

PROCESSAMENTO DE CONSULTAS

Os *processamentos de consultas* são as atividades envolvidas em extrair dados de um banco de dados. Essas atividades incluem a tradução de consultas expressas em linguagens de alto nível do banco de dados em expressões que podem ser implementadas no nível físico do sistema de arquivos, otimizações, traduções e na avaliação das consultas.

O custo do processamento de uma consulta é determinado pelo acesso ao disco, que é lento se comparado ao acesso à memória. Normalmente, há muitas estratégias possíveis para processar uma determinada consulta, especialmente se a consulta for complexa. A diferença entre uma estratégia boa e uma estratégia ruim, em termos do número de acessos de disco exigidos, é freqüentemente significativa, e pode ser de grande magnitude. Consequentemente, vale a pena para o sistema gastar uma quantia significativa de tempo na seleção de uma estratégia boa para processar uma consulta, até mesmo se a consulta for executada somente uma vez.

12.1 Introdução

Os passos envolvidos no processamento de uma consulta são ilustrados na Figura 12.1. Os passos básicos são:

1. Análise sintática e tradução.
2. Otimização.
3. Avaliação.

Antes de o processamento da consulta ter início, o sistema deve traduzir a consulta em uma forma utilizável. Uma linguagem como a SQL é adequada para o uso humano, mas não é adequada para ser a representação interna de uma consulta ao sistema. Uma representação interna mais útil é aquela baseada na álgebra relacional estendida.

Assim, a primeira ação que o sistema tem de tomar no processamento de uma consulta é traduzi-la em sua forma interna. Esse processo de tradução é semelhante ao trabalho realizado pelo analisador sintático de um compilador. Na geração da forma interna da consulta, o analisador sintático confere a sintaxe da consulta do usuário, verifica se os nomes das relações que aparecem na consulta são nomes de relações no banco de dados e assim por diante. Uma

representação da consulta em uma árvore sintática é gerada, e, então, traduzida para uma expressão algébrica relacional. Se a consulta estiver expressa em termos de uma visão, a fase de tradução a substituirá pela expressão algébrica relacional que define a visão.¹ A análise sintática é analisada na maioria dos textos sobre compiladores (veja Notas Bibliográficas) e está fora do escopo deste livro.

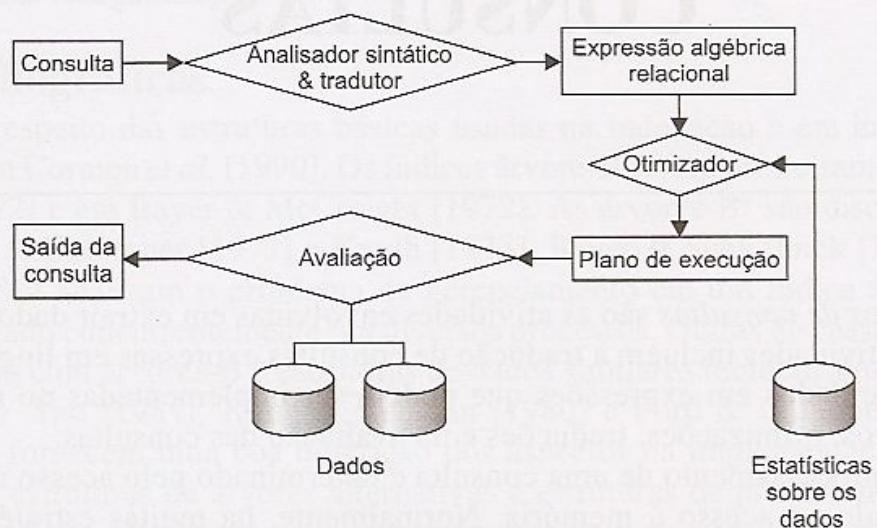


Figura 12.1 Passos no processamento de consultas.

Nos modelos de rede e hierárquico (discutidos nos Apêndices A e B) a otimização da consulta fica, na maioria das vezes, a cargo do programador da aplicação. Isso ocorre porque as declarações de linguagem para a manipulação de dados normalmente são embutidas na linguagem de programação hospedeira, e não é fácil transformar uma consulta à rede ou hierárquica em uma equivalente sem o conhecimento do programa de aplicação inteiro. Em contrapartida, linguagens de consulta relacionais são declarativas ou algébricas. As linguagens declarativas permitem aos usuários especificar o que uma consulta deve gerar, sem informar como o sistema deve fazer essa geração. As linguagens algébricas permitem a transformação algébrica das consultas dos usuários. Baseado na especificação da consulta, é relativamente fácil para um otimizador gerar uma variedade de planos equivalentes por consulta e escolher o menos oneroso dentre eles.

Neste capítulo, tratamos do modelo relacional. Veremos que a base algébrica desse modelo é consideravelmente útil para o otimizador da consulta. Dada uma consulta, geralmente, há vários métodos possíveis para calcular a resposta. Por exemplo, vimos que, em SQL, uma consulta poderia ser expressa de diversas maneiras diferentes. Cada consulta SQL pode ser traduzida em uma expressão de álgebra relacional de vários modos. Além disso, a representação algébrica relacional de uma consulta especifica apenas parcialmente como avaliar uma consulta; geralmente, há diversas maneiras de calcular expressões algébricas relacionais. Como exemplo, considere a consulta:

1. Para visões já existentes no banco, a expressão que define a visão já foi avaliada e armazenada. Portanto, a relação armazenada pode ser usada, não sendo feita a substituição. As visões recursivas são tratadas de outra forma, por meio de um procedimento de ponto fixo (*fixpoint*), conforme discutido na Seção 5.3.6.

```
select saldo
from conta
where saldo < 2500
```

Essa consulta pode ser traduzida nas duas expressões algébricas relacionais seguintes:

- $\sigma_{saldo < 2500}(\pi_{saldo}(conta))$
- $\pi_{saldo}(\sigma_{saldo < 2500}(conta))$

Além disso, podemos executar cada operação algébrica relacional usando um entre diversos algoritmos diferentes. Por exemplo, para implementar essa seleção, podemos procurar em todas as tuplas de *conta* a fim de encontrar as tuplas com saldo menor que 2.500. Se um índice árvore-B⁺ estiver disponível no atributo *saldo*, podemos usar o índice em vez de localizar as tuplas.

Para especificar como avaliar uma consulta completamente, precisamos não só prover a expressão algébrica relacional, mas também anotá-la com instruções que especificam como avaliar cada operação. As anotações podem definir o algoritmo a ser usado para uma operação específica ou o(s) índice(s) que será(ão) usado(s). Uma operação algébrica relacional anotada com instruções sobre como ser avaliada é chamada de *avaliação primitiva*. Várias avaliações primitivas podem ser agrupadas em *pipeline*, na qual várias operações são executadas em paralelo. Uma seqüência de operações primitivas, que podem ser usadas para avaliar uma consulta, é um *plano de execução de consulta* ou *plano de avaliação de consulta*. A Figura 12.2 ilustra um plano de avaliação para nossa consulta do exemplo no qual um índice em particular (denotado na figura como “índice 1”) é especificado para a operação de seleção. A *máquina de execução da consulta* toma um plano de avaliação da consulta, executa aquele plano e retorna as respostas para a consulta.

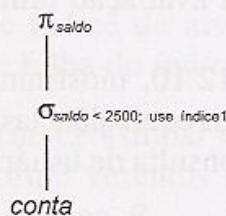


Figura 12.2 Um plano de avaliação de consulta.

Os diferentes planos de avaliação para uma determinada consulta podem ter custos diferentes. Não se espera que os usuários escrevam suas consultas de uma maneira que sugiram o plano de avaliação mais eficiente. Em vez disso, é responsabilidade do sistema construir um plano de avaliação de consulta que minimize seu custo. Como explicamos, normalmente a medida de desempenho mais relevante é o número de acessos ao disco.

A *otimização de consultas* é o processo de selecionar o plano de avaliação de consulta mais eficiente para uma consulta. Um aspecto da otimização acontece no nível da álgebra relacional. Faz-se uma tentativa de encontrar uma expressão que é equivalente à expressão dada, tentando encontrar uma expressão mais eficiente para execução. Discutiremos a equivalência de expressões com outros detalhes na Seção 12.9. Outro aspecto envolve a

seleção de uma estratégia detalhada para o processamento da consulta, como a escolha de um algoritmo para ser usado na execução de uma operação, a escolha de índices específicos para usar e assim por diante.

Para escolher entre diferentes planos de avaliação de consulta, o otimizador deve *estimar* o custo de cada plano de avaliação. O cálculo do custo exato da avaliação de um plano normalmente não seria possível sem executar o plano de fato.

Em vez disso, os otimizadores fazem uso de informações estatísticas sobre as relações, como os tamanhos das relações e as profundidades dos índices, para relizar uma boa estimativa do custo de um plano.

Considere o exemplo precedente de uma seleção aplicada à relação *conta*. O otimizador estima o custo dos diferentes planos de avaliação.

Se há um índice disponível no atributo *saldo* de *conta*, então o plano de avaliação mostrado na Figura 12.2, no qual a seleção é feita usando o índice, provavelmente terá um custo menor, e, assim, deve ser escolhido.

Uma vez escolhido o plano de consulta, a consulta é avaliada com aquele plano e o resultado da consulta é produzido.

A seqüência de passos já descrita para o processamento de uma consulta é ilustrativa; nem todos os bancos de dados seguem esses passos exatamente. Por exemplo, em vez de usar uma representação algébrica relacional, vários bancos de dados usam uma representação de árvore sintática anotada baseada na estrutura da consulta de SQL dada. Entretanto, os conceitos que descrevemos aqui formam a base do processamento de consultas em bancos de dados.

Na Seção 12.2, construímos um modelo de custo que nos permite estimar o custo de várias operações. Usando essa medida de custo, obtemos as avaliações ótimas das operações individuais nas Seções 12.3 a 12.7. Na Seção 12.8 examinamos a eficiência que podemos alcançar combinando várias operações em uma operação pipeline. Essas ferramentas permitem-nos determinar o custo aproximado da avaliação otimizada de uma determinada expressão algébrica relacional.

Finalmente, nas Seções 12.9 e 12.10, mostramos as equivalências entre expressões algébricas relacionais. Podemos usar essas equivalências para substituir uma expressão algébrica relacional construída a partir de uma consulta de usuário por uma expressão equivalente cujo custo estimado de avaliação é menor.

12.2 Catálogo de Informações para a Estimativa de Custo

A estratégia que escolhemos para a avaliação da consulta depende de seu custo estimado. Os otimizadores de consulta usam informações estatísticas armazenadas no catálogo do DBMS para estimar o custo de um plano. O catálogo de informações relevantes sobre as relações incluem:

- n_r é o número de tuplas na relação r .
- b_r é o número de blocos que contêm tuplas da relação r .
- s_r é o tamanho em bytes de uma tupla da relação r .
- f_r é o fator de bloco da relação r , ou seja, o número de tuplas da relação r que cabe em um bloco.

- $V(A, r)$ é o número de valores distintos que aparecem na relação r para o atributo A . Esse valor é igual ao tamanho de $\Pi_A(r)$. Se A é uma chave para a relação r , $V(A, r)$ é n_r .
- $SC(A, r)$ é a cardinalidade de seleção do atributo A da relação r . Dados uma relação r e um atributo A da relação, $SC(A, r)$ é o número médio de registros que satisfazem uma condição de igualdade no atributo A , dado que pelo menos um registro satisfaz a condição de igualdade. Por exemplo, $SC(A, r) = 1$ se A é um atributo-chave de r ; para um atributo que não é chave, estimamos que os valores distintos de $V(A, r)$ são distribuídos uniformemente entre as tuplas, produzindo $SC(A, r) = (n_r / V(A, r))$.

As duas últimas estatísticas, $V(A, r)$ e $SC(A, r)$, se desejado, também podem ser mantidas para conjuntos de atributos, em vez de apenas para atributos individuais. Assim, dado um conjunto de atributos \mathcal{A} , $V(\mathcal{A}, r)$ é o tamanho de $\Pi_{\mathcal{A}}(r)$.

Se as tuplas da relação r estiverem armazenadas fisicamente juntas em um arquivo, a seguinte equação é válida:

$$b_r = \left[\frac{n_r}{f_r} \right]$$

Além do catálogo de informações sobre as relações, o seguinte catálogo de informações sobre índices também é usado:

- f_i é o *fan-out* médio dos nós internos do índice i para índices estruturados em árvore, como árvores-B⁺.
- HT_i é o número de níveis no índice i – ou seja, a altura do índice i . Para um índice de árvore balanceada (como uma árvore-B⁺) no atributo A da relação r , $HT_i = \lceil \log_{f_i}(V(A, r)) \rceil$. Para um índice hash, HT_i é 1.
- LB_i é o número de blocos de índice de nível mais baixo no índice i , ou seja, o número de blocos no nível de folha do índice.

Usamos variáveis estatísticas a fim de estimar o tamanho do resultado e do custo para várias operações e algoritmos, conforme veremos nas seções seguintes. Referimo-nos à estimativa de custo de um algoritmo A como E_A .

Se desejarmos manter estatísticas precisas, então, toda vez que uma relação for modificada, também temos de atualizar as estatísticas. Essa atualização gera um overhead significativo. Por isso, a maioria dos sistemas não atualiza as estatísticas em todas as modificações. Em vez disso, as atualizações são feitas durante períodos de carga do sistema. Como resultado, as estatísticas usadas para escolher uma estratégia de processamento de consultas podem não ser completamente precisas. Porém, se não ocorrem muitas atualizações no intervalo entre as atualizações das estatísticas, as estatísticas serão suficientemente precisas para prover uma boa estimativa dos custos relativos de planos diferentes.

A informação estatística mencionada aqui é simplificada. Otimizadores reais mantêm mais informações estatísticas para melhorar a precisão de suas estimativas de custo dos planos de avaliação.

12.3 Medidas do Custo de uma Consulta

O custo de avaliação de uma consulta pode ser medido por meio de vários recursos diferentes, incluindo acessos a disco, tempo de CPU para executar uma consulta e, em um sistema de banco de dados distribuído ou paralelo, o custo da comunicação (que discutiremos nos Capítulos 17 e 18). O tempo de resposta para um plano de avaliação de consulta (ou seja, o tempo necessário para executar o plano), supondo que nenhuma outra atividade esteja sendo executada no computador, contabilizaria todos esses custos e poderia ser usado como uma boa medida do custo do plano.

Entretanto, em grandes sistemas de banco de dados, os acessos ao disco (os quais medimos como o número de transferências de blocos do disco), normalmente, é o custo mais importante, já que esses acessos são lentos se comparados com as operações feitas na memória. Além disso, a velocidade das CPUs têm aumentado muito mais rapidamente que as velocidades de disco. Assim, é provável que o tempo gasto em atividades de disco continue dominando o tempo total para executar uma consulta. Finalmente, estimar o tempo de CPU é relativamente difícil, comparado com a estimativa de custo de acesso a disco. Assim, o custo de acesso a disco é considerado uma medida adequada do custo para um plano de avaliação de consultas.

Para simplificar nosso cálculo do custo de acesso a disco, supomos que todas as transferências de blocos têm o mesmo custo. Essa hipótese não considera a variação proveniente da latência diagonal (aguardar até os dados desejados girarem abaixo da cabeça de leitura e escrita) e o tempo de busca (o tempo gasto para mover a cabeça para cima da trilha ou cilindro desejados). Embora esses fatores sejam significativos, eles são difíceis de calcular em um sistema compartilhado. Por isso, usamos simplesmente o *número de blocos transferidos* do disco como medida do custo.

Também ignoramos o custo de escrever o resultado final de uma operação de volta para o disco. Qualquer que seja o plano de avaliação de consulta utilizado, esse custo não se altera; assim, ignorá-lo não afeta a escolha de um plano.

Os custos de todos os algoritmos que consideramos dependem significativamente do tamanho do buffer na memória principal. No melhor caso, todos os dados podem ser lidos para o buffer, e o disco não precisa ser acessado novamente. No pior caso, supomos que o buffer pode manter apenas alguns blocos de dados – aproximadamente um bloco por relação. Quando apresentarmos estimativas de custo, geralmente faremos a suposição do pior caso.

12.4 Operação de Seleção

No processamento de consultas, a *varredura de arquivos* é o operador de mais baixo nível para ter acesso aos dados. As varreduras de arquivo são algoritmos de procura que localizam e recuperam os registros que estão de acordo com uma condição de seleção. Em sistemas relacionais, a varredura de arquivos permite ler uma relação inteira nos casos nos quais a relação é armazenada em um arquivo único e dedicado.

12.4.1 Algoritmos Básicos

Considere uma operação de seleção em uma relação cujas tuplas são armazenadas juntas em um único arquivo. Dois algoritmos de varredura para implementar a operação de seleção são os seguintes:

- **A1 (busca linear).** Em uma busca linear, cada bloco de arquivo é varrido e todos os registros² são testados para verificar se satisfazem a condição de seleção. Como todos os blocos precisam ser lidos, $E_{A1} = b_r$. (Lembre-se de que E_{A1} denota o custo estimado do algoritmo A1.) Para uma seleção em um atributo-chave, supomos que a metade dos blocos é varrida antes de o registro ser encontrado, ponto no qual a varredura termina. A estimativa nesse caso é $E_{A1} = (b_r/2)$.

Embora possa ser ineficiente em muitos casos, o algoritmo de busca linear pode ser aplicado a qualquer arquivo, indiferentemente da ordem do arquivo ou da disponibilidade de índices.

- **A2 (busca binária).** Se o arquivo é ordenado em um atributo e a condição de seleção é uma comparação de igualdade no atributo, podemos usar uma busca binária para localizar os registros que satisfazem a seleção. A busca binária é realizada nos blocos do arquivo, dando a seguinte estimativa para os blocos do arquivo a serem varridos:

$$E_{A2} = \lceil \log_2(b_r) \rceil + \left\lceil \frac{SC(A, r)}{f_r} \right\rceil - 1$$

O primeiro termo, $\lceil \log_2(b_r) \rceil$, contabiliza o custo para localizar a primeira tupla por meio da busca binária nos blocos. O número total de registros que satisfarão a seleção é $SC(A, r)$, e esses registros ocuparão $\lceil SC(A, r)/f_r \rceil$ blocos,³ dos quais um já havia sido recuperado, dando a estimativa acima.

Se a condição de igualdade estiver em um atributo-chave, então $SC(A, r) = 1$, e a estimativa se reduz a $E_{A2} = \lceil \log_2(b_r) \rceil$.

As estimativas de custo para a busca binária baseiam-se na hipótese de que os blocos de uma relação são armazenados no disco de forma contígua. Caso contrário, o custo da procura nas estruturas de acesso a arquivos (que podem estar no disco) para localizar o endereço físico de um bloco em um arquivo deve ser somado às estimativas. As estimativas de custo também dependem do tamanho do resultado da seleção.

Se supormos uma distribuição uniforme de valores (ou seja, cada valor aparece com probabilidade igual), então estimamos que a consulta $s_{A=a}(r)$ tenha:

$$SC(A, r) = \frac{n_r}{V(A, r)}$$

tuplas, supondo que o valor a apareça no atributo A de algum registro r . A suposição de que o valor a na seleção aparece em algum registro geralmente é verdadeira, então as estimativas de custo fazem essa suposição implicitamente. Entretanto, não é realista supor que cada valor aparece com probabilidade igual. O atributo *nome_agência* da relação *conta* é um

2. Estamos supondo que o arquivo contém uma relação, de forma que cada registro corresponda exatamente a uma tupla.

3. Essa fórmula é baseada na hipótese (normalmente válida) de que o conjunto de registros para um determinado valor começa na fronteira de um bloco. Se essa hipótese não for válida, um acesso de bloco adicional é necessário, no pior caso.

exemplo em que essa suposição não é válida. Há uma tupla na relação *conta* para cada conta. É razoável esperar que as grandes agências tenham mais contas que as agências menores. Então, certos valores de *nome_agência* aparecem com maior probabilidade que outros. Apesar de a suposição de distribuição uniforme não ser freqüentemente correta, ela é uma aproximação razoável da realidade em muitos casos, e ajuda-nos a manter nossa apresentação relativamente simples.

Como uma ilustração do uso da estimativa de custo, suponha que temos a seguinte informação estatística sobre a relação *conta*:

- $f_{conta} = 20$ (ou seja, 20 tuplas de *conta* cabem em um único bloco).
- $V(nome_agência, conta) = 50$ (ou seja, há 50 agências diferentes).
- $V(saldo, conta) = 500$ (ou seja, há 500 valores diferentes de *saldo*).
- $n_{conta} = 10.000$ (ou seja, a relação *conta* possui 10.000 tuplas).

Considere a consulta:

$$\sigma_{nome_agência = "Perryridge"}(conta)$$

Como a relação tem 10.000 tuplas, e cada bloco mantém 20 tuplas, o número de blocos é $b_{conta} = 500$. Então, uma varredura de arquivo simples em *conta* gasta 500 acessos de blocos.

Suponha que *conta* esteja ordenado por *nome_agência*. Como, então, $V(nome_agência, conta) = 50$, esperamos que $10.000/50 = 200$ tuplas da relação *conta* pertençam à agência Perryridge. Essas tuplas caberiam em $200/20 = 10$ blocos. Uma busca binária para encontrar o primeiro registro levaria $\lceil \log_2(500) \rceil = 9$ acessos de bloco. Assim, o custo total seria $9 + 10 - 1 = 18$ acessos de bloco.

12.4.2 Seleções Usando Índices

As estruturas de índices são chamadas de *paths (caminhos) de acesso*, pois fornecem um caminho pelo qual os dados podem ser localizados e acessados. No Capítulo 11, mostramos que é eficiente ler os registros de um arquivo em uma ordem próxima de sua ordem física. Lembre-se de que um índice primário é um índice que permite que os registros de um arquivo sejam lidos em uma ordem que corresponde à ordem física no arquivo. Um índice que não é um índice primário é chamado de índice secundário.

