

Elmasri • Navathe

# Sistemas de banco de dados

6<sup>a</sup> Edição



Companion  
Website

PEARSON

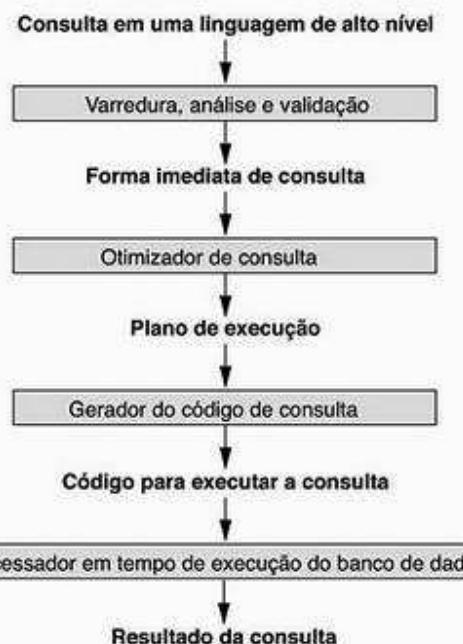
# Algoritmos para processamento e otimização de consulta

capítulo

# 19

Neste capítulo, discutimos as técnicas usadas internamente por um SGBD para processar, otimizar e executar consultas de alto nível. Uma consulta expressa em uma linguagem de consulta de alto nível, como SQL, primeiro precisa ser lida, analisada e validada.<sup>1</sup> A varredura identifica os tokens de consulta — como as palavras-chave SQL, nomes de atributo e nomes de relação — que aparecem no texto da consulta, enquanto o analisador sintático verifica a sintaxe da consulta para determinar se ela está formulada de acordo com as regras de sintaxe (regras de gramática) da linguagem de consulta. A consulta também precisa ser validada verificando se todos os nomes de atributo e relação são válidos e semanticamente significativos no esquema do banco de dados em particular sendo consultado. Uma representação interna da consulta é então criada, normalmente como uma estrutura de dados de árvore chamada árvore de consulta. Também é possível representar a consulta usando uma estrutura de dados de grafo chamada grafo de consulta. O SGBD precisa então idealizar uma estratégia de execução ou plano de consulta para recuperar os resultados da consulta com base nos arquivos de banco de dados. Uma consulta costuma ter muitas estratégias de execução possíveis, e o processo de escolha de uma estratégia adequada para processá-la é conhecido como otimização de consulta.

A Figura 19.1 mostra as diferentes etapas do processamento de uma consulta de alto nível. O módulo otimizador de consulta tem a tarefa de produzir um bom plano de execução, e o gerador de código dá origem ao código para executar esse plano.



Código pode ser:

- Executado diretamente (modo interpretado)
- Armazenado e executado mais tarde, sempre que possível (modo compilado)

Figura 19.1

Etapas típicas ao processar uma consulta de alto nível.

<sup>1</sup> Não discutiremos aqui a fase de análise e verificação de sintaxe do processamento da consulta; esse material é discutido nos livros-texto sobre compilador.

O processador em tempo de execução do banco de dados tem a tarefa de rodar (executar) o código da consulta, sejam no modo compilado ou interpretado, para produzir o resultado da consulta. Se houver um erro em tempo de execução, uma mensagem de erro é gerada pelo processador em tempo de execução do banco de dados.

O termo *otimização* é, na realidade, um nome errado, pois em alguns casos o plano de execução escolhido não é a estratégia ótima (ou a melhor absoluta) — trata-se apenas de uma *estratégia razoavelmente eficiente* para executar a consulta. Encontrar a estratégia ideal em geral é muito demorado — exceto para a mais simples das consultas. Além disso, tentar encontrar a estratégia de execução de consulta ideal pode exigir informações detalhadas sobre como os arquivos são implementados e até mesmo sobre seu conteúdo — informações que podem não estar totalmente disponíveis no catálogo do SGBD. Logo, o *planejamento de uma boa estratégia de execução* pode ser uma descrição mais precisa do que a *otimização de consulta*.

