br + bs mudanças de bloco e duas buscas.

6.4.1 Junção por loop aninhado em bloco

Se o buffer for pequeno para receber qualquer relação completa na memória, ainda podemos economizar significativamente manipulando as relações bloco a bloco em vez de tupla. O pseudocódigo a seguir apresenta uma junção por loop aninhado em bloco, em que cada bloco da relação interna é colocado lado a lado com cada bloco na relação externa (SILBERSCHATZ; KORTH; SUDARSHAN, 2020,

p. 399). Em cada par de blocos, cada tupla em um bloco é emparelhada com cada tupla no outro bloco, para gerar todos os pares de tuplas. Todos os pares de tuplas que atendem à condição da junção são adicionados ao resultado.

```
for each bloco B_r of r do begin
    for each bloco B_s of s do begin
        for each tupla t_r in B_r do begin
            for each tupla t_s in B_s do begin
                testar par (t_r, t_s) para ver se satisfaz a condição de junção
                se satisfizer, acrescente t_r · t_s ao resultado;
            end
        end
    end
end
```

A diferença no custo da junção por loop aninhado em bloco e da junção por loop aninhado básico está no pior cenário. Cada bloco na relação interna será lido somente uma vez para cada bloco na relação externa, e não uma vez para cada tupla na relação externa. Dessa forma, teremos um total de br × bs + br transferências de bloco, em que br e bs representam o número de blocos que compreendem os registros de r e s. Precisamos de uma busca cada vez que se analisa a relação interna, e a análise da relação externa exige uma busca por bloco, totalizando 2 × br buscas. Certamente, se a memória não receber nenhuma das relações, é mais eficiente utilizar a relação menor como relação externa. Pensando no melhor cenário, em que a relação interna cabe na memória, haverá br + bs transferências de bloco e apenas duas buscas.

Podemos melhorar ainda mais o desempenho dos métodos de loop aninhado e loop aninhado em bloco:

- caso os atributos de uma junção natural ou de uma condição de junção que envolva apenas comparações de igualdade formem uma chave na relação interna, considerando cada tupla da relação externa, o loop interno pode terminar quando a primeira correspondência for encontrada;

- considerando o algoritmo por loop aninhado em bloco, poderíamos usar blocos de disco como unidade de bloco para a relação externa, utilizando o maior tamanho suportado pela memória, enquanto permitimos espaço satisfatório nos buffers para a relação interna e a saída;

- podemos varrer o loop interno para a frente e para trás, de modo intermitente, o que organiza as requisições para blocos de disco de modo que os dados que pertencem ao buffer a partir da varredura anterior poderão ser reutilizados, diminuindo a quantidade de acessos ao disco;

- caso exista um índice sobre o atributo de junção do loop interno, aconselha-se substituir as varreduras de arquivo por pesquisas de índice mais eficientes.

6.5 Plano de avaliação de consulta

O plano de avaliação de consultas constitui-se de uma árvore de álgebra relacional estendida, com anotações adicionais em cada nó informando os métodos de acesso a serem utilizados por cada tabela e o método de execução de cada operador relacional. Vamos considerar a seguinte consulta SQL (RAMAKRISHNAN; GEHRKE, 2008, p. 362):

```
SELECT  M.nome-marin
FROM    Reservas R, Marinheiros M
WHERE   R.id-marin = M.id-marin
        AND  R.id_barco = 100 AND M.avaliação > 5
```

Podemos expressar essa consulta utilizando álgebra relacional (RAMAKRISHNAN; GEHRKE, 2008, p. 362):

$$\pi_{\text{nome-marin}}(\sigma_{\text{id-barco = 100} \wedge \text{avaliação>5}}(\text{Reservas} \bowtie_{\text{id-marin = id-marin}} \text{Marinheiros}))$$

Essa expressão é apresentada na forma de uma árvore na figura 16. A expressão relacional caracteriza uma parte de como avaliar a consulta. Primeiramente calculamos a junção natural de Reservas e Marinheiros, depois executamos as seleções e por último projetamos o campo nome-marin.