Os algoritmos de busca que usam um índice são chamados de *varreduras de índice*. Os índices ordenados, como as árvores-B⁺, também permitem acesso ordenado às tuplas, o que é útil para a implementação de consultas de faixa. Embora os índices possibilitem acesso rápido, direto e ordenado, sua utilização impõe um overhead de acesso àqueles blocos que contêm o índice. Precisamos considerar esses acessos a bloco quando estimamos o custo de uma estratégia que envolve o uso de índices. Usamos o predicado da seleção para guiar-nos na escolha do índice a ser usado no processamento de consultas.

- **A3 (índice primário, igualdade na chave).** Para uma comparação de igualdade em um atributo-chave com um índice primário, podemos usar o índice para recuperar um único registro que satisfaz a condição de igualdade correspondente. Para recobrar um único registro, precisamos recuperar um bloco a mais que o número de níveis do índice (HT_i); o custo é $E_{A3} = HT_i + 1$.

- **A4 (índice primário, igualdade em atributo que não é chave).** Podemos recuperar registros múltiplos usando um índice primário quando a condição de seleção especifica uma comparação de igualdade em um atributo A que não é chave. $SC(A, r)$ registros satisfarão uma condição de igualdade e $[SC(A, r)/f_r]$ blocos de arquivo serão acessados; consequentemente:

$$E_{A4} = HT_i + \left\lceil \frac{SC(A, r)}{f_r} \right\rceil$$

- **A5 (índice secundário, igualdade).** Seleções que especificam uma condição de igualdade podem usar um índice secundário. Essa estratégia pode recuperar um único registro se o campo de indexação for uma chave; podem ser recuperados registros múltiplos se o campo de indexação não for uma chave. Para uma condição de igualdade no atributo A , $SC(A, r)$ registros satisfazem a condição. Dado que o índice é um índice secundário, supomos o cenário do pior caso no qual cada registro que satisfaz a condição reside em um bloco diferente, assim obtemos $E_{A5} = HT_i + SC(A, r)$, ou, para um atributo de indexação que é chave, $E_{A5} = HT_i + 1$.

Vamos supor a mesma informação estatística sobre *conta* usada no exemplo anterior. Também vamos supor que os seguintes índices existem em *conta*:

- Um índice árvore-B⁺ primário para o atributo *nome_agência*.
- Um índice árvore-B⁺ secundário para o atributo *saldo*.⁴

Conforme mencionado anteriormente, para simplificar, supomos que os valores são distribuídos uniformemente.

Considere a consulta:

$$\sigma_{\text{nome_agência} = \text{"Perryridge}}(\text{conta})$$

Já que $V(\text{nome_agência}, \text{conta}) = 50$, esperamos que $10.000/50 = 200$ tuplas da relação *conta* pertençam à agência Perryridge. Suponha que usamos o índice para *nome_agência*. Uma vez que o índice é um índice clustering, $200/20 = 10$ leituras de blocos são necessárias para ler as tuplas de *conta*. Além disso, vários blocos de índice devem ser lidos. Suponha que o índice árvore-B⁺ armazene 20 ponteiros por nó. Como há 50 nomes diferentes de agências, o índice árvore-B⁺ deve ter entre três e cinco nós folha. Com esse número de nós folha, a árvore inteira tem uma profundidade de dois, então dois blocos de índice devem ser lidos. Assim, a estratégia precedente requer um total de 12 leituras de blocos.

12.4.3 Seleções Que Envolvem Comparações

Considere uma seleção da forma $\sigma_{A \leq v}(r)$. Na ausência de qualquer informação adicional sobre a comparação, supomos que aproximadamente a metade dos registros satisfará a condição da comparação; consequentemente, o resultado tem $n/2$ tuplas.

4. Na prática, é pouco provável que um índice seja mantido para o *saldo*, uma vez que os saldos mudam com muita frequência. Entretanto, o exemplo é útil, neste ponto, para ilustrar os custos relativos de vários planos de acesso.

Se o valor real usado na comparação (v) estiver disponível na hora da estimativa do custo, uma estimativa mais precisa poderá ser feita. Os valores mais baixos e mais altos ($\min(A, r)$ e $\max(A, r)$) para o atributo podem ser armazenados no catálogo. Supondo que os valores são uniformemente distribuídos, podemos estimar que o número de registros que satisfazem a condição $A \leq v$ será 0 se $v < \min(A, r)$, será n_r se $v \geq \max(A, r)$, e

$$n_r \cdot \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$$

para os demais casos.

Podemos implementar as seleções que envolvem comparações usando busca linear ou busca binária, ou, ainda, usando índices de algum dos seguintes modos:

- **A6 (índice primário, comparação).** Um índice primário ordenado (por exemplo, um índice primário árvore-B⁺) pode ser usado quando a condição de seleção for uma comparação. Para condições de comparação da forma $A > v$ ou $A \geq v$, o índice primário pode ser usado para obter a recuperação das tuplas, como segue. Para $A \geq v$, procuramos o valor v no índice para encontrar a primeira tupla no arquivo que tem o valor de $A = v$. Uma varredura do arquivo a partir daquela tupla até o final do arquivo retorna todas as tuplas que satisfazem a condição. Para $A > v$, a varredura do arquivo começa na primeira tupla que satisfaz $A > v$.

Para comparações da forma $A < v$ ou $A \leq v$, uma procura no índice não é necessária. Para $A < v$, usamos uma varredura de arquivo simples a partir do início do arquivo, e continuamos até (mas não inclusive) a primeira tupla com atributo $A = v$. O caso de $A \leq v$ é semelhante, exceto que a varredura continua até (inclusive) a primeira tupla com atributo $A > v$. Em ambos os casos, o índice não é necessário.

Vamos supor que aproximadamente a metade dos registros satisfaz uma das condições. Sob essa suposição, a recuperação usando o índice tem o seguinte custo:

$$E_{A6} = HT_i + \frac{b_r}{2}$$

Se o valor real usado na comparação estiver disponível na hora da estimativa de custo, uma estimativa mais precisa pode ser feita. Seja o número estimado de valores que satisfazem a condição (conforme descrito anteriormente) c . Então:

$$E_{A6} = HT_i + \left\lceil \frac{c}{f_r} \right\rceil$$

- **A7 (índice secundário, comparação).** Podemos usar um índice ordenado secundário para guiar a recuperação para condições de comparação envolvendo $<$, \leq , \geq ou $>$. Os blocos de índice de mais baixo nível são varridos a partir do valor menor até v (para $<$ e \leq) ou a partir de v até o valor máximo (para $>$ e \geq). Para essas comparações, se supusermos que pelo menos a metade dos registros satisfaz a condição, então a metade dos blocos de índice de mais baixo nível é acessada e, por meio do índice, a metade dos registros do arquivo é acessada. Além disso, um caminho deve ser

percorrido no índice a partir da raiz do bloco para o primeiro bloco folha a ser usado. Assim, a estimativa de custo é a seguinte:

$$E_{A7} = HT_i + \frac{LB_i}{2} + \frac{n_r}{2}$$

Assim como podemos fazer com comparações de desigualdade em índices clustering, é possível obter uma estimativa mais precisa se soubermos o valor real a ser usado na comparação na hora da estimativa do custo.

No Sistema SQL Non-Stop de Tandem, as árvores-B⁺ são usadas tanto para o armazenamento primário de dados quanto em caminhos de acesso secundário. Os índices primários são índices clustering, já os secundários não o são. Em lugar de ponteiros para a localização física de registros, os índices secundários contêm as chaves para a procura na árvore-B⁺ primária. A fórmula do custo, descrita previamente para índices secundários, precisa ser modificada ligeiramente se esses índices forem usados.

Embora os algoritmos precedentes mostrem que os índices são úteis no processamento de seleções com comparações, eles nem sempre são tão úteis. Como uma ilustração, considere a consulta:

$$\sigma_{saldo < 1200}(conta)$$

Suponha que a informação estatística sobre as relações seja a mesma que usamos anteriormente. Se não temos nenhuma informação sobre os valores mínimo e máximo de saldo na relação *conta*, supomos que metade das tuplas satisfaz a seleção.

Se usarmos um índice para o *saldo*, estimamos o número de acessos de bloco como se segue. Vamos supor que cabem 20 ponteiros em um nó do índice árvore-B⁺ para o *saldo*. Uma vez que há 500 valores diferentes de saldo, e cada nó folha da árvore deve preencher pelo menos a metade, a árvore possui entre 25 e 50 nós folha. Assim, como no caso do índice para o *nome_agência*, o índice para *saldo* possui uma profundidade de dois, e dois acessos de bloco são necessários para ler o primeiro bloco de índice. No pior caso, há 50 nós folha, metade dos quais deve ser acessada. Esses acessos levam a mais 25 leituras de blocos. Finalmente, para cada tupla que localizamos no índice, temos de recuperar aquela tupla da relação. Estimamos que 5.000 tuplas (metade das 10.000 tuplas) satisfaçam a condição. Considerando que o índice não é clustering, no pior caso cada um desses acessos de tupla exigirá um acesso de bloco separado. Assim, obtemos um total de 5.027 acessos de bloco.

Em contrapartida, uma varredura simples de arquivo fará somente $10.000/20 = 500$ acessos de bloco. Nesse caso, obviamente não é uma boa idéia usar o índice, em vez disso, deveríamos usar a varredura de arquivo.

12.4.4 A Implementação de Seleções Complexas

Até o momento, consideramos somente condições de seleção simples da forma *A op B*, em que *op* é uma operação de igualdade ou de comparação. Agora, consideraremos predicados de seleção mais complexos.

- **Conjunção:** uma *seleção de conjunção* é uma seleção da forma:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$$

Podemos estimar o resultado dessa seleção como segue. Para cada θ_i , estimamos o tamanho da seleção $\sigma_{\theta_i}(r)$, denotado por s_i , como descrito anteriormente. Assim, a probabilidade de que uma tupla na relação satisfaça a condição de seleção θ_i é s_i/n_r .

A probabilidade precedente é chamada de *seletividade* da seleção $\sigma_{\theta_i}(r)$. Supondo que as condições sejam *independentes* entre si, a probabilidade de que uma tupla satisfaça todas as condições é simplesmente o produto de todas as probabilidades. Assim, estimamos o tamanho da seleção completa como:

$$n_r * \frac{s_1 * s_2 * \dots * s_n}{n_r^n}$$

- **Disjunção:** Uma seleção de disjunção é uma seleção da forma:

$$\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$$

Uma condição de disjunção é satisfeita pela união de todos os registros que satisfaçam as condições individuais simples θ_i .

Como anteriormente, supõe-se que s_i/n_r denote a probabilidade com que uma tupla satisfaz a condição θ_i . Então, a probabilidade de que uma tupla satisfaça a disjunção é dada por 1 menos a probabilidade de que ela não satisfaça nenhuma das condições, ou

$$1 - \left(1 - \frac{s_1}{n_r}\right) * \left(1 - \frac{s_2}{n_r}\right) * \dots * \left(1 - \frac{s_n}{n_r}\right)$$

Multiplicando esse valor por n_r , obtemos o número de tuplas que satisfazem a seleção.

- **Negação:** o resultado de uma seleção $\sigma_{\neg\theta}(r)$ são as tuplas de r que não estão em $\sigma_\theta(r)$. Já sabemos como estimar o tamanho de $\sigma_\theta(r)$. Portanto, o tamanho estimado de $\sigma_{\neg\theta}(r)$ é:

$$tamanho(r) - tamanho(\sigma_\theta(r))$$

Podemos implementar uma operação de seleção que envolve uma conjunção ou uma disjunção de condições simples usando um dos seguintes algoritmos:

- **A8 (seleção de conjunção usando um índice).** Primeiro, determinamos se um caminho de acesso está disponível para um atributo em uma das condições simples. Se há um caminho, um dos algoritmos de seleção A2 até A7 pode recuperar os registros que satisfazem aquela condição. Completamos a operação testando, no buffer da memória, se cada registro recuperado satisfaz ou não as condições simples restantes.

A seletividade é fundamental para determinar em que ordem as condições simples em uma seleção conjuntiva devem ser testadas. A condição mais seletiva (quer dizer, aquela com menor seletividade) recuperará o menor número de registros; consequentemente, aquela condição deveria constituir a primeira varredura

- **A9 (seleção de conjunção usando índice composto).** Um índice composto apropriado pode estar disponível para algumas seleções conjuntivas. Se a seleção especifica

uma condição de igualdade em dois ou mais atributos, e há um índice composto nesses campos de atributo combinados, então pode-se procurar diretamente no índice. O tipo de índice determina qual dos algoritmos, A3, A4 ou A5, será utilizado.

- **A10** (*seleção de conjunção por meio da interseção de identificadores*). Outra alternativa para implementar as operações de seleção conjuntiva envolve o uso de ponteiros de registros ou identificadores de registros. Esse algoritmo requer índices com registros de ponteiros nos campos envolvidos nas condições individuais. Cada índice é varrido em busca de ponteiros para tuplas que satisfaçam uma condição individual. A interseção de todos os ponteiros recuperados é o conjunto de ponteiros para tuplas que satisfazem a condição conjuntiva. Então, usamos os ponteiros para recuperar os registros de fato. Se os índices não estão disponíveis em todas as condições individuais, então os registros recuperados são testados em relação às condições restantes.
- **A11** (*seleção de disjunção por meio da união de identificadores*). Se caminhos de acesso estiverem disponíveis em todas as condições de uma seleção de disjunção, cada índice é varrido em busca de ponteiros para tuplas que satisfaçam a condição individual. A união de todos os ponteiros recuperados fornece o conjunto de ponteiros para todas as tuplas que satisfaçam a condição de disjunção. Usamos, então, os ponteiros para recuperar os registros de fato.

Entretanto, se uma das condições não possui um caminho de acesso, teremos de executar uma varredura linear da relação para encontrar as tuplas que satisfaçam a condição. Então, se houver uma condição de disjunção nessa situação, o método de acesso mais eficiente é a varredura linear, com a conjunção de disjunção sendo testada em cada tupla durante a varredura.

Deixamos a implementação de seleções com condições de negação como exercício (Exercício 12.15). Também deixamos como exercício as condições que combinam conjunções e disjunções (Exercício 12.16).

Para ilustrar os algoritmos precedentes, suponha que temos a consulta:

```
select número_conta
  from conta
 where nome_agência = "Perryridge" and saldo = 1200
```

Supomos que a informação estatística sobre a relação *conta* é a mesma do exemplo anterior.

Se usarmos o índice para *nome_agência*, teremos um total de 12 leituras de bloco, conforme discutido na seção anterior. Se usarmos o índice para *saldo*, estimamos o número de acessos de bloco como segue. Como $V(saldo, conta) = 500$, esperamos que $10.000/500 = 20$ tuplas da relação *conta* tenham um saldo de 1.200 dólares. Porém, como o índice para *saldo* não é clustering, antecipamos que uma leitura de bloco será necessária para cada tupla. Assim, 20 leituras de bloco são necessárias para ler as tuplas de *conta*.

Vamos supor que caibam 20 ponteiros em um nó do índice árvore-B⁺ para *saldo*. Como há 500 valores de saldo diferentes, a árvore tem entre 25 e 50 nós folha. Assim, como no caso para o índice árvore-B⁺ para *nome_agência*, o índice para *saldo* tem uma profundidade

de dois, e são requeridos dois acessos a bloco para ler os blocos de índice necessários. Então, essa estratégia requer um total de 22 leituras de bloco.

Assim, concluímos que é preferível usar o índice para *nome_agência*. Observe que, se ambos os índices não fossem clustering, preferiríamos usar o índice para *saldo*, já que esperaríamos que somente 10 tuplas tivessem *saldo* = 1.200, contra 200 tuplas com *nome_agência* = "Perryridge". Sem a propriedade de clustering, nossa estratégia necessitaria de 200 acessos de bloco para ler os dados, uma vez que, no pior caso, cada tupla está em um bloco diferente. Somamos esses 200 acessos aos 2 acessos de bloco de índice e obtemos um total de 202 leituras de blocos. Porém, devido à propriedade de clustering do índice *nome_agência*, de fato, é mais barato, nesse exemplo, usar o índice para *nome_agência*.

Outro modo no qual poderíamos usar os índices para processar nosso exemplo de consulta é por meio da utilização da interseção de identificadores. Usamos o índice para *saldo* a fim de recuperar ponteiros para registros com *saldo* = 1.200, em lugar de recuperar os próprios registros. Façamos S_1 denotar esse conjunto de ponteiros. De forma semelhante, usamos o índice para *nome_agência* para recuperar ponteiros para registros com *nome_agência* = "Perryridge". Façamos S_2 denotar esse conjunto de ponteiros. Então, $S_1 \cap S_2$ é um conjunto de ponteiros para registros com *nome_agência* = "Perryridge" e *saldo* = 1.200.

Essa técnica exige que ambos os índices sejam acessados. Ambos os índices têm uma altura de dois e, para cada índice, o número de ponteiros recuperados, estimados anteriormente como 20 e 200, caberá em uma única página folha. Assim, lemos um total de quatro blocos de índice para recuperar os dois conjuntos de ponteiros. A interseção dos dois conjuntos de ponteiros pode ser calculada sem I/O de disco adicional. Estimamos o número de blocos que devem ser lidos do arquivo *conta* estimando o número de ponteiros em $S_1 \cap S_2$.

Uma vez que $V(\text{nome_agência}, \text{conta}) = 50$ e $V(\text{saldo}, \text{conta}) = 1.000$, estimamos que uma tupla em $50 * 1.000$ ou uma tupla em 50.000 tem simultaneamente *nome_agência* = "Perryridge" e *saldo* = 1.200. Essa estimativa baseia-se na suposição de distribuição uniforme (a qual fizemos anteriormente) e em uma suposição adicional de que a distribuição de nomes de agência e de saldos são independentes. Baseado nessas hipóteses, estima-se que $S_1 \cap S_2$ tenha só um pontoiro. Assim, somente um bloco de *conta* precisa ser lido. O custo total estimado dessa estratégia são cinco leituras de blocos.

12.5 Classificação

A classificação de dados tem um papel importante em sistemas de banco de dados por duas razões. Primeiro, as consultas SQL podem especificar que o resultado seja apresentado ordenadamente. Segundo, e igualmente importante para o processamento de consultas, diversas das operações relacionais, como as junções, podem ser implementadas eficazmente se as relações de entrada forem primeiramente classificadas. Por isso, discutimos a classificação antes de discutir a operação de junção na Seção 12.6.

Podemos realizar a classificação construindo um índice na chave de classificação e, então, usando aquele índice para ler a relação na ordem de classificação. Entretanto, esse processo somente ordena a relação *logicamente*, por meio de um índice, em lugar de fazê-lo *fisicamente*. Conseqüentemente, a leitura de tuplas na ordem de classificação pode conduzir a um acesso de disco para cada tupla. Por essa razão, pode ser desejável ordenar as tuplas fisicamente.

O problema da classificação tem sido estudado exaustivamente, tanto para o caso em que a relação cabe completamente na memória principal, quanto para o caso em que a relação é maior que a memória. No primeiro caso, as técnicas-padrão de classificação, como o *quicksort*, podem ser usadas. Aqui, discutimos como tratar do segundo caso.

A classificação de relações que não cabem na memória é chamada de *classificação externa*. A técnica mais comum usada para a classificação externa é o algoritmo de *sort-merge⁵ externo*. Descrevemos o algoritmo de sort-merge externo a seguir. Seja M a notação para o número de frames de página no buffer da memória principal (o número de blocos de disco cujos conteúdos podem ser colocados no buffer da memória principal).

1. No primeiro estágio, são executadas várias classificações temporárias.

```
i = 0;
repeat
```

leia M blocos da relação, ou o resto da relação, aquilo que for menor;

ordene a parte da relação que está na memória;

escreva os dados ordenados no arquivo temporário R_i ;

$i = i + 1$;

```
until o fim da relação
```

2. Na segunda fase, faz-se o *merge* nos arquivos temporários. Suponha, por enquanto, que o número total de temporários, N , seja menor que M , de forma que possamos alocar um frame de página para cada temporário e tenhamos espaço para manter uma página de resultado. A fase de merge opera da seguinte forma:

leia um bloco de cada um dos N arquivos R_i para uma página de buffer na memória;

```
repeat
```

escolha a primeira tupla (na ordem de classificação) entre todas as páginas do buffer;

escreva a tupla no resultado e apague-a da página de buffer;

if a página de buffer de qualquer temporário R_i está vazia and not fim de arquivo(R_i)

then leia o próximo bloco de R_i na página de buffer;

```
until todas as páginas de buffer que estiverem vazias
```

O resultado da fase de merge é a relação classificada. O arquivo de resultado fica no buffer para reduzir o número de operações de escrita em disco. A operação de merge precedente é uma generalização do merge de duas vias usado pelo algoritmo-padrão de sort-merge em memória; ele faz o merge em N temporários, sendo então chamado de *merge de n-vias*.

Geralmente, se a relação é muito maior que a memória, pode haver M ou mais temporários gerados na primeira fase, e não é possível alocar um frame de página para cada temporário

5. N.T.: Sort = classificar; colocar em uma determinada ordem.

Merge = intercalar; fundir, combinar.

durante a fase de merge. Nesse caso, faz-se a operação de merge em múltiplos passos. Então, como há memória suficiente para $M - 1$ páginas de buffer de entrada, cada merge terá $M - 1$ temporários como entrada.

O *passo* inicial funciona da seguinte forma. Faz-se o merge sobre os primeiros $M - 1$ temporários (conforme descrito anteriormente) para obter um único temporário para o próximo passo. Então, é feito o merge dos próximos $M - 1$ temporários de forma semelhante, e assim por diante, até que todos os temporários iniciais tenham sido processados. Nesse ponto, o número de temporários foi reduzido para um fator de $M - 1$. Se esse número reduzido de temporários ainda é maior ou igual a M , outro passo é dado, usando os temporários criados pelo primeiro passo como entrada. Cada passo reduz o número de temporários por um fator de $M - 1$. Esses passos são repetidos tantas vezes quantas forem necessárias, até que o número de temporários seja menor que M ; então, um passo final gera o resultado classificado.

A Figura 12.3 ilustra os passos do sort-merge externo de uma relação de exemplo. Para fins de ilustração, supomos que apenas uma tupla caiba em um bloco ($f_r = 1$), e supomos que a memória mantém três frames de página no máximo. Durante o estágio de merge, dois frames de página são usados para entrada e um para o resultado.