Para as linguagens de banco de dados navegacionais de nível mais baixo nos sistemas legados — como a DML de rede ou a DL/1 hierárquica (ver Seção 2.6) —, o programador deve escolher a estratégia de execução de consulta enquanto escreve o programa de banco de dados. Se um SGBD oferece apenas uma linguagem navegacional, existe uma *necessidade ou oportunidade limitada* para otimização de consulta extensiva pelo SGBD; em vez disso, o programador recebe a responsabilidade de escolher a estratégia de execução de consulta. Por sua vez, uma linguagem de consulta de alto nível — como SQL para SGBDs relacionais (SGBDRs) ou OQL (ver Capítulo 11) para SGBDs de objeto (SGBDOs) — é mais declarativa por natureza, pois especifica quais são os resultados intencionados da consulta, em vez de identificar os detalhes de *como* o resultado deve ser obtido. A otimização de consulta é, portanto, necessária para consultas que são especificadas em uma linguagem de consulta de alto nível.

Vamos nos concentrar na descrição da otimização de consulta no *contexto de um SGBDR*, pois muitas das técnicas que descrevemos também foram adaptadas para outros tipos de sistemas de gerenciamento de banco de dados, como os SGBDOs.<sup>2</sup> Um SGBD relacional deve avaliar sistematicamente estratégias alternativas de execução de consulta e esco-

lher uma estratégia razoavelmente eficiente ou quase ideal. Cada SGBD normalmente tem uma série de algoritmos gerais de acesso de banco de dados que implementam operações da álgebra relacional, como SELEÇÃO ou JUNÇÃO (ver Capítulo 6) ou combinações dessas operações. Somente estratégias de execução que podem ser implementadas pelos algoritmos de acesso do SGBD e que se aplicam à consulta em particular, bem como ao *projeto de banco de dados físico em particular*, podem ser consideradas pelo módulo de otimização de consulta.

Este capítulo começa com uma discussão geral sobre como as consultas SQL costumam ser traduzidas para consultas da álgebra relacional e depois otimizadas na Seção 19.1. Depois, discutimos algoritmos para implementar operações da álgebra relacional nas seções 19.2 a 19.6. Na sequência, damos uma visão geral das estratégias de otimização de consulta. Existem duas técnicas principais que são empregadas durante a otimização da consulta. A primeira técnica é baseada em regras heurísticas para ordenar as operações em uma estratégia de execução de consulta. Uma heurística é uma regra que funciona bem na maioria dos casos, mas não garante funcionar bem em todos eles. As regras em geral reordenam as operações em uma árvore de consulta. A segunda técnica envolve estimar sistematicamente o custo de diferentes estratégias de execução e escolher o plano de execução com a estimativa de custo mais baixa. Essas técnicas normalmente são combinadas em um otimizador de consulta. Abordamos a otimização heurística na Seção 19.7 e a estimativa de custo na Seção 19.8. Depois, oferecemos uma breve visão geral dos fatores considerados durante a otimização de consulta no SGBD comercial Oracle na Seção 19.9. A Seção 19.10 introduz o assunto de otimização de consulta semântica, em que restrições conhecidas são usadas como um recurso para idealizar estratégias de execução de consulta eficientes.

Os tópicos abordados neste capítulo exigem que o leitor esteja acostumado com o material apresentado em vários capítulos anteriores. Em particular, os capítulos sobre SQL (capítulos 4 e 5), álgebra relacional (Capítulo 6) e estruturas e indexação de arquivo (capítulos 17 e 18) são um pré-requisito para este capítulo. Além disso, é importante observar que o tópico de processamento e otimização de consulta é vasto, e só podemos oferecer aqui uma introdução aos princípios e técnicas básicas.

<sup>2</sup> Existem alguns problemas e técnicas de otimização de consulta que são pertinentes apenas aos SGBDOs. Contudo, não os discutimos aqui porque oferecemos apenas uma introdução ao assunto.

## 19.1 Traduzindo consultas SQL para álgebra relacional

Na prática, a SQL é a linguagem de consulta usada na maioria dos SGBDRs comerciais. Uma consulta SQL é primeiro traduzida para uma expressão equivalente da álgebra relacional estendida — representada como uma estrutura de dados de árvore de consulta — que é, então, otimizada. Normalmente, as consultas SQL são decompostas em *blocos de consulta*, que formam as unidades básicas que podem ser traduzidas em operadores algébricos e otimizadas. Um bloco de consulta contém uma única expressão SELECT-FROM-WHERE, bem como cláusulas GROUP BY e HAVING, se estas fizerem parte do bloco. Logo, as consultas aninhadas em uma consulta são identificadas como blocos de consulta separados. Como a SQL inclui operadores de agregação — como MAX, MIN, SUM e COUNT — esses operadores também precisam ser incluídos na álgebra estendida, conforme discutimos na Seção 6.4.