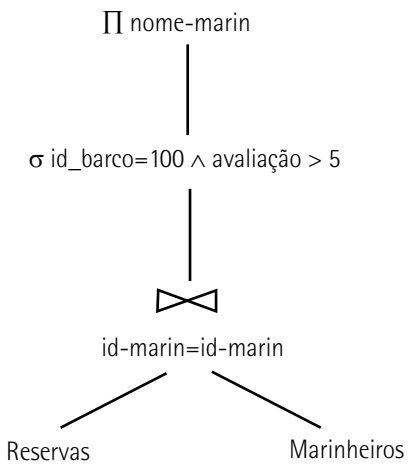

Figura 16 – Consulta expressa como uma árvore de álgebra relacional

Fonte: Ramakrishnan e Gehrke (2008, p. 362).

Para definir totalmente um plano de avaliação, precisamos considerar para a implementação cada uma das operações algébricas necessárias. Para exemplificar, vamos observar a figura 17, na qual foi utilizada uma junção de loops aninhados indexados, orientada a páginas com Reservas como a tabela externa e aplicando seleções e projeções em cada tupla no resultado da junção conforme ela é produzida – nunca armazenamos integralmente o resultado da junção antes das seleções e projeções. No desenho do plano de avaliação de consultas, adotou-se como convenção que a tabela externa é a filha à esquerda do operador de junção.

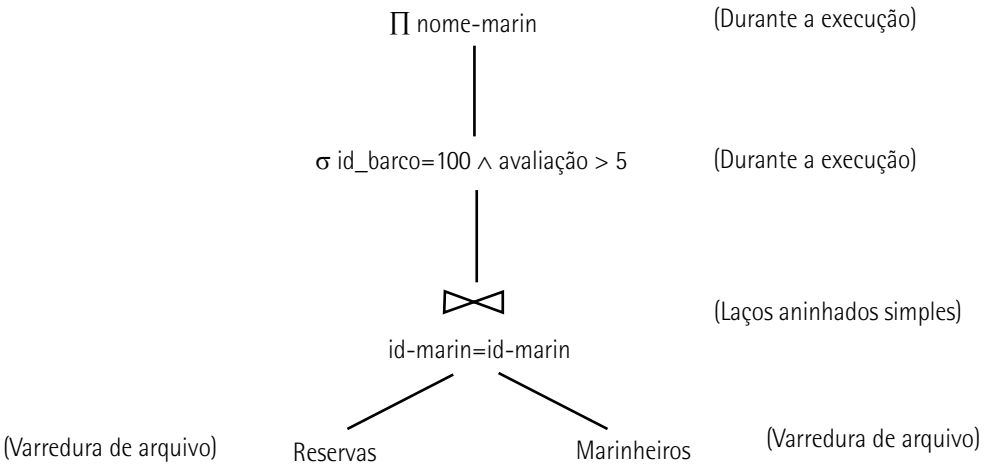

Figura 17 – Exemplo de plano de avaliação de consultas

Fonte: Ramakrishnan e Gehrke (2008, p. 362).

Resumo

Vimos nesta unidade as teorias que lidam com consultas em bancos de dados e como melhorá-las.

Um índice em arquivo funciona da mesma forma que o índice em um livro: nos ajuda a localizar determinado assunto, procurando por uma palavra ou frase, verificar a página em que o tópico aparece e depois ler as páginas indicadas para encontrar o tema. Com o uso de índices no banco de dados, podemos melhorar significativamente o desempenho das consultas.

Na ciência da computação, uma árvore B+ e uma árvore B são estruturas de dados do tipo árvore, que são utilizadas para manter a eficiência independentemente da inserção e exclusão dos dados.

As funções de hashing, por sua vez, são algoritmos que mapeiam uma massa de dados de tamanho variável para pequenos dados de tamanho fixo. Essas funções ajudam a melhorar o desempenho das consultas nos bancos de dados.

Por fim, a otimização da consulta é um método vital quando trabalhamos com volumosos bancos de dados. Seu objetivo é melhorar o plano de avaliação para obter consultas mais eficientes considerando as diversas estratégias geralmente possíveis para seu processamento, especialmente se a consulta for complexa.

Exercícios

Questão 1. Em um sistema de bancos de dados, o resultado de consultas é uma pequena proporção dos registros. Não é produtivo para o sistema ler cada registro e verificar se o campo ou os campos de condições atendem aos requisitos da pesquisa. O ideal é que o sistema consiga localizar os registros rapidamente. A respeito desse tema, avalie as afirmativas a seguir.