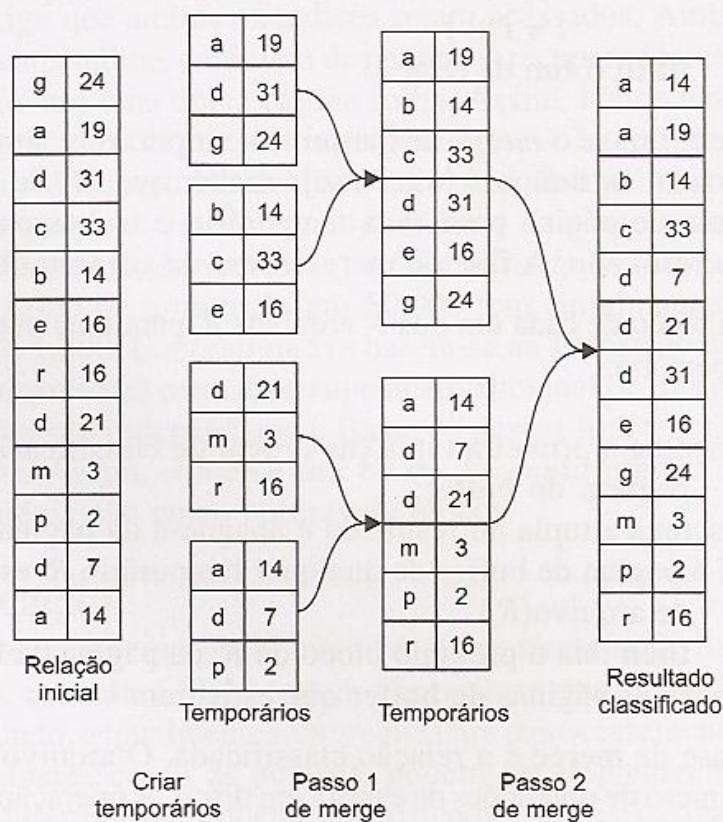


Figura 12.3 Classificação externa usando sort-merge.

Vamos, agora, calcular quantas transferências de bloco são necessárias para o sort-merge externo. Na primeira fase, todo bloco da relação é lido e escrito novamente, dando um total de $2b_r$ acessos de disco. O número inicial de temporários é $[b_r / M]$. Como o número de temporários diminui por um fator de $M - 1$ em cada passo de merge, o número total de passos

de merge necessários é dado por $\lceil \log_{M-1}(b_r / M) \rceil$. Cada um desses passos lê cada bloco da relação uma vez e os escreve uma vez, com duas únicas exceções. Primeiro, o passo final pode produzir um resultado ordenado sem escrevê-lo no disco. Segundo, pode haver temporários que não são lidos ou não são escritos durante um passo – por exemplo, se houver M temporários para fazer o merge em um passo, $M - 1$ são lidos e sofrem o merge, e um temporário não é acessado durante o passo. Ignorando as economias (relativamente pequenas) devido ao efeito anterior, o número total de acessos de disco para a classificação externa da relação é:

$$b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$

Aplicando essa equação ao exemplo da Figura 12.3, obtemos um total de $12 * (4 + 1) = 60$ transferências de blocos, conforme pode ser verificado na figura. Observe que esse valor não inclui o custo de escrita do resultado final.

12.6 Operação de Junção

Nesta seção, primeiro mostramos como estimar o tamanho do resultado de uma junção. Então, estudamos vários algoritmos para calcular a junção de relações e analisamos seus respectivos custos. Usamos a palavra *equi-join* para referir-nos a uma junção da forma $r \bowtie_{r.A = s.B} s$, em que A e B são atributos ou conjuntos de atributos das relações r e s , respectivamente.

Usamos como exemplo a expressão:

$$\text{depositante} \bowtie \text{cliente}$$

Vamos supor o seguinte catálogo de informação sobre as duas relações:

- $n_{\text{cliente}} = 10.000$.
- $f_{\text{cliente}} = 25$, o que implica $b_{\text{cliente}} = 10.000/25 = 400$.
- $n_{\text{depositante}} = 5.000$.
- $F_{\text{depositante}} = 50$, o que implica $b_{\text{depositante}} = 5.000/50 = 100$.
- $V(\text{nome_cliente}, \text{depositante}) = 2.500$, o que implica que, em média, cada cliente tem duas contas.

Vamos supor, também, que o *nome_cliente* em *depositante* seja uma chave estrangeira em *cliente*.

12.6.1 Estimativa do Tamanho das Junções

O produto cartesiano $r \times s$ contém $n_r * n_s$ tuplas. Cada tupla de $r \times s$ ocupa $s_r + s_s$ bytes, a partir dos quais podemos calcular o tamanho do produto cartesiano.

A estimativa do tamanho de uma junção natural é um pouco mais complicada que a estimativa do tamanho de uma seleção ou de um produto cartesiano. Sejam $r(R)$ e $s(S)$ relações.

- Se $R \cap S = \emptyset$ – ou seja, as relações não têm nenhum atributo em comum –, então, $r \bowtie s$ é igual a $r \times s$, e podemos usar nossa técnica de estimativa para produtos cartesianos.
- Se $R \cap S$ é uma chave para R , então sabemos que uma tupla de s irá juntar-se com pelo menos uma tupla de r . Assim, o número de tuplas em $r \bowtie s$ não é maior que o

número de tuplas em s . Se $R \cap S$ é uma chave estrangeira para R , então o número de tuplas em $r \bowtie s$ é exatamente igual ao número de tuplas em s . O caso em que $R \cap S$ é uma chave para S é simétrico ao caso anterior.

Em nosso exemplo de $\text{depositante} \bowtie \text{cliente}$, name_cliente em depositante é uma chave estrangeira de cliente ; consequentemente, o tamanho do resultado é exatamente $n_{\text{depositante}}$, que é 5.000.

- O caso mais difícil a considerar é quando $R \cap S$ não é uma chave para R ou para S . Nesse caso, supomos, como fizemos para as seleções, que cada valor aparece com probabilidade igual. Considere uma tupla t de r e suponha $R \cap S = \{A\}$. Estimamos que a tupla t produz:

$$\frac{n_s}{V(A, s)}$$

tuplas em $r \bowtie s$, uma vez que esse é o número médio de tuplas em s com um determinado valor para os atributos A . Considerando todas as tuplas em r , estimamos que há

$$\frac{n_r * n_s}{V(A, s)}$$

tuplas em $r \bowtie s$. Observe que, se invertêssemos os papéis de r e s na estimativa precedente, obteríamos uma estimativa de:

$$\frac{n_r * n_s}{V(A, r)}$$

tuplas em $r \bowtie s$. Essas duas estimativas diferem se $V(A, r) \neq V(A, s)$. Se essa situação acontece, há a probabilidade de haver tuplas pendentes que não participam da junção. Assim, a mais baixa das duas estimativas provavelmente é a mais precisa.

Essa estimativa do tamanho de junção será muito alta se os valores de $V(A, r)$ para os atributos A em r possuírem poucos valores em comum com $V(A, s)$ para os atributos A em s . Entretanto, essa situação é improvável de acontecer no mundo real, já que as tuplas pendentes não existem ou constituem somente uma pequena fração das tuplas na maioria das relações do mundo real. Mais importante é salientar que a estimativa depende da suposição de que cada valor aparece com a mesma probabilidade. Técnicas mais sofisticadas para a estimativa do tamanho devem ser usadas se essa hipótese não for válida.

Agora vamos calcular uma estimativa do tamanho para $\text{depositante} \bowtie \text{cliente}$ sem utilizar as informações sobre chaves estrangeiras. Como $V(\text{name_cliente}, \text{depositante}) = 2.500$ e $V(\text{name_cliente}, \text{cliente}) = 10.000$, as duas estimativas que obtemos são $5.000 * 10.000 / 2.500 = 20.000$ e $5.000 * 10.000 / 10.000 = 5.000$, escolhemos a menor. Nesse caso, a mais baixa dessas estimativas é igual àquela calculada anteriormente usando as informações sobre chaves estrangeiras.

12.6.2 Junção de Laço Aninhado

O procedimento da Figura 12.4 mostra um algoritmo simples para calcular a junção teta, $r \bowtie_{\theta} s$, de duas relações r e s . Esse algoritmo é chamado de *algoritmo de junção de laço aninhado* e consiste, basicamente, em um par de laços **for** aninhados. Conforme mostrado no procedimento, r é chamado de relação *externa* e s , de relação *interna* da junção, já que o laço para r inclui o laço para s . O algoritmo usa a notação $t_r \cdot t_s$, em que t_r e t_s são tuplas; $t_r \cdot t_s$ denota a tupla obtida concatenando os valores dos atributos das tuplas t_r e t_s .

```

for each tupla  $t_r$  in  $r$  do begin
    for each tupla  $t_s$  in  $s$  do begin
        teste o par  $(t_r, t_s)$  para ver se satisfazem a condição de junção  $\theta$ 
        se satisfizerem, adicione  $t_r \cdot t_s$  ao resultado.
    end
end

```

Figura 12.4 Junção de laço aninhado.

Assim como o algoritmo de varredura linear de arquivo para a seleção, o algoritmo de junção de laço aninhado não requer índices, e pode ser usado seja qual for a condição de junção. A extensão do algoritmo para calcular a junção natural é direta, uma vez que a junção natural pode ser expressa como uma junção teta seguida da eliminação de atributos repetidos por meio de uma projeção. A única mudança necessária é um passo adicional para remover os atributos repetidos das tuplas $t_r \cdot t_s$ antes de adicioná-las ao resultado.

O algoritmo de junção de laço aninhado é caro, já que examina todos os pares de tuplas nas duas relações. Considere o custo do algoritmo de junção de laço aninhado. O número de pares de tuplas a ser considerado é $n_r * n_s$. Para cada registro em r , temos de executar uma varredura completa em s . No pior caso, o buffer pode manter apenas um bloco de cada relação, e um total de $n_r * b_s + b_r$ acessos de bloco serão necessários. No melhor caso, há espaço suficiente para que ambas as relações caibam na memória, assim cada bloco terá de ser lido somente uma vez; consequentemente, apenas $b_s + b_r$ acessos de bloco serão necessários.

Se a relação menor couber completamente na memória principal, é melhor usar essa relação como a relação interna. Essa opção aumenta a velocidade do processamento da consulta significativamente, já que é necessário ler a relação interna apenas uma vez. Portanto, se s é pequeno o bastante para caber na memória principal, nossa estratégia requer apenas um total de $b_s + b_r$ acessos – o mesmo custo para o caso em que ambas as relações cabem na memória.

Agora, considere a junção natural de *depositante* e *cliente*. Vamos supor, por enquanto, que não temos nenhum índice em nenhuma das relações, e que não desejamos criar qualquer índice. Podemos usar laços aninhados para calcular a junção; vamos supor que *depositante* é a relação externa e que *cliente* é a relação interna na junção. Teremos de examinar $5.000 * 10.000 = 50 * 10^6$ pares de tuplas. No pior caso, o número de acessos de blocos é $5.000 * 400 + 100 = 2.000.100$. Entretanto, no cenário do melhor caso, podemos ler ambas as

relações de uma só vez, e executar o cálculo. Esse cálculo requer no máximo $100 + 400 = 500$ acessos a blocos – uma melhoria significativa sobre o cenário do pior caso. Se tivéssemos usado o *cliente* como relação do laço externo e *depositante* para o laço interno, o custo do pior caso para nossa estratégia final teria sido mais baixo: $10.000 * 100 + 400 = 1.000.400$.

12.6.3 Junção de Laço Aninhado de Blocos

Se o buffer for muito pequeno para manter ambas as relações na memória, podemos obter uma maior economia de acessos a bloco se processarmos as relações em uma base de blocos, em lugar de fazermos na base de tuplas. O procedimento da Figura 12.5 mostra uma variante da junção de laço aninhado, em que cada bloco da relação interna é emparelhado com cada bloco da relação externa. Dentro de cada par de blocos, cada tupla em um bloco é emparelhada com cada tupla do outro bloco, para gerar todos os pares de tuplas. Como no exemplo anterior, todos os pares de tuplas que satisfaçam a condição de 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
                teste o par  $(t_r, t_s)$  para ver se satisfazem a condição de
                junção
                se satisfizerem, adicione  $t_r, t_s$  ao resultado.
            end
        end
    end

```

Figura 12.5 Junção de laço aninhado de blocos.

A diferença primária no custo entre a junção de laço aninhado de blocos e a junção de laço aninhado básica é que, no pior caso, cada bloco da relação interna s é lida apenas uma vez para cada bloco da relação exterior, em vez de uma vez para cada tupla da relação exterior. Assim, no pior caso, haverá um total de $b_r * b_s + b_r$ acessos de blocos. Obviamente, é mais eficiente usar a relação menor como relação externa. No melhor caso, haverá $b_r + b_s$ acessos de bloco.

Vamos retornar a nosso exemplo de cálculo de $depositante \bowtie cliente$, usando o algoritmo de junção de laço aninhado de blocos. No pior caso, temos de ler cada bloco de *cliente* uma vez para cada bloco de *depositante*. Assim, no pior caso, um total de $100 * 400 + 100 = 40.100$ acessos de bloco são necessários. Esse custo é uma melhoria significativa sobre os $5.000 * 400 + 100 = 2.000.100$ acessos de bloco necessários no pior caso para a junção de laço aninhado básica. O número de acessos de bloco no melhor caso continua o mesmo – nominalmente, $100 + 400 = 500$.

O desempenho dos procedimentos de junção de laço aninhado e de junção de laço aninhado de blocos ainda pode ser melhorado:

- Se os atributos de junção em uma junção natural ou em uma *equi-join* formam uma chave na relação interna, então o laço interno pode terminar assim que a primeira correspondência for encontrada.
- No algoritmo de junção de laço aninhado de blocos, em vez de usar blocos de disco como unidade de bloco para a relação externa, podemos usar o maior tamanho que couber na memória, deixando espaço suficiente para os buffers da relação interna e para o resultado. Essa alteração reduz o número de vezes em que a relação interna é varrida.
- Podemos varrer o laço interno de maneira alternada para frente e para trás. Esse método de varredura ordena as solicitações de blocos de tal forma que os dados de uma varredura anterior que permanecem no buffer podem ser usados novamente, reduzindo, assim, o número de acessos de disco necessários.
- Se um índice estiver disponível no atributo de junção do laço interno, substituímos as varreduras de arquivo por procura por meio de índices, que são mais eficientes. Essa otimização é descrita na Seção 12.6.4.

12.6.4 Junção de Laço Aninhado Indexada

Em uma junção de laço aninhado (Figura 12.4), se um índice (permanente ou temporário) estiver disponível no atributo de junção do laço interno, procura por meio de índice podem substituir as varreduras de arquivos. Para cada tupla t_r na relação externa r , o índice é usado para procurar as tuplas em s que satisfaçam a condição de junção com a tupla t_r .

O método de junção precedente é chamado de *junção de laço aninhado indexada*. Ele pode ser usado com índices existentes, como também com índices temporários criados com o único propósito de avaliar a junção.

A procura de tuplas em s que satisfarão as condições de junção com uma dada tupla t_r é essencialmente uma seleção em s . Por exemplo, considere $depositante \bowtie cliente$. Suponha que temos uma tupla *depositante* com *nome_cliente* “John”. Então, as tuplas relevantes em s são aquelas que satisfazem a seleção “*nome_cliente* = John”.

O custo da junção de laço aninhado indexada pode ser calculado da seguinte forma. Para cada tupla na relação externa r , a procura é executada no índice para s e as tuplas relevantes são recuperadas. No pior caso, há espaço no buffer somente para uma página de r e para uma página do índice. Então, b_r acessos de disco são necessários para ler a relação r , e, para cada tupla em r , executamos uma procura no índice em s . Então, o custo da junção pode ser calculado como $b_r + n_r * c$, em que c é o custo de seleção única em s usando a condição de junção. Já vimos como calcular o custo de uma única seleção em uma relação usando vários tipos de índices. Esse cálculo nos dá o valor de c . A fórmula de custo indica que, se os índices estiverem disponíveis nas relações r e s , geralmente é mais eficiente usar aquela que tem menos tuplas, como a relação externa.

Por exemplo, considere uma junção de laço aninhado indexada de $depositante \bowtie cliente$, com *depositante* como relação externa. Suponha também que *cliente* possui um índice árvore-B⁺ primário no atributo de junção *nome_cliente*, que contém 20 entradas em cada nó do índice.

Uma vez que *cliente* possui 10.000 tuplas, a altura da árvore é 4, e é necessário mais um acesso para encontrar os dados. Como $n_{depositante}$ é 5.000, o custo total é $100 + 5.000 * 5 = 25.100$ acessos de disco. Esse custo é menor que os 40.100 acessos necessários para a junção de laço aninhado de bloco.

12.6.5 Merge-Junção

O algoritmo *merge-junção* (também chamado de *algoritmo sort-merge-junção*) pode ser usado para calcular junções natural e equi-join. Sejam $r(R)$ e $s(S)$ relações cuja junção natural será calculada, e seja $R \cap S$ a notação para seus atributos em comum. Suponha que ambas as relações estejam classificadas nos atributos $R \cap S$. Então, sua junção pode ser calculada por um processo muito parecido com o estágio de merge do algoritmo merge-sort.

O algoritmo merge-junção é mostrado na Figura 12.6. No algoritmo, *AtribJunção* refere-se aos atributos em $R \cap S$, e $t_r \bowtie t_s$, em que t_r e t_s são tuplas que possuem os mesmos valores para *AtribJunção*, denota a concatenação dos atributos das tuplas, seguida pela projeção dos atributos repetidos. O algoritmo merge-junção associa um ponteiro a cada relação. Esses ponteiros apontam inicialmente para a primeira tupla da primeira relação. Conforme o algoritmo prossegue, os ponteiros movem-se por meio da relação. Um grupo de tuplas de uma relação com o mesmo valor nos atributos de junção é lido em S_s . O algoritmo mostrado na Figura 12.6 requer que todos os conjuntos de tuplas S_s caibam na memória principal; posteriormente, nesta seção, vamos examinar extensões do algoritmo para evitar essa hipótese. Então, as tuplas correspondentes (se houver) da outra relação são lidas, e processadas conforme são lidas.

A Figura 12.7 mostra duas relações que estão classificadas no atributo de junção *a1*. É interessante verificar os passos do algoritmo merge-junção nas relações da figura. Como as relações estão na ordem de classificação, as tuplas com o mesmo valor nos atributos de junção estão em ordem consecutiva. Por isso, cada tupla na ordem de classificação precisa ser lida somente uma vez, e, como resultado, cada bloco também é lido somente uma vez. Como apenas um único passo é realizado por meio de ambos os arquivos, o método merge-junção é eficiente; o número de acessos a bloco é igual à soma do número de blocos em ambos os arquivos, $b_r + b_s$.

Se nenhuma das relações de entrada r e s estiver ordenada nos atributos de junção, elas podem primeiramente ser ordenadas e, então, o algoritmo merge-junção pode ser usado. O algoritmo merge-junção também pode ser estendido facilmente a partir de junção natural para o caso mais genérico de equi-join.

Suponha que o esquema merge-junção seja aplicado a nosso exemplo *depositante* \bowtie *cliente*. Aqui, o atributo de junção é *nome_cliente*. Suponha que as relações já estejam ordenadas no atributo de junção *nome_cliente*. Nesse caso, o merge-junção gasta um total de $400 + 100 = 500$ acessos de bloco. Suponha que as relações não estejam ordenadas e que o tamanho da memória seja o pior caso, ou seja, três blocos. Para classificar *cliente* são necessárias $400 * (2[\log_2(400/3)] + 1)$, ou 6.800 transferências de blocos, e mais 400 transferências para escrever o resultado. De forma semelhante, para classificar *depositante* são necessárias $100 * (2[\log_2(100/3)] + 1)$, ou 1.300 transferências de blocos, e mais 100 transferências para escrever o resultado. Assim, o custo total são 9.100 transferências de blocos se as relações não estiverem ordenadas.