Considere a seguinte consulta SQL na relação FUNCIONARIO da Figura 3.5:

```
SELECT Unome, Pname
FROM FUNCIONARIO
WHERE Salario > ( SELECT MAX (Salario)
                    FROM FUNCIONARIO
                    WHERE Dnr=5 );
```

Essa consulta recupera os nomes dos funcionários (de qualquer departamento na empresa) que ganham um salário maior que o *maior salário no departamento 5*. A consulta inclui uma subconsulta aninhada e, portanto, seria decomposta em dois blocos. O bloco mais interno é:

```
( SELECT MAX (Salario)
  FROM FUNCIONARIO
  WHERE Dnr=5 )
```

Isso recupera o salário mais alto no departamento 5. O bloco de consulta mais externo é:

```
SELECT Unome, Pname
FROM FUNCIONARIO
WHERE Salario > c
```

onde c representa o resultado retornado do bloco interno. O bloco interno poderia ser traduzido para a seguinte expressão da álgebra relacional estendida:

$$\pi_{\text{Uname}, \text{Pname}}(\sigma_{\text{Dnr}=5}(\text{FUNCIONARIO}))$$

e o bloco externo para a expressão:

$$\pi_{\text{Uname}, \text{Pname}}(\sigma_{\text{Salario} > c}(\text{FUNCIONARIO}))$$

O *otimizador de consulta*, então, escolheria um plano de execução para cada bloco de consulta. Observe que, no exemplo acima, o bloco interno só precisa ser avaliado uma vez para produzir o salário máximo dos funcionários no departamento 5, que é então utilizado — como a constante c — pelo bloco externo. Chamamos isso de uma consulta aninhada (*sem correlação com a consulta externa*) na Seção 5.1.2. É muito mais difícil otimizar as *consultas aninhadas correlacionadas* mais complexas (ver Seção 5.1.3), em que uma variável de tupla do bloco de consulta externo aparece na cláusula WHERE do bloco de consulta interno.

## 19.2 Algoritmos para ordenação externa

A intercalação (sorting) é um dos principais algoritmos utilizados no processamento de consulta. Por exemplo, sempre que uma consulta SQL especifica uma cláusula ORDER BY, o resultado da consulta precisa ser ordenado. A intercalação também é um componente chave nos algoritmos ordenação-intercalação (sort-merge) usados para JUNÇÃO e outras operações (como UNIÃO e INTERSECÇÃO), e em algoritmos de eliminação de duplicata para a operação PROJEÇÃO (quando uma consulta SQL especifica a opção DISTINCT na cláusula SELECT). Discutiremos um desses algoritmos nesta seção. Observe que a intercalação de determinado arquivo pode ser evitada se um índice apropriado — como um índice primário ou de agrupamento (ver Capítulo 18) — existir no atributo de arquivo desejado para permitir o acesso ordenado aos registros no arquivo.

Intercalação externa refere-se a algoritmos de intercalação que são adequados para grandes arquivos de registros armazenados no disco que não cabem inteiramente na memória principal, como a maioria dos arquivos de banco de dados.<sup>3</sup> O algoritmo de classificação externa típico usa uma estratégia ordenação-intercalação (sort-merge), que começa classificando pequenos subarquivos — chamados pedaços — do arquivo principal e depois mescla os pedaços classificados, criando subarquivos classificados maiores, que, por sua vez, são intercalados. O algoritmo ordenação-intercalação (sort-merge), como outros algoritmos de banco de dados, exige espaço de buffer na memória principal, onde a classificação e mesclagem reais dos pedaços são realizadas.

<sup>3</sup> Algoritmos de classificação interna são adequados para classificação de estruturas de dados, como tabelas e listas, que podem caber inteiramente na memória principal. Esses algoritmos são descritos com detalhes em livros de estruturas de dados e algoritmos, e incluem técnicas como quick sort, heap sort, bubble sort e muitas outras. Não discutiremos esses algoritmos aqui.

O algoritmo básico, esboçado na Figura 19.2, consiste em duas fases: a fase de ordenação e a fase de intercalação. O espaço de buffer na memória principal faz parte do cache de SGBD — uma área na memória principal do computador que é controlada pelo SGBD. O espaço de buffer é dividido em buffers individuais, onde cada buffer tem o mesmo tamanho em bytes que o tamanho de um bloco de disco. Assim, um buffer pode manter o conteúdo de exatamente *um bloco de disco*.