I – O uso de índices pode piorar significativamente o desempenho das consultas do banco de dados.

II – Há dois tipos básicos de índices: ordenados e hash.

III – Entre os índices tipo hash, podemos citar a organização por árvore B+.

É correto o que se afirma em

A) II, apenas.

B) III, apenas.

C) I e II, apenas.

D) II e III, apenas.

E) I, II, III.

Resposta correta: alternativa A.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: um índice é uma ferramenta de otimização capaz de melhorar substancialmente a performance das consultas dos bancos de dados. Um índice em arquivo funciona de forma similar ao índice em um livro, que facilita a localização de um determinado assunto nesse conteúdo.

II – Afirmativa correta.

Justificativa: os índices ordenados e os índices hash compõem os dois tipos básicos de índices em arquivo. Os índices ordenados utilizam uma ordem classificada dos valores, e os índices hash usam uma distribuição uniforme de valores por um intervalo.

III – Afirmativa incorreta.

Justificativa: as organizações por árvore B+ são um tipo de indexação ordenada. Uma árvore B+ é uma estrutura de dados do tipo árvore, sendo amplamente utilizada porque mantém a eficiência de busca, independentemente da inserção ou da exclusão de dados.

Questão 2. (FGV/2018, adaptada) Leia o texto a seguir a respeito da linguagem SQL.

A linguagem SQL (Structured Query Language) é uma linguagem declarativa, ao contrário das linguagens tradicionais, que são do tipo procedural. Isso permite ao programador expressar aquilo que pretende dizer exatamente da forma que o computador terá que operar para obter os resultados solicitados. Por exemplo, ao indicar que se pretende apresentar o resultado de uma pesquisa de forma ordenada, apenas iremos adicionar ao comando de seleção a cláusula ORDER BY, não indicando qual algoritmo de ordenação o computador deverá utilizar.

Embora as linguagens procedurais sejam mais rápidas ao serem executadas, as linguagens declarativas são mais flexíveis, pois referenciam os seus elementos por meio de um nome compreensível, e não por meio de uma posição física em disco ou memória. A linguagem SQL é orientada para o processamento de conjuntos (set-based language), algo que não se verifica com as linguagens mais tradicionais. A consulta a qualquer banco de dados relacional é sempre realizada utilizando o comando SELECT, que é, sem sombra de dúvidas, o comando mais utilizado e importante da linguagem.

DAMAS, L. SQL – *Structured Query Language*. Rio de Janeiro: LTC, 2014. p. 131 (com adaptações).

A otimização de consultas em gerenciadores de bancos de dados é fundamental para o desempenho do sistema. Consultas escritas em SQL são particularmente propícias à otimização, porque essa linguagem

A) É declarativa, permite a criação de diferentes planos de execução.

B) Tem uma sintaxe simples e é largamente utilizada.

C) Suporta todas as operações da álgebra relacional.

D) Permite o uso de subconsultas, facilitando os processos de busca.

E) Permite a criação de camadas de software de persistência.

Resposta correta: alternativa A.

Análise das alternativas

A) Alternativa correta.

Justificativa: a linguagem SQL não segue paradigma procedural, mas sim declarativo, no qual o programa expressa a lógica de um cálculo sem descrever seu fluxo de controle. Os diversos planos de execução permitem chegar ao mesmo resultado de consulta com pior ou melhor desempenho. Desse modo, essa característica é relevante ao processo de otimização de consultas.

B) Alternativa incorreta.

Justificativa: a sintaxe e a ampla utilização da linguagem SQL não são características relevantes à otimização de consultas.

C) Alternativa incorreta.

Justificativa: o suporte às operações da álgebra relacional é um requisito de um sistema de gerenciamento de bancos de dados, não estando associado à otimização de consultas.

D) Alternativa incorreta.

Justificativa: uma subconsulta é uma consulta que está aninhada dentro de uma instrução SELECT, INSERT, UPDATE, DELETE ou em outra subconsulta. O uso de subconsultas não é particularmente relevante para a otimização de consultas.

E) Alternativa incorreta.

Justificativa: a criação de camadas de software de persistência é uma característica do sistema de gerenciamento de bancos de dados, não sendo relevante à otimização de consultas.