```

 $pr :=$  endereço da primeira tupla de  $r$ ;
 $ps :=$  endereço da primeira tupla de  $s$ ;
while ( $ps \neq$  nulo and  $pr \neq$  nulo) do
    begin
         $t_s :=$  tupla para qual  $ps$  aponta;
         $S_s := \{t_s\}$ ;
        configure  $ps$  para apontar para a próxima tupla de  $s$ ;
         $acabou :=$  falso;
        while (not  $acabou$  and  $ps \neq$  nulo) do
            begin
                 $t'_s :=$  tupla para qual  $ps$  aponta;
                if ( $t'_s[AtribJunção] = t_s[AtribJunção]$ )
                    then begin
                         $S_s := S_s \cup \{t'_s\}$ ;
                        configure  $ps$  para apontar para a próxima tupla de  $s$ ;
                    end
                else  $acabou :=$  verdadeiro;
            end
             $t_r :=$  tupla para a qual  $pr$  aponta;
            while ( $pr \neq$  nulo and  $t_r[AtribJunção] < t_s[AtribJunção]$ ) do
                begin
                    configure  $pr$  para apontar para a próxima tupla de  $r$ ;
                     $t_r :=$  tupla para qual  $pr$  aponta;
                end
            while ( $pr \neq$  nulo and  $t_r[AtribJunção] = t_s[AtribJunção]$ ) do
                begin
                    for each  $t_s$  in  $S_s$  do
                        begin
                            adicione  $t_s \bowtie t_r$  ao resultado;
                        end
                    configure  $pr$  para apontar para a próxima tupla de  $r$ ;
                     $t_r :=$  tupla para a qual  $pr$  aponta;
                end
            end

```

Figura 12.6 Merge-junção.

Conforme mencionado anteriormente, o algoritmo merge-junção da Figura 12.6 requer que o conjunto S_s de todas as tuplas com o mesmo valor para os atributos de junção caiba na memória principal. Normalmente, essa exigência pode ser satisfeita, até mesmo se a relação s for grande. Se ela não puder ser satisfeita, uma junção de laço aninhado de bloco deve ser executada entre S_s e as tuplas em r com os mesmos valores para os atributos de junção. Como resultado, o custo global do merge-junção aumenta.

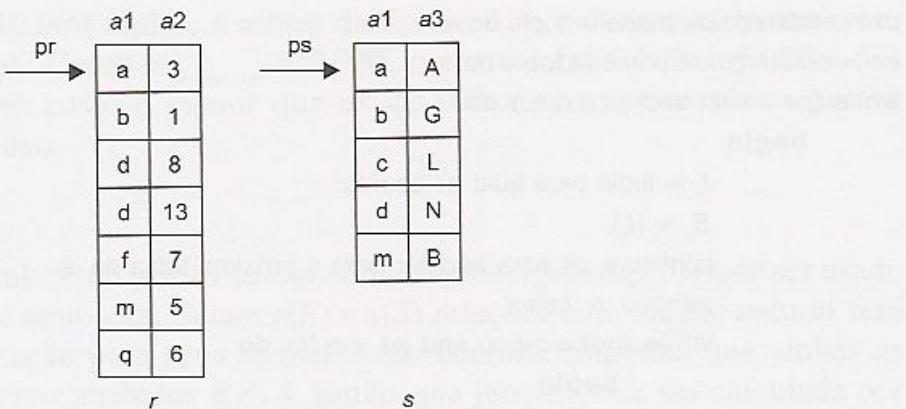


Figura 12.7 Relações classificadas para merge-junção.

Também é possível executar uma variação da operação merge-junção em tuplas não classificadas se houver índices secundários em ambos os atributos de junção. Os registros são varridos por meio dos índices, resultando em sua recuperação na ordem de classificação. Entretanto, essa variação apresenta uma desvantagem significativa, já que os registros podem estar espalhados pelos blocos de arquivo. Conseqüentemente, cada acesso a uma tupla poderia envolver o acesso a um bloco de disco, o que seria oneroso.

Para evitar esse custo, podemos usar uma técnica merge-junção híbrida, que combina índices com merge-junção. Suponha que uma das relações esteja classificada; a outra não está classificada, mas tem um índice árvore-B⁺ secundário nos atributos de junção. O algoritmo *merge-junção híbrido* faz o merge da relação classificada com as entradas das folhas do índice árvore-B⁺. O arquivo de resultado contém tuplas da relação classificada e endereços para as tuplas da relação não classificada. O arquivo de resultado é ordenado, então, nos endereços das tuplas da relação não classificada, possibilitando a recuperação eficiente das tuplas correspondentes, na ordem física de armazenamento, para completar a junção. Extensões da técnica com o objetivo de tratar duas relações não classificadas ficam como exercício.

12.6.6 Hash-Junção

Tal como o algoritmo merge-junção, o algoritmo hash-junção pode ser usado para implementar as junções naturais e as equi-joins. No algoritmo hash-junção, uma função hash h é usada para particionar as tuplas de ambas as relações. A idéia básica é dividir as tuplas das relações em conjuntos que têm o mesmo valor hash nos atributos de junção.

Vamos supor que:

- h seja uma função hash que faz o mapeamento dos valores de *AtribJunção* para $\{0, 1, \dots, max\}$, em que *Atribjunção* denota os atributos comuns de r e s usados na junção natural.
- $H_{r_0}, H_{r_1}, \dots, H_{r_{max}}$ denote as partições das tuplas de r , cada uma inicialmente vazia. Cada tupla $t_r \in r$ é colocada na partição H_{r_i} , em que $i = h(t_r, [AtribJunção])$.
- $H_{s_0}, H_{s_1}, \dots, H_{s_{max}}$ denote as partições das tuplas de s , cada uma inicialmente vazia. Cada tupla $t_s \in s$ é colocada na partição H_{s_i} , em que $i = h(t_s, [AtribJunção])$.

A função hash h deveria ter as propriedades randômica e de uniformidade que discutimos no Capítulo 11. O particionamento das relações é mostrado graficamente na Figura 12.8.

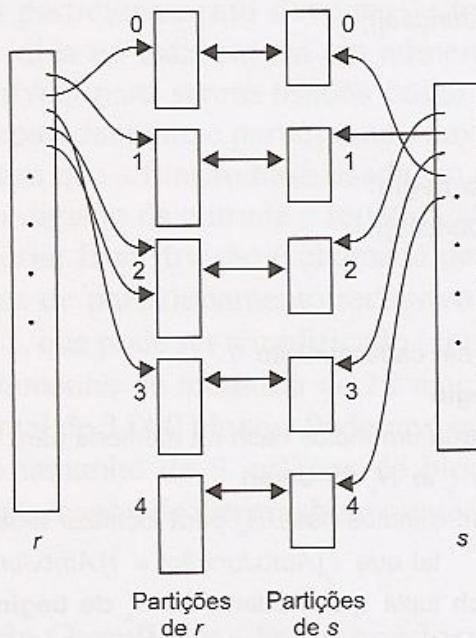


Figura 12.8 Partições hash das relações.

A idéia por trás do algoritmo hash-junção é explicada a seguir. Suponha que uma tupla de r e uma tupla de s satisfazem a condição de junção; então, elas terão o mesmo valor para os atributos de junção.

Se aquele valor aplicado a uma função hash retorna um valor i , a tupla de r tem de estar em H_{r_i} e a tupla de s em H_{s_i} . Portanto, as tuplas de r em H_{r_i} precisam apenas ser comparadas com as tuplas de s em H_{s_i} ; elas não precisam ser comparadas com as tuplas de s em qualquer outra partição.

Por exemplo, se d é uma tupla em *depositante*, c , uma tupla em *cliente* e h , uma função hash nos atributos *nome_cliente* das tuplas, então d e c devem ser testados somente se $h(c) = h(d)$. Se $h(c) \neq h(d)$, então c e d devem ter valores diferentes para *nome_cliente*. Porém, se $h(c) = h(d)$, temos de testar c e d para ver se os valores em seus atributos de junção são iguais, desde que seja possível que c e d tenham *nome_cliente* diferentes e tenham o mesmo valor hash.

A Figura 12.9 mostra os detalhes do algoritmo hash-junção para calcular a junção natural das relações r e s . Assim como no algoritmo merge-junção, $t_r \bowtie t_s$ denota a concatenação dos atributos das tuplas t_r e t_s , seguidos pela projeção dos atributos repetidos. Depois de particionar as relações, o restante do código hash-junção executa uma junção de laço aninhado indexada separada em cada um dos pares de partição i , para $i = 0 \dots max$.

Para fazê-lo, ele primeiro constrói um índice hash em cada H_{s_i} e, então, testa (quer dizer, procura em H_{s_i}) com as tuplas de H_{r_i} . A relação s é a *entrada da construção* e r é a *entrada de teste*.