Na fase de ordenação, os pedaços (ou partes) do arquivo que podem caber no espaço de buffer disponível são lidos para a memória principal, ordenados usando um algoritmo de ordenação *interno* e

gravados de volta para o disco como subarquivos ordenados (ou pedaços) temporários. O tamanho de cada pedaço e o número de pedaços iniciais ( $n_R$ ) são ditados pelo número de blocos de arquivo ( $b$ ) e o espaço de buffer disponível ( $n_B$ ). Por exemplo, se o número de buffers disponíveis da memória principal  $n_B = 5$  blocos de disco e o tamanho do arquivo  $b = 1.024$  blocos de disco, então  $n_R = \lfloor (b/n_B) \rfloor$  ou 205 pedaços iniciais cada, com tamanho de cinco blocos (exceto o último pedaço, que terá apenas quatro blocos). Logo, após a fase de ordenação, 205 pedaços ordenados (ou 205 subarquivos ordenados do arquivo original) são armazenados como subarquivos temporários no disco.

```
atribua  $i \leftarrow 1;$ 
       $j \leftarrow b;$           {tamanho do arquivo em blocos}
       $k \leftarrow n_B;$         {tamanho do buffer em blocos}
       $m \leftarrow \lceil (j/k) \rceil;$ 
```

#### (Fase de Ordenação)

```
enquanto ( $i \leq m$ )
faça {
    lê próximos  $k$  blocos do arquivo no buffer ou, se houver menos de  $k$  blocos
    restantes, então lê nos blocos restantes;
    ordena os registros no buffer e grava como um subarquivo temporário;
     $i \leftarrow i + 1;$ 
}
```

#### (Fase de Intercalação: intercala subarquivos até que apenas 1 permaneça)

```
atribua  $i \leftarrow 1;$ 
       $p \leftarrow \lceil \log_{k-1} m \rceil$   { $p$  é o número de passos para a fase de intercalação}
       $j \leftarrow m;$ 
enquanto ( $i \leq p$ )
faça {
     $n \leftarrow 1;$ 
     $q \leftarrow \lceil (j/(k-1)) \rceil;$       {número de subarquivos a gravar nesse passo}
    enquanto ( $n \leq q$ )
    faça {
        lê próximos  $k-1$  subarquivos ou subarquivos restantes (do passo anterior)
        um bloco por vez;
        intercala e grava como novo subarquivo um bloco por vez;
         $n \leftarrow n + 1;$ 
    }
     $j \leftarrow q;$ 
     $i \leftarrow i + 1;$ 
}
```

Figura 19.2

Esboço do algoritmo ordenação-intercalação (sort-merge) para ordenação externa.

Na fase de intercalação, os pedaços ordenados são intercalados usando um ou mais passos de intercalação. Cada passo de intercalação pode ter uma ou mais etapas de intercalação. O grau de intercalação ( $d_M$ ) é o número de subarquivos ordenados que podem ser mesclados em cada etapa de intercalação. Durante cada etapa de intercalação, um bloco de buffer é necessário para manter um bloco de disco de cada um dos subarquivos ordenados sendo intercalados, e um buffer adicional é necessário para manter um bloco de disco do resultado da intercalação, que produzirá um arquivo ordenado maior, que é o resultado da intercalação de vários subarquivos ordenados menores. Logo,  $d_M$  é o menor de  $(n_B - 1)$  e  $n_R$ , e o número de passos de intercalação é  $\lceil (\log_{d_M}(n_R)) \rceil$ . Em nosso exemplo, onde  $n_B = 5$ ,  $d_M = 4$  (intercalação quádrupla), de modo que os 205 pedaços iniciais ordenados seriam intercalados quatro de cada vez em cada etapa em 52 subarquivos ordenados maiores, ao final do primeiro passo de intercalação. Esses 52 arquivos ordenados são, então, intercalados quatro de cada vez em 13 arquivos ordenados, que são depois intercalados em quatro arquivos ordenados, e, por fim, para um arquivo totalmente ordenado, o que significa que *quatro passos* são necessários.