```

/* Particionando s */
for each tupla  $t_s$  in  $s$  do begin
     $i := h(t_s, [AtribJunção]);$ 
     $H_{si} := H_{si} \cup \{t_s\};$ 
end
/* Particionando r */
for each tupla  $t_r$  in  $r$  do begin
     $i := h(t_r, [AtribJunção]);$ 
     $H_{ri} := H_{ri} \cup \{t_r\};$ 
end
/* Executando a junção em cada partição */
for  $i := 0$  to  $\max$  do begin
    leia  $H_{si}$  e construa um índice hash na memória para ele
    for each tupla  $t_r$  in  $H_{ri}$  do begin
        verifique o índice hash  $H_{si}$  para localizar todas as tuplas  $t_s$ 
        tal que  $t_s[AtribJunção] = t_r[AtribJunção]$ 
        for each tupla  $t_s$  coincidente in  $H_{si}$  do begin
            acrescente  $t_r \bowtie t_s$  ao resultado
        end
    end
end

```

Figura 12.9 Hash-junção.

O índice hash em H_{si} é construído na memória, assim não há necessidade de acessar o disco para recuperar as tuplas. A função hash usada para construir esse índice hash é diferente da função hash h usada anteriormente, mas ainda é aplicada somente aos atributos da junção. No transcorrer da junção de laço aninhado indexada, o sistema usa esse índice hash para recuperar registros que coincidem com os registros na entrada de teste.

As fases de construção e de teste requerem somente um único passo. A extensão do algoritmo de hash-junção para calcular equi-joins genéricas é direta.

O valor \max deve ser grande o suficiente para que, para cada i , as tuplas na partição H_{si} da relação de construção, junto com o índice hash na partição, caibam na memória. Não é necessário que as partições de teste caibam na memória.

Obviamente, é melhor usar a menor relação de entrada no papel de relação de construção. Se o tamanho da relação de construção é b_s blocos, então, para cada uma das \max partições que tiverem tamanho menores ou iguais a M , \max deve ser pelo menos $[b_s/M]$.

Mais precisamente, também temos de contabilizar o espaço adicional ocupado pelo índice hash no particionamento, assim \max deveria ser correspondentemente maior. Por simplicidade, às vezes ignoramos a exigência de espaço do índice hash em nossa análise.

12.6.6.1 Particionamento Recursivo

Se o valor de \max é maior ou igual ao número de frames de página da memória, o particionamento das relações não pode ser feito em um único passo, já que não haverá buffers de páginas suficientes. Em vez disso, o particionamento deve ser feito em passos repetidos. Em um passo, a entrada pode ser dividida no máximo em um número de partições igual ao número de frames de páginas disponíveis para serem usados como buffers de saída. Cada bucket gerado por um passo é lido separadamente e particionado novamente no passo seguinte, para criar partições menores. É claro que a função hash usada em um passo é diferente da função usada no passo anterior. Essa divisão da entrada é repetida até que cada partição da entrada de construção caiba na memória. Essa divisão é chamada de *particionamento recursivo*.

Uma relação não precisa de particionamento recursivo se $M > \max + 1$ ou, de forma equivalente, se $M > (b_s/M) + 1$, que pode ser simplificado (aproximadamente) para $M > \sqrt{b_s}$. Por exemplo, considere um tamanho de memória de 12 megabytes, divididos em blocos de 4 kilobytes; ela conteria um total de 3.000 blocos. Podemos usar uma memória desse tamanho para particionar relações do tamanho de 9 milhões de blocos, que são 36 gigabytes. De forma similar, uma relação do tamanho de um gigabyte requer $\sqrt{250.000}$ blocos ou, aproximadamente, dois megabytes.

12.6.6.2 O Tratamento de Overflows (Transbordamentos)

O *overflow na tabela hash* ocorre na partição i da relação de construção s , se o índice hash H_{s_i} for maior que a memória principal. O overflow na tabela hash pode ocorrer se houver muitas tuplas na relação de construção com os mesmos valores para os atributos de junção, ou se a função hash não tiver as propriedades randômica e de uniformidade. Em qualquer caso, algumas das partições terão mais tuplas que a média, enquanto outras terão menos tuplas; a partição é dita *desbalanceada*.

Podemos ajustar pequenos desbalanceamentos aumentando o número de partições tal que o tamanho esperado de cada partição (inclusive o índice hash na partição) seja um pouco menor que o tamanho da memória. Portanto, o número de partições é aumentado pelo *fudge factor* (*fator de camuflagem*), que normalmente é de aproximadamente 20 por cento do número de partições hash, calculadas conforme descrito anteriormente.

Mesmo se formos conservadores nos tamanhos das partições, por meio da utilização do fudge factor, os overflows ainda podem acontecer. Os overflows da tabela hash podem ser tratados por *correção* ou por *prevenção*. A correção do overflow é executada durante a fase de construção, se um overflow de índice hash é detectado. A correção de overflow é feita da seguinte forma. Se H_{s_i} , para qualquer i , for muito grande, ele é dividido em partições menores usando uma função hash diferente. De forma similar, também particionamos H_{r_i} usando a função hash nova, e a junção só precisa ser feita nas tuplas cujas partições coincidam.

Em contrapartida, a prevenção de overflow cuida do particionamento: que os overflows nunca aconteçam durante a fase de construção. Implementamos a prevenção do overflow por meio do particionamento da relação de construção s inicialmente em muitas partições pequenas e, então, combinamos algumas partições, tal que cada partição combinada caiba na

memória. A relação de teste r deve ser particionada da mesma maneira que as partições combinadas em s , mas os tamanhos de H_r não importam.

Se um número grande de tuplas em s tem o mesmo valor para os atributos de junção, as técnicas de correção e de prevenção podem falhar em algumas partições. Nesse caso, em vez de criar um índice hash na memória e usar uma junção de laço aninhado para a junção das partições, podemos usar outras técnicas de junção nessas partições, como a junção de laço aninhado de blocos.

12.6.6.3 Custo do Hash-Junção

Vamos agora considerar o custo do hash-junção. Nossa análise supõe que não há nenhum overflow de tabela hash. Primeiro, considere o caso em que o particionamento recursivo não é necessário. O particionamento das duas relações r e s exige uma leitura completa de ambas as relações e uma subsequente escrita das mesmas. Essa operação requer $2(b_r + b_s)$ acessos de blocos.

As fases de construção e de teste leram uma vez cada uma das partições e pedem por mais $b_r + b_s$ acessos. O número de blocos ocupados pelas partições poderia ser ligeiramente maior que $b_r + b_s$, devido à existência de blocos parcialmente cheios. O acesso a esses blocos parcialmente cheios pode gerar um overhead de no máximo $2 * max$, já que cada uma das max partições poderia ter um bloco parcialmente cheio que precisa ser escrito e lido. Assim, a estimativa de custo para um hash-junção é:

$$3(b_r + b_s) + 2 * max$$

Agora, considere o caso em que o particionamento recursivo é necessário. Cada passo reduz o tamanho de cada uma das partições por um fator de $M - 1$; essa redução de tamanho é repetida até que cada partição seja no máximo do tamanho de M blocos. O número esperado de passos necessários para particionar s é então $\lceil \log_{M-1}(b_s) - 1 \rceil$. Como, em cada passo, todos os blocos de s são lidos e escritos, o total de transferência de blocos para particionar s é $2b_s \lceil \log_{M-1}(b_s) - 1 \rceil$.

O número de passos para particionar r é igual ao número de passos para particionar s , dando a seguinte estimativa para o custo da junção:

$$2(b_r + b_s) \lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

Por exemplo, considere a junção $cliente \bowtie depositante$. Com um tamanho de memória de 20 blocos, $depositante$ pode ser alocado em 5 partições, cada uma com o tamanho de 20 blocos, tamanho este que caberá na memória. Somente um passo é necessário. De forma similar, a relação $cliente$ é alocada em 5 partições, cada uma com o tamanho de 80. Ignorando o custo de escrita dos blocos parcialmente cheios, o custo é de $3(100 + 400) = 1.500$ transferências de blocos.

O hash-junção pode ser melhorado se o tamanho da memória principal for grande. Quando a entrada de construção inteira pode ser mantida na memória principal, max pode ser fixado em 0; então, o algoritmo hash-junção é executado rapidamente, sem particionar as relações em arquivos temporários, a despeito do tamanho da entrada de teste. A estimativa de custo cai para $b_r + b_s$.

12.6.6.4 Hash-Junção Híbrido

O algoritmo *hash-junção híbrido* executa outra otimização; ele é útil quando o tamanho da memória for relativamente grande, mas a relação de construção não cabe na memória. A fase de particionamento do algoritmo hash-junção precisa de um bloco de memória como um buffer para cada partição que é criada e um bloco de memória como um buffer de entrada. Conseqüentemente, um total de $\max + 1$ blocos de memória são necessários para o particionamento das duas relações. Se a memória é maior que $\max + 1$, podemos usar o resto de memória ($M - \max - 1$ blocos) para servir de buffer para a primeira partição da entrada de construção (ou seja, H_{s_0}), assim ela não precisará ser escrita e relida. Além disso, a função hash é projetada para que o índice hash em H_{s_0} caiba em $M - \max - 1$ blocos, de forma que, ao final do processo de particionamento de s , H_{s_0} esteja completamente na memória e um índice hash possa ser construído em H_{s_0} .

Quando r é particionado, as tuplas em H_{r_0} não são escritas novamente no disco; em vez disso, como elas são geradas, o sistema as utiliza para testar o índice hash H_{s_0} residente na memória e gerar as tuplas de saída da junção. Depois de serem usadas para o teste, as tuplas podem ser descartadas e, assim, a partição H_{r_0} não ocupa nenhum espaço de memória. Assim, um acesso de escrita e um acesso de leitura foram eliminados para cada bloco de H_{r_0} e de H_{s_0} . As tuplas nas outras partições são escritas como sempre, e sofrem a junção posteriormente. A economia proporcionada pelo hash-junção híbrido pode ser significativa se a entrada de construção for apenas um pouco maior que a memória.

Se o tamanho da relação de construção é b_s , \max é aproximadamente igual a b_s/M . Assim, o hash-junção híbrido é muito útil se $M \gg b_s/M$, ou se $M \gg \sqrt{b_s}$, em que a notação \gg denota *muito maior que*. Por exemplo, suponha que o tamanho do bloco seja 4 kilobytes, e a relação de construção seja do tamanho de 1 gigabyte. Então, o algoritmo de hash-junção híbrido é útil se o tamanho de memória é significativamente maior que 2 megabytes; memórias do tamanho de 50 a 100 megabytes ou mais é comum nos computadores atuais.

Considere a junção *cliente* \bowtie *depositante* novamente. Com um tamanho de memória de 25 blocos, *depositante* pode ser particionado em 5 partições, cada uma com o tamanho de 20 blocos, e a primeira das partições da relação de construção pode ser mantida na memória. Esse particionamento ocupa 20 blocos de memória; um bloco é usado para entrada e cada um dos blocos restantes é usado como buffer para as outras 4 partições. A relação *cliente* é particionada de forma semelhante em 5 partições, cada uma do tamanho de 80 blocos; o primeiro desses blocos é usado para o teste, em vez de ser escrito e relido. Ignorando o custo da escrita de blocos parcialmente cheios, o custo é $3(80 + 320) + 20 + 80 = 1.300$ transferências de blocos, em vez das 1.500 transferências de blocos sem a otimização do hash híbrido.

12.6.7 Junções Complexas

Junções de laço aninhado e de laço aninhado por blocos podem ser usadas independentemente das condições de junção. As outras técnicas de junção são mais eficientes que a junção de laço aninhado e suas variantes, mas podem tratar apenas de condições de junção simples, como junção natural ou equi-joins. Podemos implementar junções com condições de junção complexas, como conjunções e disjunções, usando as técnicas de junção eficientes, por meio da aplicação das técnicas desenvolvidas na Seção 12.4.4 para o tratamento de seleções complexas.

Considere a seguinte junção com uma condição conjuntiva:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

Uma ou mais das técnicas de junção descritas anteriormente podem ser aplicáveis às junções nas condições individuais $r \bowtie_{\theta_1} s$, $r \bowtie_{\theta_2} s$, $r \bowtie_{\theta_3} s$ e assim por diante. Podemos calcular a junção global primeiro calculando o resultado de uma dessas junções mais simples $r \bowtie_{\theta_i} s$; cada par de tuplas do resultado intermediário é composto de uma tupla de r e de uma tupla de s .

O resultado da junção completa consiste nas tuplas do resultado intermediário que satisfazem as condições restantes.

$$\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$$

Essas condições podem ser testadas conforme as tuplas em $r \bowtie_{\theta_i} s$ são geradas.

Uma junção cuja condição seja disjuntiva pode ser calculada conforme descrito a seguir. Considere:

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

A junção pode ser calculada como a união dos registros nas junções individuais $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$$

Os algoritmos para calcular a união de relações são descritos na Seção 12.7.

Vamos agora considerar uma junção envolvendo três relações:

$$\text{emprestimo} \bowtie \text{depositante} \bowtie \text{cliente}$$

Não apenas temos a escolha de estratégia para o processamento da junção, mas também temos a escolha de qual junção calcular primeiro.

Há muitas estratégias possíveis a considerar. Descrevemos algumas aqui e deixamos outras como um exercício (Exercício 12.4).

- **Estratégia 1.** Calcule a junção $\text{depositante} \bowtie \text{cliente}$ usando uma das técnicas que apresentamos. Usando o resultado intermediário, calcule:

$$\text{emprestimo} \bowtie (\text{depositante} \bowtie \text{cliente})$$

- **Estratégia 2.** Faça como na Estratégia 1, mas calcule primeiro $\text{emprestimo} \bowtie \text{depositante}$, e, então, faça a junção do resultado com cliente . As junções também podem ser calculadas em outras ordens.
- **Estratégia 3.** Em vez de executar duas junções, execute o par de junções imediatamente. Primeiramente, essa técnica envolve a construção de dois índices:

- Sobre número_emprestimo em emprestimo .
- Sobre nome_cliente em cliente .

A seguir, considere cada tupla t em depositante . Para cada t , procure as tuplas correspondentes em cliente e as tuplas correspondentes em agência . Assim, cada tupla de depositante é examinada exatamente uma vez.

A Estratégia 3 representa uma forma que não foi considerada anteriormente. Ela não corresponde diretamente a uma operação da álgebra relacional. Em vez disso, ela combina duas operações em uma operação de propósito especial. Com essa estratégia, é freqüentemente possível executar uma junção de três relações de forma mais eficaz que com a utilização de duas junções de duas relações. Os custos relativos dependem do modo pelo qual as relações são armazenadas, da distribuição de valores dentro das colunas e da presença de índices. O Exercício 12.5 propõe o cálculo desses custos.

12.7 Outras Operações

Outras operações relacionais e operações relacionais estendidas – como eliminação de duplicidade, projeção, operações de conjuntos, junção externa e agregação – podem ser implementadas conforme esboçado nas Seções 12.7.1 até 12.7.5.

12.7.1 Eliminação de Duplicidade

Podemos facilmente implementar a eliminação de duplicidade usando a classificação. As tuplas idênticas aparecerão adjacentes umas das outras durante a classificação, e todas, exceto uma cópia, podem ser removidas. Com o sort-merge externo, as duplicatas encontradas enquanto um temporário está sendo criado podem ser removidas antes que o temporário seja escrito no disco, reduzindo, assim, o número de transferências de blocos. As duplicatas restantes podem ser eliminadas durante o merge, e o resultado final da classificação não terá duplicata alguma. A estimativa de custo para o pior caso de eliminação de duplicidade é igual à estimativa de custo do pior caso de classificação da relação.

Também podemos implementar a eliminação de duplicidade usando hashing de uma maneira similar ao algoritmo hash-junção. Primeiro, a relação é particionada baseada em uma função hash na tupla inteira. Então, cada partição é lida e um índice hash é construído na memória. Enquanto o índice hash está sendo construído, uma tupla só é inserida se ela não estiver presente. Caso contrário, ela é descartada. Após todas as tuplas na partição serem processadas, as tuplas no índice hash são escritas no resultado. A estimativa de custo é igual ao custo do processamento (particionamento e leitura de cada partição) da relação de construção em um hash-junção.

Por causa do custo relativamente alto da eliminação de duplicidade, as linguagens de consulta comerciais exigem um pedido explícito do usuário para remover duplicatas; caso contrário, as duplicatas são mantidas.

12.7.2 Projeção

Podemos implementar a projeção por meio da execução da projeção em cada tupla, o que resulta em uma relação que poderia ter registros duplicados, e então remover os registros duplicados. A eliminação de duplicidade pode ser feita conforme descrito na Seção 12.7.1. Se os atributos na lista de projeção incluem uma chave da relação, nenhuma duplicata existirá; consequentemente, a eliminação de duplicidade não é necessária. Uma projeção generalizada (que foi discutida na Seção 3.5.1) pode ser implementada da mesma maneira que a projeção. O tamanho de uma projeção da forma $\Pi_A(r)$ é calculado como $V(A, r)$, uma vez que a projeção elimina as duplicatas.

12.7.3 Operações de Conjunto

Podemos implementar as *operações de união, interseção e diferença de conjuntos* primeiro por meio da classificação de ambas as relações e, então, fazendo a varredura uma vez em cada uma das relações ordenadas para obter o resultado. Em $r \cup s$, quando uma varredura concorrente em ambas as relações revela a mesma tupla em ambos os arquivos, uma única tupla é mantida.

O resultado de $r \cap s$ conterá somente as tuplas que aparecem em ambas as relações. Implementamos a *diferença de conjuntos*, $r - s$, de forma semelhante, mantendo as tuplas em r apenas se elas não existirem em s .

Para todas essas operações, apenas uma varredura das duas relações de entrada é necessária, assim o custo é $b_r + b_s$. Se as relações não estão classificadas inicialmente, o custo da classificação tem de ser incluído. Qualquer ordem de classificação pode ser usada na avaliação de operações de conjuntos, contanto que ambas as entradas tenham a mesma ordem de classificação.

O uso da função hash provê outro modo para implementar essas operações de conjuntos. O primeiro passo em cada caso é particionar as duas relações usando a mesma função hash e assim criar as partições $H_{r_0}, \dots, H_{r_{max}}$ e $H_{s_0}, \dots, H_{s_{max}}$. Então, é realizado o seguinte para cada partição $i = 0 \dots max$:

- $r \cup s$
 1. Construa um índice hash na memória em H_{r_i} .
 2. Adicione as tuplas em H_{s_i} ao índice hash somente se elas ainda não estiverem presentes.
 3. Adicione as tuplas do índice hash ao resultado.
- $r \cap s$
 1. Construa um índice hash na memória em H_{r_i} .
 2. Para cada tupla em H_{s_i} , teste o índice hash e adicione a tupla ao resultado somente se ela já estiver presente no índice hash.
- $r - s$
 1. Construa um índice hash na memória em H_{r_i} .
 2. Para cada tupla em H_{s_i} , teste o índice hash e, se a tupla estiver presente no índice hash, remova-a do índice hash.
 3. Adicione as tuplas restantes do índice hash ao resultado.

12.7.4 Junção Externa

Vamos retomar as *operações de junção externa* descritas na Seção 3.5.2. Por exemplo, a junção natural externa esquerda $cliente \bowtie depositante$ contém a junção de $cliente$ e $depositante$ e, adicionalmente, para cada tupla $cliente$ t que não tenha nenhuma tupla coincidente em $depositante$ (ou seja, em que $nome_cliente$ não está em $depositante$), a tupla t_1 é adicionada ao resultado. Para todos os atributos no esquema de $cliente$, a tupla t_1 tem os mesmos valores que a tupla t . Os atributos restantes (do esquema de $depositante$) da tupla t contêm o valor nulo.

Podemos implementar as operações de junção externa usando uma entre duas estratégias. A primeira é calcular a junção correspondente e, então, adicionar tuplas ao resultado da junção para obter o resultado da junção externa. Considere a operação de junção externa esquerda e duas relações: $r(R)$ e $s(S)$.

Para avaliar $r \bowtie_0 s$, primeiro calculamos $r \bowtie_0 s$ e salvamos o resultado como uma relação temporária q_1 . A seguir, calculamos $r - \Pi_r(q_1)$, que dá as tuplas em r que não participaram da junção. Podemos usar qualquer um dos algoritmos para calcular as junções, projeção e diferença de conjuntos descritas anteriormente para calcular a junção externa. Preenchemos cada uma dessas tuplas com valores nulos para atributos de s e os acrescentamos em q para obter o resultado da junção externa.

A operação de junção externa direita $r \bowtie_0 s$ é equivalente a $r \bowtie_0 s$, e pode, portanto, ser implementada de forma simétrica à junção externa esquerda. Podemos implementar a operação de junção externa completa $r \bowtie_0 s$ por meio do cálculo da junção $r \bowtie s$ e, então, acrescentar as tuplas adicionais das operações de junção externa esquerda e direita, conforme descrito anteriormente.

A segunda estratégia para implementar as junções externas é por meio da modificação dos algoritmos de junção. É fácil estender os algoritmos de junção de laço aninhado para calcular a junção externa esquerda: as tuplas na relação externa que não coincidem com nenhuma tupla na relação interna são escritas no resultado depois de serem preenchidas com valores nulos. Entretanto, é difícil estender a junção de laço aninhado para calcular a junção externa completa.

As junções externas naturais e as junções externas com uma condição equi-join podem ser calculadas como extensões dos algoritmos merge-sort e hash-junção. O merge-sort pode ser estendido para calcular a junção externa completa conforme descrito a seguir. Quando o merge de duas relações está sendo feito, as tuplas de qualquer uma das relações que não coincidem com nenhuma tupla da outra relação podem ser preenchidas com valores nulos e escritas no resultado. De forma similar, podemos estender a merge-junção para calcular as junções externas esquerda e direita por meio da escrita das tuplas não-coincidentes (preenchidas com nulos) de apenas uma das relações. Como as relações são ordenadas, é fácil descobrir se uma tupla coincide ou não com quaisquer tuplas da outra relação. Por exemplo, quando uma merge-junção de *cliente* e *depositante* é feita, as tuplas são lidas na ordem de *nome_cliente*, e é fácil conferir, para cada tupla, se há uma tupla coincidente na outra relação.

As estimativas de custo para implementar as junções externas usando o algoritmo de merge-junção são as mesmas das junções correspondentes. A única diferença está no tamanho do resultado e, portanto, nas transferências de bloco para as escritas que não foram contadas em nossas estimativas de custo anteriores.

A extensão do algoritmo de hash-junção para calcular a junção externa fica como exercício (Exercício 12.17).

12.7.5 Agregação

Lembre-se do operador de agregação \mathcal{G} discutido na Seção 3.5.3. Por exemplo, a operação:

$$\text{nome_agência} \mathcal{G} \text{ sum(saldo)} (\text{conta})$$

agrupa as tuplas de *conta* por *nome_agência* e calcula o saldo total de todas as contas em cada agência.

A operação de agregação pode ser implementada, até certo ponto, de forma semelhante à eliminação de duplicidade. Usamos a classificação ou o hash da mesma maneira que fizemos na eliminação de duplicidade, mas baseado no atributo de agregação (*nome_agência* no exemplo anterior). Entretanto, em vez de eliminar as tuplas com o mesmo valor no atributo de agregação, elas são agrupadas, e aplicamos as operações de agregação a cada grupo para obter o resultado.

O tamanho de ${}_A \mathcal{G}_F(r)$ é simplesmente $V(A, r)$, já que há uma tupla em ${}_A \mathcal{G}_F(r)$ para cada valor distinto de A . A estimativa de custo para implementar a operação de agregação é igual ao custo para a eliminação de duplicidade, para as funções de agregação **min**, **max**, **sum**, **count** e **avg**.

Em vez de reunir todas as tuplas em um grupo e, então, aplicar as operações de agregação, podemos implementar as operações de agregação **min**, **max**, **sum**, **count** e **avg** enquanto os grupos estão sendo construídos.

Para o caso de **sum**, **min** e **max**, quando duas tuplas são encontradas no mesmo grupo, elas são substituídas por uma única tupla contendo **sum**, **min** e **max**, respectivamente, das colunas sendo agregadas. Para a operação **count**, um contador é mantido para cada grupo para o qual uma tupla foi encontrada. Finalmente, implementamos a operação **avg** calculando a soma e a quantidade de valores e dividindo a soma pelo número de valores para obter a média.

Se as tuplas $V(A, r)$ do resultado caberão na memória, tanto a implementação baseada na classificação quanto a implementação baseada em hashing não precisam escrever nenhuma tupla no disco. Conforme as tuplas são lidas, elas podem ser inseridas em uma estrutura de árvore ordenada ou em um índice hash. Quando usamos as técnicas de agregação *fly*, somente uma tupla precisa ser armazenada para cada um dos grupos $V(A, r)$. Conseqüentemente, a estrutura de árvore ordenada ou o índice hash caberá na memória e a agregação pode ser processada com apenas b , transferências de blocos, em vez das $3b$, transferências que seriam necessárias.

12.8 Avaliação de Expressões

Até agora, estudamos como operações relacionais individuais são realizadas. Agora, consideraremos como avaliar uma expressão que contém operações múltiplas. A maneira óbvia de avaliar uma expressão seria avaliar uma operação por vez, em uma ordem apropriada. O resultado de cada avaliação é *materializado* em uma relação temporária para ser usada na seqüência. Uma desvantagem nessa abordagem é a necessidade de construir relações temporárias, as quais (a menos que sejam pequenas) devem ser escritas no disco.

Uma abordagem alternativa é avaliar várias operações simultaneamente em um *pipeline*, com os resultados de uma operação sendo passados para a próxima, sem a necessidade de armazenar uma relação temporária.

Nas Seções 12.8.1 e 12.8.2, consideramos as abordagens de *materialização* e de *pipeline*. Veremos que os custos dessas abordagens podem diferir substancialmente, mas veremos também que há casos em que somente a abordagem de *materialização* é possível.

12.8.1 Materialização

É mais fácil entender intuitivamente como avaliar uma expressão observando uma representação gráfica da expressão em um *operador árvore*. Considere a expressão:

$$\Pi_{\text{nome_cliente}} (\sigma_{\text{saldo} < 2500} (\text{conta}) \bowtie \text{cliente})$$

mostrada na Figura 12.10.

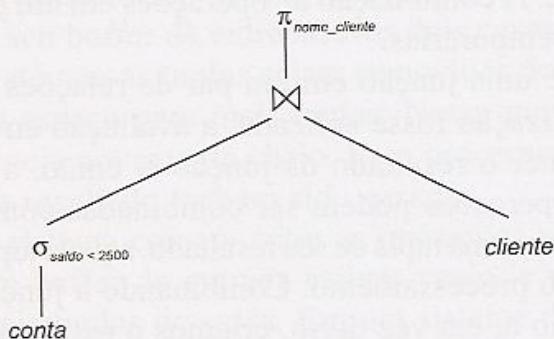


Figura 12.10 Representação gráfica de uma expressão.

Se aplicamos a abordagem de materialização, começamos com as operações de nível mais baixo (na parte mais baixa da árvore). Em nosso exemplo, só há uma operação: a operação de seleção em *conta*. As entradas para as operações de nível mais baixo são as relações no banco de dados. Executamos essas operações usando os algoritmos que estudamos anteriormente e armazenamos os resultados em relações temporárias. Podemos usar essas relações temporárias para executar as operações no nível acima na árvore, em que agora as entradas são as relações temporárias ou as relações armazenadas no banco de dados. Em nosso exemplo, as entradas para a junção são a relação *cliente* e a relação temporária criada pela seleção em *conta*. A junção pode ser avaliada agora criando outra relação temporária.

Repetindo o processo, eventualmente avaliaremos a operação na raiz da árvore, dando o resultado final da expressão. Em nosso exemplo, obtemos o resultado final executando a operação de projeção que está na raiz da árvore, usando como entrada a relação temporária criada pela junção.

A avaliação que acabamos de descrever é chamada de *avaliação materializada*, já que os resultados de cada operação intermediária são criados (materializados) e, então, usados na avaliação das operações do próximo nível.

O custo de uma avaliação materializada não é simplesmente a soma dos custos das operações envolvidas. Quando calculamos o custo estimado dos algoritmos, ignoramos o custo de escrita do resultado da operação no disco. Para calcular o custo da avaliação de uma expressão conforme feito aqui, temos de somar os custos de todas as operações, assim como o custo de escrita dos resultados intermediários no disco. Supomos que os registros do resultado são colocados em um buffer e, quando o buffer está cheio, são escritos no disco. O custo de escrita do resultado pode ser estimado como $n_r f_r$, em que n_r é o número estimado de tuplas na relação de resultado e f_r é o fator de bloco da relação de resultado. Dobrando o buffer (usando dois buffers, um dando prosseguimento à execução do algoritmo

enquanto o outro está sendo escrito no disco) é permitido ao algoritmo executar mais depressa por meio da realização da atividade da CPU em paralelo com atividade de I/O.

12.8.2 Pipelining

Podemos melhorar a eficiência da avaliação de consultas reduzindo o número de arquivos temporários produzidos. Alcançamos essa redução combinando várias operações relacionais em um *pipeline* de operações em que os resultados de uma operação são passados para a próxima operação no pipeline. A combinação de operações em um pipeline elimina o custo de leitura e escrita de relações temporárias.

Por exemplo, considere uma junção em um par de relações seguida de uma projeção ($\Pi_{a_1, a_2}(r \bowtie s)$). Se a materialização fosse aplicada, a avaliação envolveria a criação de uma relação temporária para manter o resultado da junção e, então, a leitura do resultado para executar a projeção. Essas operações podem ser combinadas conforme descrição a seguir. Quando a operação de junção gera uma tupla de seu resultado, aquela tupla é passada imediatamente à operação de projeção para o processamento. Combinando a junção e a projeção, evitamos criar o resultado intermediário e, em vez disso, criamos o resultado final diretamente.

12.8.2.1 Implementação de Pipelining

Podemos implementar um pipeline construindo uma única e complexa operação que combina as operações que constituem o pipeline. Embora essa abordagem seja possível para várias situações que ocorrem com freqüência, em geral, é desejável usar novamente o código para as operações individuais na construção de um pipeline. Portanto, cada operação no pipeline é modelada como um processo separado ou encaixada no sistema, que toma um fluxo de tuplas da entrada do pipeline e gera um fluxo de tuplas como resultado. Para cada par de operações adjacentes no pipeline, um buffer é criado para manter as tuplas que são passadas de uma operação para a próxima.

No exemplo da Figura 12.10, todas as três operações podem ser colocadas em um pipeline, no qual os resultados da seleção são passados para a junção assim que gerados. Por sua vez, os resultados da junção são passados para a projeção assim que são gerados. As exigências de memória são baixas, pois os resultados de uma operação não são armazenados por muito tempo. Porém, como resultado do pipelining, as entradas para as operações não estão todas disponíveis de uma só vez para o processamento.

Os pipelines podem ser executados em um dos dois modos:

1. Dirigido por demanda.
2. Dirigido pelo produtor.

No pipeline *dirigido por demanda*, o sistema faz repetidas solicitações de tuplas à operação no topo do pipeline. Cada vez que uma operação recebe uma solicitação de tuplas, ela calcula a próxima tupla (ou tuplas) a ser retornada e, então, retorna a tupla. Se a entrada da operação não estiver no pipeline, a(s) próxima(s) tupla(s) a ser(em) retornada(s) pode(m) ser calculada(s) a partir das relações de entrada, sendo mantido o que foi retornado até então. Se alguma de suas entradas estiver no pipeline, a operação também faz solicitações de

tuplas para suas entradas no pipeline. Usando as tuplas recebidas de suas entradas no pipeline, a operação calcula as tuplas de seu resultado e as passa a seu pai.

Em um pipeline *dirigido pelo produtor*, as operações não esperam solicitações para produzir tuplas, mas, em vez disso, geram tuplas “ansiosamente”. Cada operação no nível mais baixo do pipeline gera tuplas de resultado continuamente e as põe em seu resultado até o buffer ficar cheio. Uma operação em qualquer outro nível do pipeline gera tuplas de resultado quando ela obtém tuplas de entrada das operações dos níveis mais baixos no pipeline, até que seu buffer de saída esteja cheio. Uma vez que a operação usa uma tupla de uma entrada no pipeline, ela a remove de seu buffer de entrada. Nos dois casos, uma vez que o buffer esteja cheio, a operação espera até que as tuplas sejam removidas do buffer pela operação pai, de forma que o buffer tenha espaço para mais tuplas. Nesse momento, a operação gera mais tuplas, até que o buffer esteja novamente cheio. Esse processo é repetido por uma operação até que todas as tuplas de resultado tenham sido geradas.

É necessário que o sistema comute entre as operações somente quando um buffer de saída estiver cheio ou um buffer de entrada estiver vazio, e mais tuplas de entrada forem necessárias para gerar mais tuplas de saída. Em um sistema de processamento paralelo, as operações em um pipeline podem ser executadas de forma concorrente em processadores distintos (veja Capítulo 17).

A utilização do pipeline dirigido pelo produtor pode ser vista como um *empurrão*, de baixo para cima, nos dados em uma árvore de operações, enquanto a utilização do pipeline dirigido por demanda pode ser vista como sendo um *pxuão* de dados para cima, de uma árvore de operações a partir de seu topo. Cada operação em um pipeline dirigido por demanda pode ser implementada como um *iterator*, que provê as seguintes funções: *open*, *next* e *close*. Depois de uma chamada de *open*, cada chamada de *next* retorna a próxima tupla de resultado da operação. Por sua vez, a implementação da operação faz uma chamada de *next* sobre suas entradas, para obter suas tuplas de entrada quando necessário. A operação *close* indica a um iterator que não é necessária mais nenhuma tupla. O iterator mantém o *estado* de sua execução entre as chamadas, de forma que sucessivas chamadas de *next* recebem as tuplas de resultado sucessivas. Os detalhes da implementação de um iterator são deixados como exercício (Exercício 12.18). O pipeline dirigido por demanda é usado com mais freqüência que o pipeline dirigido pelo produtor porque é mais fácil de implementar.

12.8.2.2 Avaliação de Algoritmos para Pipelining

Considere uma operação de junção cuja entrada à esquerda está no pipeline. Como ela está no pipeline, a entrada não se encontra toda disponível de uma vez para o processamento da operação de junção. Essa indisponibilidade limita a escolha de um algoritmo de junção para ser usado. Por exemplo, o algoritmo merge-junção não pode ser usado se a entrada não estiver ordenada, já que não é possível ordenar uma relação até que todas as tuplas estejam disponíveis – assim, de fato, transformando o pipelining em materialização. Entretanto, a junção de laço aninhado indexada pode ser usada: conforme as tuplas são recebidas para o lado à esquerda da junção, elas podem ser usadas para indexar a relação do lado direito e para gerar as tuplas do resultado da junção. Esse exemplo ilustra que as escolhas relativas ao algoritmo usado para uma operação e as escolhas relativas ao pipelining não são independentes.

As restrições nos algoritmos de avaliação que podem ser usados são um fator de limitação para a técnica de pipelining. Como resultado, apesar das vantagens aparentes de pipelining, há casos em que a materialização alcança um custo global mais baixo. Suponha que a junção de r e s seja solicitada e a entrada r está em um pipeline. Se a junção de laço aninhado indexada é usada para apoiar o pipelining, pode ser necessário um acesso de disco para cada tupla na relação de entrada do pipeline. O custo dessa técnica é $n_r * HT_r$, em que HT_r é a altura do índice em s . Com a materialização, o custo de escrita de r seria b_r . Com uma técnica hash-junção pode ser possível executar a junção com um custo aproximado de $3(b_r + b_s)$. Se n_r é substancialmente maior que $4b_r + 3b_s$, a materialização seria mais barata.

O uso efetivo de pipelining requer o uso de algoritmos de avaliação que podem gerar tuplas de resultado mesmo enquanto as tuplas estão sendo recebidas pelas entradas da operação. Podemos distinguir dois casos:

1. Apenas uma das entradas de uma junção está no pipeline.
2. Ambas as entradas da junção estão no pipeline.

Se uma única entrada de uma junção está no pipeline, a junção de laço aninhado indexada é uma escolha natural. Se as tuplas de entrada que se encontram no pipeline estão ordenadas nos atributos de junção, e a condição de junção é uma equi-join, o merge-join também pode ser usado. O hash-junção híbrido pode ser usado com a entrada que está no pipeline como relação de teste. Porém, as tuplas que não estão na primeira partição só serão processadas depois que a relação de entrada que está no pipeline for inteiramente recebida. O hash-junção híbrido é útil se a entrada que não está no pipeline couber completamente na memória ou se pelo menos a maior parte dessa entrada couber na memória.

Se ambas as entradas estão no pipeline, a escolha de algoritmos de junção é mais restrita. Se ambas as entradas estão ordenadas no atributo de junção, e a condição de junção é uma equi-join, a merge-junção pode ser usada. Outra alternativa é a técnica de *pipelined-junção*, mostrada na Figura 12.11. O algoritmo supõe que as tuplas de entrada para ambas as relações de entrada, r e s , estão no pipeline. As tuplas disponíveis para ambas as relações são colocadas em uma fila única para o processamento. Entradas de fila especiais, chamadas de End_r e End_s , que servem como marcadores de fim de arquivo, são inseridas na fila após todas as tuplas de r e s (respectivamente) serem geradas. Para uma avaliação eficiente, deveriam ser construídos índices apropriados nas relações r e s . Conforme as tuplas são adicionadas a r e s , os índices devem ser mantidos atualizados.

12.9 Transformação de Expressões Relacionais

Até o momento, estudamos algoritmos para avaliar operações da álgebra relacional estendida e estimar seus custos. Como mencionamos anteriormente, uma consulta pode ser expressa de diversas maneiras diferentes, com diferentes custos de avaliação. Nesta seção, em vez de usar a expressão relacional da forma fornecida, consideraremos expressões alternativas equivalentes.

```

    acabou_r := falso;
    acabou_s := falso;
    r := Ø;
    s := Ø;
    resultado := Ø;
    while not acabou_r or not acabou_s do
        begin
            if fila está vazia then espere até que a fila não esteja vazia;
            t := entrada no topo da fila;
            if t = End, then acabou_r := verdadeiro
            else if t = End_s then acabou_s := verdadeiro
            else if t é da entrada r then
                begin
                    r := r ∪ {t};
                    resultado := resultado ∪ ({t} ⋙ s);
                end
            else /* t é da entrada s */
                begin
                    s := s ∪ {t};
                    resultado := resultado ∪ (r ⋙ {t});
                end
        end
    end

```

Figura 12.11 Algoritmo de pipelined-junção.

12.9.1 Equivalência de Expressões

Considere a expressão da álgebra relacional para a consulta “encontre os nomes de todos os clientes que possuem uma conta em qualquer agência localizada no Brooklyn”.

$$\Pi_{\text{nome_cliente}} (\sigma_{\text{cidade_agência} = \text{"Brooklyn"}} (\text{agência} \bowtie (\text{conta} \bowtie \text{depositante})))$$

Essa expressão constrói uma relação intermediária grande, $\text{agência} \bowtie \text{conta} \bowtie \text{depositante}$. Entretanto, estamos interessados em apenas algumas tuplas dessa relação (aqueles pertencentes às agências localizadas no Brooklyn) e em apenas um dos seis atributos dessa relação. Como estamos preocupados apenas com aquelas tuplas na relação agência que pertencem a agências localizadas no Brooklyn, não precisamos considerar aquelas tuplas que não têm $\text{cidade_agência} = \text{"Brooklyn"}$. Reduzindo o número de tuplas da relação agência a que precisamos ter acesso, reduzimos o tamanho do resultado intermediário. Nossa consulta é representada agora pela expressão da álgebra relacional:

$$\Pi_{\text{nome_cliente}} ((\sigma_{\text{cidade_agência} = \text{"Brooklyn"}} (\text{agência})) \bowtie (\text{conta} \bowtie \text{depositante}))$$

que é equivalente a nossa expressão algébrica original, mas que gera relações intermediárias menores. A expressão inicial e a expressão transformada são mostradas graficamente na Figura 12.12.

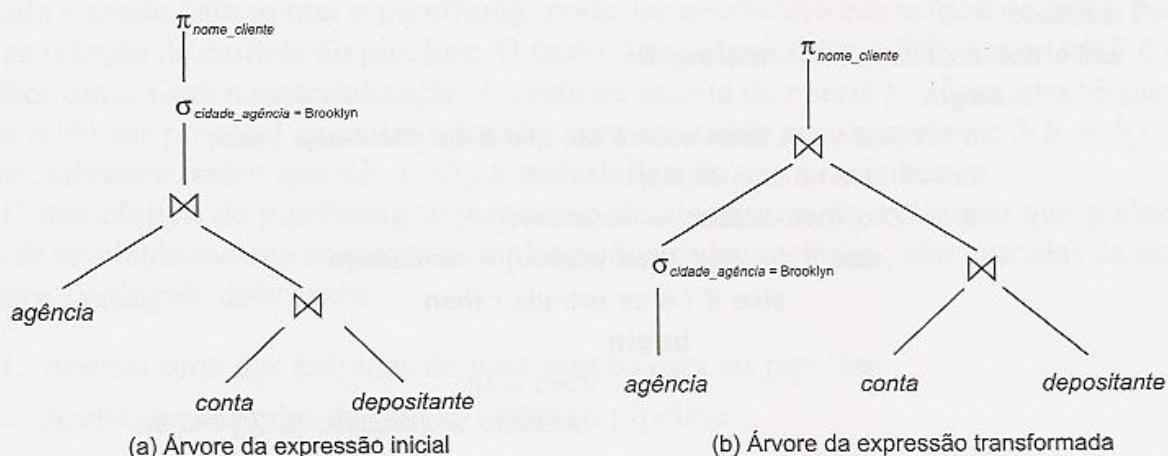


Figura 12.12 Expressões equivalentes.

Dada uma expressão da álgebra relacional, é função do otimizador de consulta propor um plano de avaliação da consulta que gere o mesmo resultado da expressão fornecida e que seja uma maneira menos onerosa de gerar o resultado (ou que, pelo menos, não seja muito mais cara que a maneira mais barata).

Para encontrar o plano de avaliação de consulta menos caro, o otimizador precisa gerar planos alternativos que produzam o mesmo resultado da expressão dada e escolher o plano menos caro.

A geração de planos de avaliação de consultas envolve dois passos: (1) geração de expressões que são logicamente equivalentes à expressão dada e (2) escrever as expressões resultantes de maneiras alternativas para gerar planos de avaliação alternativos. Os dois passos são intercalados no otimizador de consultas – algumas expressões são geradas e escritas, então expressões adicionais são geradas e escritas, e assim por diante.

Para implementar o primeiro passo, o otimizador de consultas tem de gerar expressões equivalentes para uma determinada expressão. Isto é feito por meio de *regras de equivalência* que especificam como transformar uma expressão em outra logicamente equivalente. Descrevemos essas regras a seguir.

Na Seção 12.10, descrevemos como escolher um plano de avaliação de consulta. Podemos fazer a escolha com base no custo *estimado* dos planos. Considerando que o custo é uma estimativa, o plano selecionado não é necessariamente o menos caro; porém, contanto que as estimativas sejam boas, é provável que o plano seja o menos caro ou não muito mais caro que ele. Essa otimização é chamada de *otimização baseada no custo*, descrita na Seção 12.10.2.

12.9.2 Regras de Equivalência

Uma *regra de equivalência* diz que expressões de duas formas são equivalentes: podemos transformar uma na outra preservando a equivalência. *Preservar a equivalência* significa que as relações geradas pelas duas expressões têm o mesmo conjunto de atributos e contêm o mesmo conjunto de tuplas, embora seus atributos possam estar ordenados de forma diferente. As regras de equivalência são usadas pelo otimizador para transformar expressões em outras logicamente equivalentes.

Agora, listamos algumas regras genéricas de equivalência em expressões da álgebra relacional. Algumas das equivalências listadas também são ilustradas na Figura 12.13. Usamos θ , θ_1 , θ_2 e assim por diante para denotar predicados; usamos L_1 , L_2 , L_3 , e assim por diante para denotar listas de atributos; e E , E_1 , E_2 e assim por diante para denotar expressões da álgebra relacional. Uma relação de nome r é simplesmente um caso especial de uma expressão da álgebra relacional, e pode ser usada onde quer que E apareça.

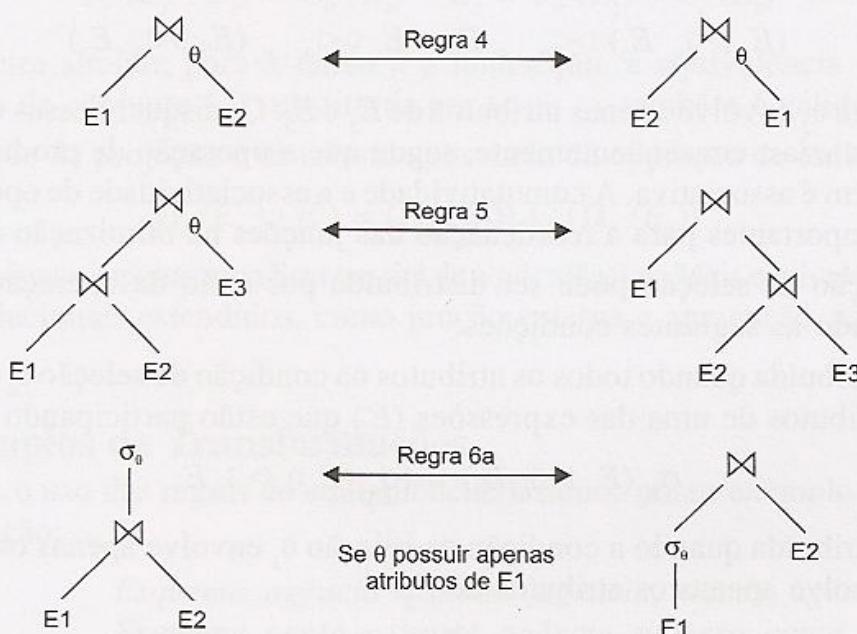


Figura 12.13 Representação gráfica de equivalências.

1. Operações de seleções conjuntivas podem ser quebradas em uma seqüência de seleções individuais. Essa transformação é chamada de cascata de σ .

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Operações de seleção são comutativas.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Apenas as operações finais em uma seqüência de operações de projeção são necessárias, as outras podem ser omitidas. Essa transformação também pode ser chamada de cascata de Π .

$$\Pi_{L1}(\Pi_{L2}(\dots(\Pi_{Ln}(E)\dots))) = \Pi_{L1}(E)$$

4. Seleções podem ser combinadas com produtos cartesianos e junções teta.

a. $\sigma_{\theta_0}(E_1 \times E_2) = E_1 \bowtie_{\theta_0} E_2$

Essa expressão é exatamente a definição da junção teta.

b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_1} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Operações junção teta são comutativas.

$$E_1 \bowtie_{\theta_0} E_2 = E_2 \bowtie_{\theta_0} E_1$$

Lembre-se de que o operador de junção natural é simplesmente um caso especial do operador de junção teta; consequentemente, a junção natural também é comutativa.

6. a. Operações de junção natural são associativas.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b. A junção teta é associativa da seguinte maneira:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

em que θ_2 envolve apenas atributos de E_2 e E_3 . Quaisquer dessas condições podem estar vazias; consequentemente, segue que a operação de produto cartesiano (\times) também é associativa. A comutatividade e a associatividade de operações de junção são importantes para a reordenação das junções na otimização de consultas.

7. A operação de seleção pode ser distribuída por meio da operação de junção teta, observando as seguintes condições:

a. É distribuída quando todos os atributos na condição de seleção θ_0 envolvem apenas os atributos de uma das expressões (E_i) que estão participando da junção.

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta_0} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta_0} E_2$$

b. É distribuída quando a condição de seleção θ_1 envolve apenas os atributos de E_1 e θ_2 envolve apenas os atributos de E_2 .

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_0} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_0} (\sigma_{\theta_2}(E_2))$$

8. A operação de projeção é distribuída por meio da operação de junção teta.

a. Sejam L_1 e L_2 atributos de E_1 e E_2 , respectivamente. Suponha que a condição de junção θ envolva somente atributos em $L_1 \cup L_2$. Então:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

b. Considere uma junção $E_1 \bowtie_{\theta} E_2$. Sejam L_1 e L_2 conjuntos de atributos de E_1 e E_2 , respectivamente. Seja L_3 atributos de E_1 que estão envolvidos na condição de junção θ , mas que não estão em $L_1 \cup L_2$, e seja L_4 atributos de E_2 que estão envolvidos na condição de junção θ , mas que não estão em $L_1 \cup L_2$. Então:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

9. As operações de conjunto união e interseção são comutativas.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

A diferença de conjunto não é comutativa.

10. A união e a interseção de conjuntos são associativas.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. A operação de seleção é distribuída por meio das operações de união, interseção e diferença de conjuntos.

$$\sigma_p(E_1 - E_2) = \sigma_p(E_1) - E_2 = \sigma_p(E_1) - \sigma_p(E_2)$$

De maneira similar, para a união e a interseção, a equivalência precedente, com diferença de conjunto ($-$) substituída por \cup ou \cap , também é válida.

12. A operação de projeção é distribuída por meio da operação de união.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

A lista precedente é apenas uma lista parcial de equivalências. Mais equivalências envolvendo os operadores relacionais estendidos, como junção externa e agregação, são discutidas nos exercícios.

12.9.3 Exemplos de Transformações

Agora ilustramos o uso das regras de equivalência. Usamos nosso exemplo bancário com os esquemas de relação:

Esquema_agência = (*nome_agência*, *cidade_agência*, *fundos*)

Esquema_conta = (*nome_agência*, *número_conta*, *saldo*)

Esquema_depositante = (*nome_cliente*, *número_conta*)

As relações *agência*, *conta* e *depositante* são instâncias desses esquemas.

Em nosso exemplo anterior, a expressão:

$$\Pi_{\text{nome_cliente}} (\sigma_{\text{cidade_agência} = \text{"Brooklyn"}} (\text{agência} \bowtie (\text{conta} \bowtie \text{depositante})))$$

foi transformada na seguinte expressão:

$$\Pi_{\text{nome_cliente}} ((\sigma_{\text{cidade_agência} = \text{"Brooklyn"}} (\text{agência})) \bowtie (\text{conta} \bowtie \text{depositante}))$$

que é equivalente a nossa expressão algébrica original, mas gera relações intermediárias menores. Podemos realizar essa transformação usando a regra 7a. Lembre-se de que a regra diz somente que as duas expressões são equivalentes; ela não diz que uma é melhor que a outra.

Regras de equivalência múltiplas podem ser usadas, uma após a outra, em uma consulta ou em partes da consulta. Como ilustração, suponha que modificamos nossa consulta original para restringi-la a clientes que têm um saldo maior que mil dólares. A nova consulta na álgebra relacional é:

$$\Pi_{\text{nome_cliente}} (\sigma_{\text{cidade_agência} = \text{"Brooklyn"} \wedge \text{saldo} > 1000} (\text{agência} \bowtie (\text{conta} \bowtie \text{depositante})))$$

Não podemos aplicar o predicado de seleção diretamente à relação *agência*, já que o predicado envolve tanto atributos da relação *agência* quanto da relação *conta*. Porém, podemos primeiro aplicar a regra 6a (associatividade da junção natural) para transformar a junção *agência* \bowtie (*conta* \bowtie *depositante*) em $(\text{agência} \bowtie \text{conta}) \bowtie \text{depositante}$:

$$\Pi_{\text{nome_cliente}} (\sigma_{\text{cidade_agência} = \text{"Brooklyn"} \wedge \text{saldo} > 1000} ((\text{agência} \bowtie \text{conta}) \bowtie \text{depositante}))$$

Então, usando a regra 7a, podemos reescrever nossa consulta como:

$$\Pi_{\text{nome_cliente}} ((\sigma_{\text{cidade_agência} = \text{"Brooklyn"} \wedge \text{saldo} > 1000} (\text{agência} \bowtie \text{conta})) \bowtie \text{depositante})$$

Vamos examinar a subexpressão de seleção dentro dessa expressão. Usando a regra 1, podemos quebrar a seleção em duas seleções, para obter a seguinte subexpressão:

$$\sigma_{\text{cidade_agência} = \text{"Brooklyn"} \wedge \text{saldo} > 1000} (\text{agência} \bowtie \text{conta})$$

Ambas as expressões precedentes selecionam tuplas com *cidade_agência* = "Brooklyn" e *saldo* > 1.000. Porém, a última forma da expressão fornece uma oportunidade para aplicar a regra "execute as seleções primeiro", resultando na subexpressão:

$$\sigma_{\text{cidade_agência} = \text{"Brooklyn"} \wedge \text{saldo} > 1000} (\text{agência}) \bowtie \sigma_{\text{saldo} > 1000} (\text{conta})$$

A expressão inicial e a expressão final após todas as transformações precedentes são mostradas graficamente na Figura 12.14. Poderíamos também ter usado a regra 7b para obter a expressão final diretamente, sem usar a regra 1 para quebrar a seleção em duas seleções. De fato, a regra 7b pode ser derivada das regras 1 e 7a.

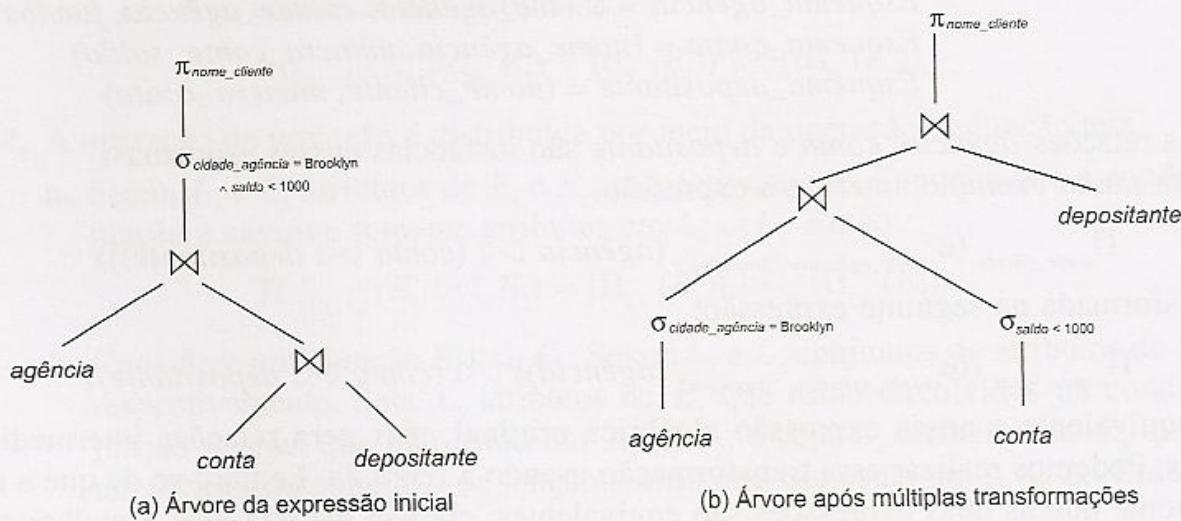


Figura 12.14 Múltiplas transformações.

Um conjunto de regras de equivalência é dito *mínimo* se nenhuma regra puder ser derivada de qualquer combinação das outras. O exemplo precedente ilustra que o conjunto de regras de equivalência na Seção 12.9.2 não é mínimo.

Agora considere a seguinte forma de nossa consulta de exemplo:

$$\Pi_{\text{nome_cliente}} ((\sigma_{\text{cidade_agência} = \text{"Brooklyn"}} (\text{agência}) \bowtie \text{conta}) \bowtie \text{depositante})$$

Com o resultado da subexpressão:

$$(\sigma_{\text{cidade_agência} = \text{"Brooklyn"}, \text{agência}} \bowtie \text{conta})$$

obtemos uma relação cujo esquema é:

$$(\text{nome_agência}, \text{cidade_agência}, \text{fundos}, \text{número_conta}, \text{saldo})$$

Podemos eliminar vários atributos do esquema por meio de projeções, usando as regras 8a e 8b. Os únicos atributos que temos de reter são aqueles que aparecem no resultado da consulta ou que são necessários para processar as operações subsequentes. Eliminando atributos desnecessários, reduzimos o número de colunas do resultado intermediário. Assim, o tamanho do resultado intermediário é reduzido. Em nosso exemplo, o único atributo de que precisamos da junção de *agência* e *conta* é *número_conta*. Então, podemos modificar a expressão para:

$$\Pi_{\text{nome_cliente}} ((\Pi_{\text{número_conta}} ((\sigma_{\text{cidade_agência} = \text{"Brooklyn"}, \text{agência}} \bowtie \text{conta})) \bowtie \text{depositante})$$

A projeção $\Pi_{\text{número_conta}}$ reduz o tamanho dos resultados da junção intermediária.

12.9.4 Ordenando Junções

Uma boa ordenação de operações de junção é importante para reduzir o tamanho dos resultados intermediários; consequentemente, a maioria dos otimizadores de consulta presta muita atenção à ordem das junções. Conforme mencionado no Capítulo 3 e na regra de equivalência 6a, a operação de junção natural é associativa. Assim, para todas as relações r_1 , r_2 e r_3 :

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

Embora essas expressões sejam equivalentes, os custos de computação delas podem diferir. Considere novamente a expressão:

$$\Pi_{\text{nome_cliente}} ((\sigma_{\text{cidade_agência} = \text{"Brooklyn"}, \text{agência}} \bowtie \text{conta}) \bowtie \text{depositante})$$

Poderíamos executar *conta* \bowtie *depositante* primeiro e, então, fazer a junção do resultado com:

$$\sigma_{\text{cidade_agência} = \text{"Brooklyn"}, \text{agência}} (\text{agência})$$

Entretanto, *conta* \bowtie *depositante* provavelmente é uma relação grande, já que contém uma tupla para cada conta. Em contrapartida,

$$\sigma_{\text{cidade_agência} = \text{"Brooklyn"}, \text{agência}} (\text{agência}) \bowtie \text{conta}$$

provavelmente é uma relação pequena. Para confirmar, observamos que, como o banco tem um grande número de agências amplamente distribuídas, é provável que apenas uma fração

pequena dos clientes do banco tenha contas em agências localizadas no Brooklyn. Assim, a expressão precedente resulta em uma tupla para cada conta mantida por um residente do Brooklyn. Então, a relação temporária que precisamos armazenar é menor que a que teríamos obtido se tivéssemos calculado *conta* \bowtie *depositante* primeiro.

Há outras opções a considerar para avaliar nossa consulta. Não nos preocupamos com a ordem na qual os atributos aparecem em uma junção, já que é fácil mudar a ordem antes de exibir o resultado. Assim, para todas as relações r_1 e r_2 :

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

ou seja, a junção natural é comutativa (regra de equivalência 5).

Usando a associatividade e comutatividade da junção natural (regras 5 e 6), podemos reescrever nossa expressão da álgebra relacional:

$$\Pi_{\text{nome_cliente}} (((\sigma_{\text{cidade_agência} = \text{"Brooklyn}}} (\text{agência})) \bowtie \text{depositante}) \bowtie \text{conta})$$

Ou seja, poderíamos fazer:

$$(\sigma_{\text{cidade_agência} = \text{"Brooklyn}}} (\text{agência})) \bowtie \text{depositante}$$

primeiro e, depois disso, fazer a junção do resultado com *conta*. Entretanto, observe que não há nenhum atributo em comum entre *Esquema_agência* e *Esquema_depositante*, assim a junção é exatamente um produto cartesiano. Se há b agências em Brooklyn e d tuplas na relação *depositante*, esse produto cartesiano gera $b * d$ tuplas, uma para cada possível par de tuplas de depositante e agência (independente de a conta em *depositante* ser mantida na agência). Assim, parece que esse produto cartesiano produzirá uma relação temporária grande. Como resultado, rejeitariamnos essa estratégia.

Porém, se o usuário tivesse utilizado a expressão, poderíamos usar a associatividade e a comutatividade da junção natural para transformá-la em uma expressão mais eficiente do que a que usamos anteriormente.

12.9.5 Enumeração de Expressões Equivalentes

Os otimizadores de consulta usam as regras de equivalência para sistematicamente gerar expressões equivalentes para uma determinada expressão de consulta. Conceitualmente, o processo é executado da seguinte forma. Dada uma expressão, se qualquer subexpressão coincide com um dos lados de uma regra de equivalência, uma nova expressão é gerada, na qual a subexpressão é transformada para coincidir com o outro lado da regra. Esse processo continua até que nenhuma outra expressão nova possa ser gerada.

O processo precedente é caro, tanto em termos de espaço quanto de tempo. A exigência de espaço é reduzida da seguinte forma. Se gerarmos uma expressão E_1 a partir de uma expressão E_2 usando uma regra de equivalência, então E_1 e E_2 são semelhantes em sua estrutura e possuem subexpressões idênticas. As técnicas de representação de expressão que permitem que ambas as expressões apontem para subexpressões compartilhadas podem reduzir a necessidade de espaço significativamente, e são usadas em muitos otimizadores de consulta.

Além disso, nem é sempre necessário gerar cada expressão que pode ser gerada usando as regras de equivalência. Se as estimativas de custo da avaliação são levadas em consideração,

um otimizador pode ser capaz de evitar o exame de algumas das expressões, conforme veremos na Seção 12.10. Podemos reduzir o tempo necessário para a otimização usando técnicas como as que acabamos de descrever.

12.10 A Escolha de Planos de Avaliação

A geração de expressões é apenas parte do processo de otimização de consultas. Cada operação na expressão pode ser implementada com diferentes algoritmos. Um plano de avaliação define exatamente que algoritmo é usado para cada operação e como a execução das operações é coordenada. A Figura 12.15 ilustra um possível plano de avaliação para a expressão da Figura 12.14. Conforme já vimos, vários algoritmos diferentes podem ser usados para cada operação relacional, gerando planos de avaliação alternativos. Além disso, as decisões sobre pipelining precisam ser tomadas. Na figura, as arestas das operações de seleção para a operação merge-junção são colocadas no pipeline; o pipeline é possível se as operações de seleção gerarem seus resultados classificados nos atributos de junção. Elas fariam desse modo se os índices para *agência* e *conta* armazenassem registros com valores iguais aos atributos de índice ordenados por *nome_agência*.

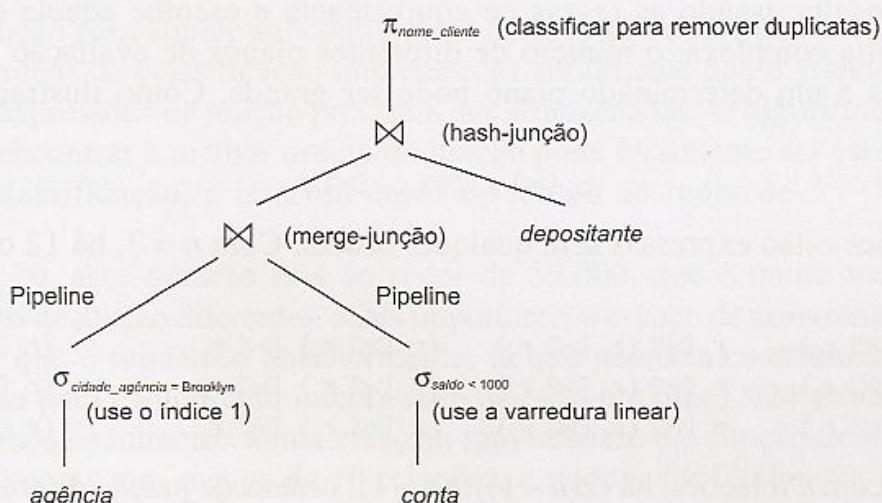


Figura 12.15 Um plano de avaliação.

12.10.1 Interação de Técnicas de Avaliação

Um modo de escolher um plano de avaliação para uma expressão de consulta é simplesmente escolher o algoritmo mais barato para avaliar cada operação. Podemos escolher qualquer ordem para as operações, desde que as operações nas camadas mais baixas da árvore sejam executadas antes das operações nas camadas mais altas.

Entretanto, a escolha do algoritmo mais barato para cada operação de forma independente não é necessariamente uma boa idéia. Embora uma merge-junção em um certo nível possa ser mais cara que uma hash-junção, ela consegue prover um resultado classificado que torna mais barata a avaliação de uma operação posterior (como a eliminação de duplicatas, a interseção ou outra merge-junção). De maneira similar, uma junção de laço aninhado com indexação pode proporcionar oportunidades para colocar os resultados em um pipeline para a próxima

operação e, assim, ela seria útil, mesmo se não fosse a forma mais barata de executar uma junção. Para escolher o melhor algoritmo global, devemos considerar até mesmo os algoritmos que não são os melhores para as operações individuais.

Assim, além de considerarmos expressões alternativas para uma consulta, devemos também considerar algoritmos alternativos para cada operação na expressão. Podemos usar regras como as de equivalência para definir quais são os algoritmos possíveis para cada operação, inclusive para definir se devemos ou não colocá-la em pipeline com outra expressão. Podemos usar essas regras para avaliar sistematicamente todos os planos de avaliação de consultas para uma dada expressão.

Dado um plano de avaliação, podemos usar as técnicas descritas anteriormente neste capítulo para estimar seu custo. Mesmo assim, o problema de escolha do melhor plano de avaliação para uma consulta ainda permanece. Há duas abordagens: a primeira procura todos os planos e escolhe o melhor com base no custo. A segunda usa a heurística para escolher o plano. Na prática, os otimizadores de consultas incorporam elementos das duas abordagens.

12.10.2 Otimização Baseada no Custo

Um otimizador baseado no custo gera uma faixa de planos de avaliação a partir de uma determinada consulta usando as regras de equivalência e escolhe aquele de menor custo. Para uma consulta complexa, o número de diferentes planos de avaliação da consulta que são equivalentes a um determinado plano pode ser grande. Como ilustração, considere a expressão:

$$r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$$