O desempenho do algoritmo ordenação-intercalação pode ser medido no número de leituras e gravações de bloco de disco (entre o disco e a memória principal) antes que a ordenação do arquivo inteiro seja concluída. A fórmula a seguir aproxima esse custo:

$$(2 * b) + (2 * b * (\log_{d_M} n_R))$$

O primeiro termo  $(2 * b)$  representa o número de acessos de bloco para a fase de ordenação, pois cada bloco de arquivo é acessado duas vezes: uma para leitura em um buffer da memória principal e uma para a gravação dos registros ordenados de volta ao disco, em um dos subarquivos ordenados. O segundo termo representa o número de acessos de bloco para a fase de intercalação. Durante cada passada de intercalação, uma quantidade de blocos de disco aproximadamente igual aos blocos do arquivo original  $b$  é lida e gravada. Como o número de passos de intercalação é  $(\log_{d_M} n_R)$ , obtemos um custo total de intercalação de  $(2 * b * (\log_{d_M} n_R))$ .

O número mínimo de buffers da memória principal necessários é  $n_B = 3$ , que produz um  $d_M$  de 2 e um  $n_R$  de  $\lceil (b/3) \rceil$ . O  $d_M$  mínimo de 2 gera o desempenho do pior caso do algoritmo, que é:

$$(2 * b) + (2 * (b * (\log_2 n_R))).$$

As próximas seções discutem os diversos algoritmos para as operações da álgebra relacional (ver Capítulo 6).

## 19.3 Algoritmos para operações SELEÇÃO e JUNÇÃO

### 19.3.1 Implementando a operação SELEÇÃO

Existem muitos algoritmos para executar uma operação SELEÇÃO, que é basicamente uma operação de pesquisa para localizar os registros em um arquivo de disco que satisfazem certa condição. Alguns dos algoritmos de pesquisa dependem de o arquivo ter caminhos de acesso específicos, e eles só podem se aplicar a determinados tipos de condições de seleção. Discutimos alguns dos algoritmos para implementar SELEÇÃO nesta seção. Usaremos as operações a seguir, especificadas no banco de dados relacional da Figura 3.5, para ilustrar nossa discussão:

- OP1:  $\sigma_{Cpf = '12345678966'}(\text{FUNCIONARIO})$
- OP2:  $\sigma_{Dnrnumero > 5}(\text{DEPARTAMENTO})$
- OP3:  $\sigma_{Dnr = 5}(\text{FUNCIONARIO})$
- OP4:  $\sigma_{Dnr = 5 \text{ AND } Salario > 30.000 \text{ AND } Sexo = 'F'}(\text{FUNCIONARIO})$
- OP5:  $\sigma_{Fcpf = '12345678966' \text{ AND } Pnr = 10}(\text{TRABALHA_EM})$

**Métodos de pesquisa para seleção simples.** Diversos algoritmos de pesquisa são possíveis para selecionar registros de um arquivo. Estes são conhecidos como varreduras de arquivo porque varrem os registros de um arquivo para procurar e recuperar registros que satisfazem uma condição de seleção.<sup>4</sup> Se o algoritmo de pesquisa envolve o uso de um índice, a pesquisa do índice é denominada varredura de índice. Os métodos de pesquisa a seguir (S1 a S6) são exemplos de alguns dos algoritmos de pesquisa que podem ser usados para implementar uma operação de seleção:

- **S1 — Pesquisa linear (algoritmo de força bruta).** Recupera *cada registro* no arquivo e testa se seus valores de atributo satisfazem a condição de seleção. Como os registros são agrupados em blocos de disco, cada um desses blocos é lido para um buffer da memória principal, e depois uma pesquisa pelos registros no bloco de disco é realizada na memória principal.
- **S2 — Pesquisa binária.** Se a condição de seleção envolver uma comparação de igualdade em um atributo chave no qual o arquivo é ordenado, a pesquisa binária — que é mais eficiente do que a pesquisa linear — pode ser utilizada. Um exemplo é OP1 se Cpf for o atributo de ordenação para o arquivo FUNCIONARIO.<sup>5</sup>

<sup>4</sup> Uma operação de seleção às vezes é chamada de **filtro**, pois ela filtra os registros no arquivo que não satisfazem a condição de seleção.

<sup>5</sup> Geralmente, a pesquisa binária não é usada em pesquisas de banco de dados porque os arquivos ordenados não são utilizados, a menos que também tenham um índice primário correspondente.