em que as junções estão expressas sem qualquer ordem. Com $n = 3$, há 12 ordens de junção diferentes:

$$\begin{array}{cccc} r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\ r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\ r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3 \end{array}$$

Em geral, com n relações, há $(2(n - 1))!/(n - 1)!$ ordens de junção diferentes. (Deixamos o cálculo dessa expressão como exercício – Exercício 12.23.) Para junções que envolvam um pequeno número de relações, esse número é aceitável; por exemplo, com $n = 5$, o número é 1.680. Entretanto, conforme n cresce, esse número aumenta muito rapidamente. Com $n = 7$, o número é 665.280; com $n = 10$, o número é maior que 176 bilhões!

Por sorte, não é necessário gerar todas as expressões equivalentes a uma determinada expressão. Por exemplo, suponha que queiramos encontrar a melhor ordem de junção para a forma:

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

que represente todas as ordens de junção em que primeiro é feita a junção de r_1 , r_2 e r_3 (em alguma ordem), e a junção do resultado é feita (em alguma ordem) com r_4 e r_5 . Há 12 diferentes ordens de junção para calcular $r_1 \bowtie r_2 \bowtie r_3$, e 12 ordens para calcular a junção de seu resultado com r_4 e r_5 . Assim, parece haver 144 ordens de junção para exame. Entretanto, uma vez que encontramos a melhor ordem de junção para o subconjunto de relações $\{r_1, r_2, r_3\}$, podemos usar essa ordem nas junções posteriores com r_4 e r_5 , e podemos ignorar todas

as ordens de junção mais caras de $r_1 \bowtie r_2 \bowtie r_3$. Assim, em vez de 144 alternativas para examinar, precisamos de apenas $12 + 12$ alternativas. Usando essa idéia, podemos desenvolver um algoritmo de programação dinâmica para encontrar as ordens ótimas de junção, cuja abordagem reduz grandemente o número de expressões examinadas. O algoritmo calcula a melhor ordem de junção para cada subconjunto de um determinado conjunto de n relações. (Veja o Exercício 12.24.)

Na verdade, a ordem na qual as tuplas são geradas pela junção de $r_1 \bowtie r_2 \bowtie r_3$ também é importante para encontrar a melhor ordem global de junção, já que isso pode afetar o custo das junções posteriores (por exemplo, se a merge-junção for utilizada). Uma determinada ordem de classificação das tuplas é dita uma *ordem de classificação interessante* se ela puder ser útil para uma operação futura. Por exemplo, dado o resultado de $r_1 \bowtie r_2 \bowtie r_3$, sua classificação segundo os atributos comuns com r_4 e r_5 poderia ser útil, mas sua classificação segundo os atributos comuns com apenas r_1 e r_2 não. A utilização da merge-junção obtendo $r_1 \bowtie r_2 \bowtie r_3$ pode ser mais cara que a utilização de alguma outra técnica de junção, mas pode prover um resultado classificado em uma ordem interessante.

Conseqüentemente, não é suficiente encontrar a melhor ordem de junção para cada subconjunto do conjunto das n relações dadas. Em vez disso, temos de encontrar a melhor ordem de junção para cada subconjunto, para cada ordem de classificação interessante do resultado da junção para aquele subconjunto. O número de subconjuntos das n relações é 2^n . O número de ordens de classificação interessantes geralmente não é grande. Assim, aproximadamente 2^n expressões de junção precisam ser armazenadas. O algoritmo de programação dinâmica para encontrar a melhor ordem de junção pode facilmente ser estendido para tratar as ordens de classificação, e tem um custo de tempo ao redor de 3^n . (Veja o Exercício 12.25.)

Com $n = 10$, esse número está ao redor de 59.000, que é muito melhor que os 176 bilhões de ordens de junção diferentes. Mais importante, o espaço de armazenamento necessário é muito menor que o requerido anteriormente, já que precisamos armazenar somente uma ordem de junção para cada ordem interessante de cada um dos 1.024 subconjuntos de r_1, \dots, r_{10} . Embora ambos os números ainda cresçam rapidamente em função de n , as junções mais comuns normalmente têm menos de 10 relações, e podem ser facilmente tratadas.

Podemos usar várias técnicas para posteriormente reduzir o custo de procura por meio de um grande número de planos. Por exemplo, quando examinamos os planos para uma expressão, podemos terminar após examinarmos apenas uma parte da expressão, se determinarmos que o plano mais barato para aquela parte já está mais caro que a avaliação mais barata para uma expressão completa examinada anteriormente. De forma semelhante, suponha que determinarmos que o modo mais barato de avaliar uma subexpressão é mais caro que o plano de avaliação mais barato para uma expressão completa examinada anteriormente. Então, nenhuma expressão completa envolvendo aquela subexpressão precisa ser examinada. Posteriormente, podemos reduzir o número de planos de avaliação que precisam ser completamente considerados, fazendo primeiramente uma suposição heurística de um bom plano e estimando o custo desse plano. Então, apenas alguns planos necessitam de uma análise completa de custo. Essas otimizações reduzem significativamente o overhead da otimização de consultas.

12.10.3 Otimização Heurística

Uma desvantagem da otimização baseada no custo é o custo da própria otimização. Embora o custo do processamento da consulta possa ser reduzido por meio de otimizações inteligentes, a otimização baseada no custo ainda é cara. Consequentemente, muitos sistemas usam a heurística para reduzir o número de alternativas que podem ser escolhidas em uma abordagem baseada no custo. Alguns sistemas até mesmo escolhem usar apenas a heurística e, absolutamente, não usam a otimização baseada no custo.

A seguir, um exemplo de uma regra de heurística para transformar consultas da álgebra relacional:

- Execute as operações de seleção assim que possível.

Um otimizador heurístico usaria essa regra sem descobrir se o custo é reduzido por essa transformação. No primeiro exemplo de transformação na Seção 12.9, a operação de seleção foi acrescentada em uma junção.

Dizemos que a regra precedente é uma heurística porque normalmente, mas nem sempre, ajuda a reduzir o custo. Para um exemplo em que ela possa resultar em um aumento no custo considere uma expressão $\sigma_{\theta}(r \bowtie s)$, em que a condição θ se refere somente a atributos em s . A seleção pode ser executada certamente antes da junção. Porém, se r for extremamente pequeno quando comparado com s e se há um índice nos atributos de junção de s , mas nenhum índice nos atributos usados por θ , então, provavelmente, é uma má idéia executar a seleção antes. Executar a seleção antes – ou seja, diretamente em s – exigiria uma varredura de todas as tuplas de s . Provavelmente é mais barato, nesse caso, obter a junção usando o índice e, então, rejeitar as tuplas que não passam na seleção.

A operação de projeção, como a operação de seleção, reduz o tamanho das relações. Assim, sempre que precisamos gerar uma relação temporária, é vantajoso aplicar imediatamente qualquer projeção possível. Essa vantagem sugere uma companheira para a heurística “execute as seleções antes” que apresentamos anteriormente:

- Execute as projeções antes.

Normalmente, é melhor executar as seleções antes que as projeções, já que as seleções têm o potencial de reduzir significativamente os tamanhos de relações e permitem o uso de índices para acesso às tuplas. Um exemplo semelhante ao usado para a heurística da seleção deve mostrar que essa heurística nem sempre reduz o custo.

Utilizando a equivalência discutida na Seção 12.9.2, um algoritmo de otimização heurística reordenará os componentes de uma árvore de consulta inicial para obter uma execução de consulta melhorada. Apresentamos agora uma visão geral dos passos de um algoritmo típico de otimização heurística. Você pode entender a heurística visualizando uma expressão de consulta como uma árvore, conforme ilustrado anteriormente.

1. Separe as seleções conjuntivas em uma seqüência de operações de seleção isoladas. Baseado na regra de equivalência 1, este passo facilita mover as operações de seleção para baixo na árvore de consulta.

2. Mova as operações de seleção para baixo na árvore de consulta para que sua execução ocorra o mais cedo possível. Este passo usa as propriedades de comutatividade e de distribuição da operação de seleção observadas nas regras de equivalência 2, 7a, 7b e 11.

Por exemplo, $\sigma_\theta(r \bowtie s)$ é transformado em $\sigma_\theta(r) \bowtie s$ ou $r \bowtie \sigma_\theta(s)$ sempre que possível. A execução, o mais cedo possível, das seleções baseadas em valor reduz o custo de classificar e fazer o merge dos resultados intermediários. O grau de reordenação permitido para uma seleção particular é determinado pelos atributos envolvidos naquela condição de seleção.

3. Determine quais operações de seleção e de junção produzirão as menores relações – ou seja, produzirão as relações com o menor número de tuplas. Usando a associatividade da operação \bowtie , reorganize a árvore de tal forma que as relações dos nós folhas, com essas seleções restritivas, sejam executadas primeiro.

Este passo considera a seletividade de uma seleção ou a condição de junção. Lembre-se de que a maioria das seleções restritivas – ou seja, a condição com menor seletividade – recupera o menor número de registros. Este passo baseia-se na associatividade das operações binárias dadas nas regras de equivalência 6 e 10.

4. Substitua as operações de produto cartesiano que são seguidas por uma condição de seleção (regra 4a) por operações de junção. Como demonstrado na Seção 12.9.4, a operação de produto cartesiano é onerosa para ser implementada. Em $r_1 \times r_2$, o resultado não somente inclui um registro para cada combinação de registros de r_1 e r_2 , mas também os atributos de resultado incluem todos os atributos de r_1 e r_2 . Então, é aconselhável evitar o uso da operação de produto cartesiano.
5. Separe e move o mais para baixo possível na árvore as listas de atributos de projeção, criando projeções novas onde forem necessárias. Este passo utiliza as propriedades da operação de projeção dadas nas regras de equivalência 3, 8a, 8b e 12.
6. Identifique aquelas subárvores cujas operações podem ser colocadas em pipeline e execute-as usando pipelining.

Em resumo, as heurísticas listadas reordenam uma representação inicial de uma árvore de consulta, de tal forma que as operações que reduzem o tamanho dos resultados intermediários são aplicadas primeiro.

Executar seleções mais cedo reduz o número de tuplas e executar projeções mais cedo reduz o número de atributos. As transformações de heurística também reestruturam a árvore, de tal forma que a seleção mais restritiva e as operações de junção são executadas antes de outras operações semelhantes.

A otimização heurística mapeia a expressão de consulta transformando-a, de modo heurístico, em seqüências alternativas de operações para produzir um conjunto de planos de avaliação candidata.

Um plano de avaliação não só inclui as operações relacionais a serem executadas, mas também os índices a serem usados, a ordem na qual as tuplas são acessadas e a ordem na qual as operações são executadas. A fase de *seleção do plano de seleção* de um otimizador heurístico escolhe a estratégia mais eficiente para cada operação.

12.10.4 A Estrutura dos Otimizadores de Consulta

Até o momento, descrevemos as duas abordagens básicas para escolher um plano de avaliação. Na prática, a maioria dos otimizadores de consulta combina elementos de ambas as abordagens. Alguns otimizadores de consulta, como o otimizador do System R, não consideram todas as ordens de junção, mas, em vez disso, restringem a procura para tipos particulares de ordens de junção. O otimizador System R considera apenas aquelas ordens de junção nas quais o operador à direita de cada junção é uma das relações iniciais r_1, \dots, r_n . Essas ordens de junção são chamadas de *ordens de junção de profundidade à esquerda*. As ordens de junção de profundidade à esquerda são particularmente convenientes para avaliação em pipeline, já que o operando à direita é uma relação armazenada, assim apenas uma entrada para cada junção é colocada no pipeline.

A Figura 12.16 ilustra a diferença entre árvores de junção de profundidade à esquerda e árvores de junção que não são de profundidade à esquerda. O tempo gasto para considerar todas as ordens de junção de profundidade à esquerda é $O(n!)$, que é muito menor que o tempo gasto para considerar todas as ordens de junção. Com o uso de algumas otimizações, o otimizador System R pode encontrar a melhor ordem de junção em um tempo aproximado de $O(2^n)$. Em contraste com esse custo, o tempo de 3^n é necessário para encontrar a melhor de todas as ordens de junção. O otimizador do System R usa heurística para levar as seleções e projeções para baixo na árvore de consulta.

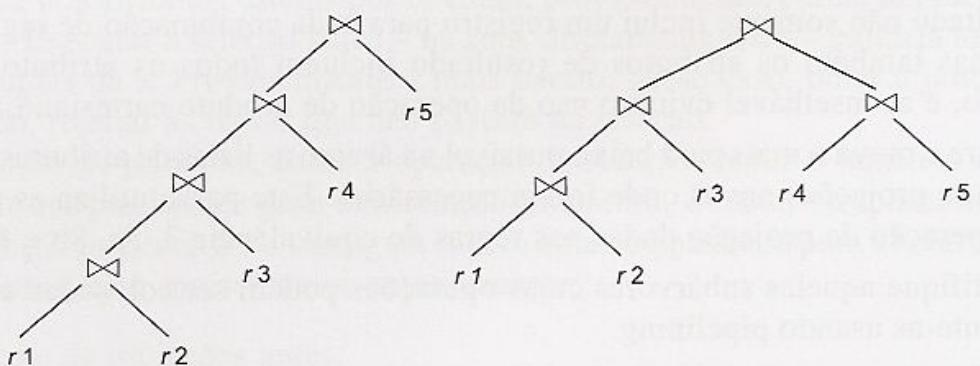


Figura 12.16 Árvores de junção de profundidade à esquerda.

A estimativa de custo que apresentamos para as varreduras usando índices secundários supõe que todos os acessos a tuplas resultam em uma operação de I/O. É provável que a estimativa seja precisa com buffers pequenos; com buffers grandes, porém, a página que contém a tupla já pode estar no buffer. O otimizador Sybase incorpora uma técnica melhor de estimativa de custo para essas varreduras: ele leva em consideração a probabilidade de a página que contém a tupla estar no buffer.

As abordagens de otimização de consulta que integram a seleção de heurística e a geração de planos de acesso alternativos foram adotadas em vários sistemas. A abordagem usada no System R e em seu sucessor, o projeto Starburst, consiste em um procedimento hierárquico baseado no conceito de bloco aninhado de SQL. As técnicas de otimização baseadas em custo, descritas aqui, são usadas separadamente para cada bloco da consulta. O Ingres

usa a decomposição heurística para reduzir uma consulta em um conjunto de subconsultas que não contém mais que duas relações. Então, planos de acesso detalhados são gerados para cada uma dessas subconsultas.

O otimizador do Oracle7 tem dois componentes: um é baseado em regras de heurística e o outro é baseado em custo.⁶ A abordagem heurística trabalha grosseiramente, conforme descreveremos. Para uma junção de n vias, ela considera n planos de avaliação construídos como descrito a seguir. Cada plano usa uma ordem de junção de profundidade à esquerda e começa com uma relação diferente das n relações. A heurística constrói a ordem de junção para cada um dos n planos de avaliação por meio da seleção repetitiva da “melhor” relação para a próxima junção, baseada em um ranking dos caminhos de acesso disponíveis. A junção de laço aninhado ou a junção sort-merge é escolhida para cada uma das junções, baseada nos caminhos de acesso disponíveis. Finalmente, um dos n planos de avaliação é escolhido de uma maneira heurística, preocupado em minimizar o número de junções de laço aninhado que não tenha um índice disponível na relação interna e no número de junções sort-merge.

As complexidades da SQL introduzem uma boa dose de complexidade nos otimizadores de consulta. Em particular, é difícil traduzir subconsultas em SQL para a álgebra relacional. Muitos otimizadores, como o System R e o Starburst, tentam reescrever as consultas SQL para transformá-las em junções onde for possível. Quando a transformação não é possível, as subconsultas são mantidas como expressões separadas e otimizadas separadamente. Os planos de avaliação escolhidos são combinados por meio de algoritmos especiais de avaliação. De forma semelhante, para consultas SQL combinadas (usando as operações \cup , \cap ou $-$), cada componente é otimizado separadamente, e os planos de avaliação são combinados para gerar o plano de avaliação global.

Até mesmo com o uso de heurística, a otimização de consulta baseada em custo impõe um overhead substancial ao processo de consulta. Porém, o custo adicional da otimização de consulta baseada em custo é normalmente mais compensador que a economia de tempo na execução da consulta, que é dominada por acessos lentos a discos. A diferença no tempo de execução entre um plano bom e um ruim pode ser enorme, tornando a otimização da consulta essencial. A economia alcançada é aumentada naquelas aplicações que são executadas regularmente, em que a consulta pode ser otimizada uma vez e o plano de consulta selecionado pode ser usado em cada execução. Então, a maioria dos sistemas comerciais inclui otimizadores relativamente sofisticados. As notas bibliográficas possuem referências para descrições de otimizadores de sistemas de banco de dados atuais.

12.11 Resumo

A primeira ação que o sistema tem de executar em uma consulta é traduzir a consulta para sua forma interna, que (para sistemas de bancos de dados relacionais) normalmente é baseada na álgebra relacional. No processo de geração da forma interna da consulta, o analisador sintático verifica a sintaxe da consulta do usuário, verifica se os nomes das relações que aparecem na consulta são nomes de relações no banco de dados e assim por diante. Se a

6. Os detalhes que descrevemos referem-se ao Oracle7 em sua versão de dezembro de 1992. As versões mais recentes podem seguir outras técnicas de otimização.

consulta foi expressa em termos de uma visão, o analisador sintático substitui todas as referências ao nome pela expressão da álgebra relacional para obter a visão.

Dada uma consulta, geralmente há uma variedade de métodos para chegar à resposta. É responsabilidade do sistema transformar a consulta, de acordo com o que foi fornecido pelo usuário, em uma consulta equivalente que pode ser calculada mais eficientemente. Essa *otimização* ou, mais precisamente, essa melhoria da estratégia de processamento de uma consulta é chamada de *otimização de consulta*.

A avaliação de consultas complexas envolve muitos acessos a disco. Como a transferência de dados do disco é relativamente lenta se comparada com a velocidade da memória principal e da CPU do sistema, vale a pena alocar uma considerável quantia de processamento para escolher um método que minimize os acessos de disco.

A estratégia que escolhemos para avaliar uma operação depende do tamanho de cada relação e da distribuição de valores dentro de colunas. A fim de que consigam escolher uma estratégia baseada em informação confiável, os sistemas de banco de dados podem armazenar estatísticas para cada relação r . Essas estatísticas incluem:

- Número de tuplas na relação r .
- Tamanho de um registro (tupla) da relação r em bytes.
- Número de valores distintos que aparecem na relação r para um determinado atributo.

Essas estatísticas nos permitem estimar os tamanhos dos resultados de várias operações, como também o custo para executar as operações. Informação estatística sobre relações é particularmente útil quando vários índices estão disponíveis para ajudar no processamento de uma consulta. A presença dessas estruturas tem uma influência significativa na escolha de uma estratégia de processamento de consulta.

Podemos processar consultas que envolvem seleções simples por meio da execução de uma varredura linear, de uma procura binária ou do uso de índices. Podemos tratar as seleções complexas computando uniões e interseções dos resultados de seleções simples. Podemos ordenar relações maiores que a memória usando o algoritmo de merge-sort externo. As consultas que envolvem uma junção natural podem ser processadas de vários modos, dependendo da disponibilidade de índices e da forma de armazenamento físico usada para as relações. Se o resultado da junção é quase tão grande quanto o produto cartesiano das duas relações, uma estratégia de *junção de laço aninhado de bloco* pode ser vantajosa. Se índices estão disponíveis, a *junção de laço aninhado indexada* pode ser usada. Se as relações estão classificadas, uma *merge-junção* pode ser desejável. Pode ser vantajoso ordenar uma relação antes de calcular a junção (de forma a permitir o uso da estratégia *merge-junção*). Também pode ser vantajoso calcular um índice temporário com o propósito exclusivo de permitir a utilização de uma estratégia de junção mais eficiente. O algoritmo de *hash-junção* partitiona as relações em vários pedaços, de forma que cada pedaço de uma das relações caiba na memória. O particionamento é feito por meio de uma função hash nos atributos da junção, de forma que se possa fazer independentemente a junção de pares correspondentes de partições. O hashing e a classificação são duais, no sentido de que muitas operações, como a eliminação de duplicatas, a agregação, a junção e a junção externa, podem ser implementadas tanto pelo hashing quanto pela classificação.

Cada expressão da álgebra relacional representa uma seqüência particular de operadores. O primeiro passo na seleção de uma estratégia de processamento de consulta é encontrar uma expressão da álgebra relacional que seja equivalente a uma determinada expressão e seja estimada para ter um custo menor de execução. Podemos usar várias regras de equivalência para transformar uma expressão em uma equivalente. Usamos essas regras para sistematicamente gerar todas as expressões equivalentes a uma determinada consulta, e escolhemos a expressão mais barata.

Planos de avaliação alternativos para cada expressão podem ser gerados por meio de regras semelhantes, e o plano mais barato entre todas as expressões pode ser escolhido. Várias técnicas de otimização estão disponíveis para reduzir o número de expressões e de planos alternativos que precisa ser gerado.

Usamos a heurística para reduzir o número de planos considerados e, assim, reduzir o custo da otimização. As regras de heurística para transformar consultas de álgebra relacional incluem “execute as operações de seleção o mais cedo possível”, “execute antes as projeções” e “evite os produtos cartesianos”.

Exercícios

- 12.1 Em que ponto durante o processamento de uma consulta ocorre a otimização?
- 12.2 Por que não é desejável forçar os usuários a fazer uma escolha explícita de uma estratégia de processamento de consulta? Há casos em que é desejável para os usuários estarem conscientes dos custos de estratégias de processamento de consultas? Justifique sua resposta.
- 12.3 Considere a seguinte consulta SQL para nosso banco de dados bancário:

```
select T.nome_agência
  from agência T, agência S
 where T.fundos > S.fundos and S.cidade_agência = "Brooklyn"
```

Escreva uma expressão de álgebra relacional eficiente que seja equivalente a essa consulta. Justifique sua escolha.

- 12.4 Considere as relações $r_1(A, B, C)$, $r_2(C, D, E)$ e $r_3(E, F)$ com chaves primárias A , C e E , respectivamente. Suponha que r_1 tenha 1.000 tuplas, r_2 tenha 1.500 tuplas e r_3 , 750 tuplas. Estime o tamanho de $r_1 \bowtie r_2 \bowtie r_3$ e forneça uma estratégia eficiente para calcular a junção.
- 12.5 Considere as relações $r_1(A, B, C)$, $r_2(C, D, E)$ e $r_3(E, F)$ do Exercício 12.4. Suponha que não haja nenhuma chave primária, exceto o esquema inteiro. Sejam $V(C, r_1) 900$, $V(C, r_2) 1.100$, $V(E, r_2) 50$ e $V(E, r_3) 100$. Suponha que r_1 tenha 1.000 tuplas, r_2 tenha 1.500 tuplas e r_3 , 750 tuplas. Estime o tamanho de $r_1 \bowtie r_2 \bowtie r_3$ e forneça uma estratégia eficiente para obter a junção.
- 12.6 Os índices clustering podem permitir acesso mais rápido a dados que os índices não-clustering. Quando temos de criar um índice não-clustering, apesar das vantagens de um índice clustering? Justifique sua resposta.

- 12.7** Quais são as vantagens e as desvantagens de índices hash em relação aos índices árvore-B⁺? Como pode o tipo de índice disponível influenciar na escolha de uma estratégia de processamento de consulta?
- 12.8** Suponha (com fim de simplicidade, neste exercício) que só uma tupla caiba em um bloco e que a memória mantenha no máximo três frames de página. Mostre os temporários criados em cada passagem do algoritmo sort-merge, quando aplicado para classificar as seguintes tuplas no primeiro atributo: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon).⁷
- 12.9** Sejam as relações $r_1(A, B, C)$ e $r_2(C, D, E)$ com as seguintes propriedades: r_1 tem 20.000 tuplas, r_2 tem 45.000 tuplas, 25 tuplas de r_1 cabem em um bloco e 30 tuplas de r_2 cabem em um bloco. Estime o número de acessos a blocos necessários, usando cada uma das seguintes estratégias de junção para $r_1 \bowtie r_2$:
- (a) Junção de laço aninhado.
 - (b) Junção de laço aninhado de bloco.
 - (c) Merge-junção.
 - (d) Hash-junção.
- 12.10** Projete uma variante do algoritmo merge-junção híbrida para o caso em que ambas as relações não estão ordenadas fisicamente, mas possuem um índice secundário classificado nos atributos de junção.
- 12.11** O algoritmo de junção de laço aninhado descrito na Seção 12.6.4 pode ser ineficiente se o índice for secundário e houver múltiplas tuplas com o mesmo valor para os atributos de junção. Por que ele é ineficiente? Descreva um modo, usando classificação, de reduzir o custo da recuperação de tuplas da relação interna. Sob que condições esse algoritmo seria mais eficiente que a merge-junção híbrida?
- 12.12** Estime o número de acessos a blocos necessários para sua solução no Exercício 12.10 para $r_1 \bowtie r_2$, em que r_1 e r_2 são definidos no Exercício 12.9.
- 12.13** Considere as relações r_1 e r_2 do Exercício 12.9 e uma relação $r_3(E, F)$. Suponha que r_3 tenha 30.000 tuplas e que 40 tuplas de r_3 caibam em um bloco. Estime os custos das três estratégias da Seção 12.6.7 para calcular $r_1 \bowtie r_2 \bowtie r_3$.
- 12.14** Sejam r e s relações sem índices e suponha que as relações não estejam classificadas. Supondo a memória infinita, qual o modo de custo mais baixo (em termos de operações de I/O) para calcular $r \bowtie s$? Qual a quantidade de memória necessária para esse algoritmo?
- 12.15** Suponha que um índice árvore-B⁺ em *nome_agência* esteja disponível na relação *agência*, e que nenhum outro índice esteja disponível. Qual seria o melhor modo para tratar as seguintes seleções que envolvem a negação?
- (a) $\sigma_{(cidade_agência < "Brooklyn")}(agência)$
 - (b) $\sigma_{(cidade_agência = "Brooklyn")}(agência)$
 - (c) $\sigma_{(cidade_agência < "Brooklyn" \vee fundos < 5000)}(agência)$

7. N.T.: Foram mantidos os nomes originais dos animais, em inglês, por se tratar de um exercício que propõe classificação e cuja resposta ficaria totalmente distorcida se fosse feita a tradução.

- 12.16** Suponha que um índice árvore-B⁺ em *nome_agência* esteja disponível na relação *agência*, e que nenhum outro índice esteja disponível. Qual seria o melhor modo para tratar a seguinte seleção?

$$\sigma_{(cidade_agência < "Brooklyn" \vee fundos < 5000) \wedge (nome_agência) \neq "Downtown"}(agência)$$

- 12.17** O algoritmo hash-junção, conforme descrito na Seção 12.6.6, calcula a junção natural de duas relações. Descreva como estender o algoritmo hash-junção para calcular a junção natural externa à esquerda, a junção natural externa à direita e a junção natural externa completa. (*Sugestão:* Mantenha informações adicionais junto com cada tupla no índice hash, para detectar se qualquer tupla na relação de teste coincide com a tupla no índice hash.) Experimente seu algoritmo nas relações *cliente* e *depositante*.
- 12.18** Escreva o pseudocódigo para um iterator que implementa a junção de laço aninhado indexada, em que a relação externa é colocada em um pipeline. Use as funções-padrão de iterator em seu pseudocódigo. Mostre quais informações de estado o iterator deve manter entre as chamadas.
- 12.19** Mostre se a equivalência seguinte é válida. Explique como você pode aplicá-la para melhorar a eficiência de certas consultas:
- $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3).$
 - $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E)),$ em que θ usa somente atributos de $A.$
 - $\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie E_2,$ em que θ usa somente atributos de $E_2.$
- 12.20** Mostre como derivar as seguintes equivalências por meio de uma seqüência de transformações, usando as regras de equivalência da Seção 12.9.2.
- $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$
 - $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2))),$ em que θ_2 envolve apenas atributos de $E_2.$
- 12.21** Para cada um dos seguintes pares de expressões, dê instâncias das relações que mostram que as expressões não são equivalentes.
- $\Pi_A(R - S)$ e $\Pi_A(R) - \Pi_A(S)$
 - $\sigma_{B < 4}({}_A\mathcal{G}_{max(B)}(R))$ e ${}_A\mathcal{G}_{max(B)}(\sigma_{B < 4}(R))$
 - $\sigma_{B < 4}({}_A\mathcal{G}_{min(B)}(R))$ e ${}_A\mathcal{G}_{min(B)}(\sigma_{B < 4}(R))$
 - $(R \bowtie S) \bowtie T$ e $R \bowtie (S \bowtie T)$
- Em outras palavras, a junção natural externa à esquerda não é associativa. (*Sugestão:* Suponha que os esquemas das três relações sejam $R(a, b1)$, $S(a, b2)$ e $T(a, b3)$, respectivamente.)
- $\sigma_\theta(E_1 \bowtie E_2)$ e $E_1 \bowtie \sigma_\theta(E_2),$ em que θ usa apenas atributos de $E_2.$
- 12.22** A linguagem SQL permite relações com duplicatas (Capítulo 4).
- Defina as versões das operações básicas da álgebra relacional σ , Π , \times , \bowtie , $-$, \cup e \cap que trabalham com relações com duplicatas, de um modo consistente com a SQL.
 - Verifique quais das regras de equivalência 1 até 7b são válidas para a versão de álgebra relacional definida na parte a.

- 12.23** Mostre que, com n relações, há $(2(n - 1))!/(n - 1)!$ ordens de junção diferentes.
- 12.24** Descreva um algoritmo de programação dinâmica para encontrar a mais eficiente ordem de junção para a junção de n relações. Suponha que haja só uma ordem de classificação interessante. (*Sugestão:* Mostre como derivar a ordem de junção de mais baixo custo (e seu custo) para um conjunto de n relações, dada a ordem de junção de mais baixo custo (e seu custo) para cada subconjunto das n relações.)
- 12.25** Mostre que a ordem de junção de mais baixo custo pode ser calculada no tempo $O(3^n)$. Suponha que você possa armazenar e possa procurar informação sobre um conjunto de relações (como a ordem de junção ótima para o conjunto e o custo dessa ordem de junção) em um tempo constante. (Se você acha este exercício difícil, pelo menos mostre o tempo gasto associado (*looser time bound*) de $O(2^{2n})$).
- 12.26** Mostre que, se considerarmos apenas as árvores de junção de profundidade à esquerda, como no otimizador do System R, o tempo gasto para achar a ordem de junção mais eficiente é por volta de 2^n . Suponha que haja somente uma ordem de classificação interessante.
- 12.27** Um conjunto de regras de equivalência é dito *completo* se, sempre que duas expressões são equivalentes, uma puder ser derivada da outra por meio do uso de uma seqüência das regras de equivalência. O conjunto de regras de equivalência que consideramos na Seção 12.9.2 é completo? *Sugestão:* Considere a equivalência $\sigma_{3=5}(r) = \{ \}$.

Notas Bibliográficas

Um processador de consulta deve analisar sintaticamente as declarações na linguagem de consulta e traduzi-las em uma forma interna. A análise sintática das linguagens de consulta difere um pouco da análise sintática de linguagens de programação tradicionais. A maioria dos textos sobre compilador, inclusive Aho *et al.* [1986] e Tremblay & Sorenson [1985], cobre as principais técnicas de análise sintática. Esses dois textos também apresentam a otimização a partir de um ponto de vista de linguagem de programação.

Knuth [1973] apresenta uma descrição excelente de algoritmos de classificação externa, inclusive uma otimização que pode criar temporários iniciais que têm (em média) duas vezes o tamanho de memória. Baseado em estudos de desempenho administrados no meio da década de 1970, sistemas de banco de dados usam apenas a junção de laço aninhado e a merge-junção. Esses estudos, que estão relacionados ao desenvolvimento do System R, determinaram que a junção de laço aninhado e a merge-junção quase sempre fornecem o método ótimo de junção [Blasgen & Eswaran, 1976]; consequentemente, estes dois foram os únicos algoritmos de junção implementados no System R.

Entretanto, o estudo do System R não incluiu uma análise dos algoritmos hash-junção. Hoje, as hash-junções são consideradas altamente eficientes. Os algoritmos de hash-junção foram desenvolvidos inicialmente para sistemas de bancos de dados paralelos. Sistemas experimentais usando métodos de hash-junção foram desenvolvidos no meio da década de 1980; notáveis entre esses sistemas são a máquina de banco de dados Grace (Kitsuregawa *et al.* [1983], Fushimi *et al.* [1986]) e a máquina de banco de dados Gamma (DeWitt *et al.* [1986, 1990]. As técnicas de hash-junção são descritas em Kitsuregawa *et al.* [1983] e extensões, inclusive a hash-junção híbrida, são descritas em Shapiro [1986]. Resultados

mais recentes de Zeller & Gray [1990] e Davison & Graefe [1994] descrevem técnicas hash-junção que podem se adaptar à memória disponível, o que é importante em sistemas em que múltiplas consultas podem estar em execução ao mesmo tempo.

O processamento de consulta na memória principal do banco de dados é coberto em DeWitt *et al.* [1984] e Whang & Krishnamurthy [1990]. Kim [1982, 1984] descreve as estratégias de junção e a utilização ótima da memória principal disponível. O processamento de consultas usando junções externas é descrito em Rosenthal & Reiner [1984], Galindo-Legaria & Rosenthal [1992] e Galindo-Legaria [1994]. Klug [1982] discute a otimização de expressões da álgebra relacional com funções de agregação. Resultados teóricos sobre a complexidade da computação de operações da álgebra relacional aparecem em Gotlieb [1975], Pecherer [1975] e Blasgen & Eswaran [1976]. Uma comparação de vários algoritmos de avaliação de consulta é dada por Yao [1979b]. Discussões adicionais são apresentadas em Kim *et al.* [1985].

Graefe [1993] apresenta um excelente survey de técnicas de avaliação de consulta. Um survey mais antigo das técnicas de processamento de consulta aparece em Jarke & Koch [1984].

O trabalho de Selinger *et al.* [1979] descreve a seleção de caminho de acesso no otimizador do System R, que foi um dos primeiros otimizadores de consultas relacionais. Wong & Youssefi [1976] descrevem uma técnica chamada *decomposição*, que é usada no otimizador de consulta Ingres. O processamento de consultas no Starburst é descrito em Haas *et al.* [1989]. A otimização de consultas no Oracle é esboçada brevemente em Oracle [1992].

Graefe & McKenna [1993] descrevem as características de extensão e a eficiência da procura de Volcano, um otimizador de consulta baseado em regras. A procura exaustiva de todos os planos de consulta não é prática para a otimização de junções envolvendo muitas relações, e técnicas baseadas na procura aleatória, que não examinam todas as alternativas, foram propostas. Ioannidis & Wong [1987], Swami & Gupta [1988] e Ioannidis & Kang [1990] apresentam resultados nessa área. Técnicas paramétricas de otimização de consultas foram propostas por Ioannidis *et al.* [1992], para manipular o processamento de consultas quando o tamanho da relação muda freqüentemente. Um conjunto de planos – um para cada um dos vários tamanhos diferentes de relações – é calculado e armazenado pelo otimizador em tempo de compilação. Um desses planos é escolhido em tempo de execução, baseado no tamanho real da relação, evitando o custo da otimização completa em tempo de execução. A distribuição não-uniforme de valores causa problemas para a estimativa do tamanho da consulta e seu custo. Técnicas de estimativa de custo que usam histogramas de distribuições de valor têm sido propostas para tentar resolver o problema. Ioannidis & Christodoulakis [1993], Ioannidis & Poosala [1995] e Poosala *et al.* [1996] apresentam resultados nessa área.

A linguagem SQL apresenta vários desafios para a otimização de consulta, inclusive a presença de duplicatas, valores nulos e a semântica de subconsultas aninhadas. A extensão da álgebra relacional para duplicatas é descrita em Dayal *et al.* [1982]; a otimização de subconsultas aninhadas é discutida em Kim [1982], Ganski & Wong [1987] e Dayal [1987]. Chaudhuri & Shim [1994] descrevem as técnicas para otimização de consultas que usam a agregação.

Sellis [1988] descreve a otimização de multiconsultas, que é o problema de otimizar a execução de várias questões como um grupo. Se um grupo inteiro de consultas é considerado,

é possível descobrir *subexpressões comuns* que podem ser avaliadas uma vez para o grupo inteiro. Finkelstein [1982] e Hall [1976] consideram a otimização de um grupo de consultas e o uso de subexpressões comuns.

A otimização de consultas pode fazer uso de informações semânticas, como dependências funcionais e outras restrições de integridade. A otimização de consultas por semântica em bancos de dados relacionais é coberta por King [1981]. Malley & Zdonik [1986] apresentam uma abordagem baseada em conhecimento para a otimização de consultas. Chakravarthy *et al.* [1990] usam as restrições de integridade para ajudar na otimização de consultas.

O processamento de consultas para bancos de dados orientados a objeto é discutido em Maier & Stein [1986], Beech [1988], Bertino & Kim [1989], Clue *et al.* [1989], Kim [1989] e Kim *et al.* [1989].

Quando as consultas são geradas por meio de visões, ocorre a junção de mais relações que o necessário para o cálculo da consulta. Um conjunto de técnicas para a minimização da junção foi agrupado sob o nome de *tabela de otimização*. O conceito dessa tabela foi introduzido por Aho *et al.* [1979a, 1979c] e posteriormente estendido por Sagiv & Yannakakis [1981]. Ullman [1988] e Maier [1983] fornecem um livro-texto que cobre essa tabela.