- **S3a — Usando um índice primário.** Se a condição de seleção envolver uma comparação de igualdade em um atributo chave com um índice primário — por exemplo, Cpf = '12345678966' em OP1 —, use o índice primário para recuperar o registro. Observe que essa condição recupera um único registro (no máximo).
- **S3b — Usando uma chave hash.** Se a condição de seleção envolver uma comparação de igualdade em um atributo chave com uma chave hash — por exemplo, Cpf = '12345678966' em OP1 —, use a chave hash para recuperar o registro. Observe que essa condição recupera um único registro (no máximo).
- **S4 — Usando um índice primário para recuperar vários registros.** Se a condição de comparação for  $>$ ,  $\geq$ ,  $<$  ou  $\leq$  em um campo chave com um índice primário — por exemplo, Dnumero  $> 5$  em OP2 —, use o índice para encontrar o registro que satisfaz a condição de igualdade correspondente (Dnumero = 5), depois recupere todos os registros subsequentes no arquivo (ordenado). Para a condição Dnumero  $< 5$ , recupere todos os registros anteriores.
- **S5 — Usando um índice de agrupamento para recuperar vários registros.** Se a condição de seleção envolver uma comparação de igualdade em um atributo não chave com um índice de agrupamento — por exemplo, Dnr = 5 em OP3 —, use o índice para recuperar todos os registros que satisfazem a condição.
- **S6 — Usando um índice secundário (B<sup>+</sup>-tree) em uma comparação de igualdade.** Este método de pesquisa pode ser utilizado para recuperar um único registro se o campo de índice for uma chave (tiver valores únicos) ou para recuperar múltiplos registros se o campo de índice não for uma chave. Este também pode ser usado para comparações envolvendo  $>$ ,  $\geq$ ,  $<$  ou  $\leq$ .

Na Seção 19.8, discutimos como desenvolver fórmulas que estimam o custo de acesso desses métodos de pesquisa em relação ao número de acessos de bloco e tempo de acesso. O método S1 (pesquisa linear) se aplica a qualquer arquivo, mas todos os outros métodos dependem de ter o caminho de acesso apropriado no atributo usado na condição de seleção. O método

S2 (pesquisa binária) exige que o arquivo seja ordenado no atributo de pesquisa. Os métodos que usam um índice (S3a, S4, S5 e S6) geralmente são conhecidos como **pesquisas de índice**, e exigem que exista um índice apropriado no atributo de pesquisa. Os métodos S4 e S6 podem ser usados para recuperar registros em certo *intervalo* — por exemplo,  $30.000 \leq \text{Salario} \leq 35.000$ . As consultas que envolvem tais condições são denominadas **consultas de intervalo**.

**Métodos de pesquisa para seleção complexa.** Se uma condição de uma operação SELEÇÃO for uma condição conjuntiva — ou seja, se ela for composta de várias condições simples ligadas com o conectivo lógico AND, como em OP4 acima —, o SGBD pode usar os seguintes métodos adicionais para implementar a operação:

- **S7 — Seleção conjuntiva usando um índice individual.** Se um atributo envolvido em qualquer condição simples isolada na condição de seleção conjuntiva tiver um caminho de acesso que permita o uso de um dos métodos S2 a S6, use essa condição para recuperar os registros e depois verificar se cada registro recuperado *satisfaz as condições simples restantes* na condição de seleção conjuntiva.
- **S8 — Seleção conjuntiva usando um índice composto.** Se dois ou mais atributos estiverem envolvidos nas condições de igualdade na condição de seleção conjuntiva e um índice composto (ou estrutura hash) existe nos campos combinados — por exemplo, se um índice tiver sido criado na chave composta (Fcpf, Pnr) do arquivo TRABALHA\_EM para OP5 —, podemos usar o índice diretamente.
- **S9 — Seleção conjuntiva por intersecção de ponteiros de registro.**<sup>6</sup> Se índices secundários (ou outros caminhos de acesso) estiverem disponíveis em mais de um dos campos envolvidos em condições simples na condição de seleção conjuntiva, e se os índices incluírem ponteiros de registro (em vez de ponteiros de bloco), então cada índice pode ser usado para recuperar o conjunto de ponteiros de registro que satisfaz a condição individual. A intersecção desses conjuntos de ponteiros de registros gera os ponteiros de registro que satisfazem a condição de seleção conjuntiva, que são então usados para recuperar esses registros diretamente. Se apenas algumas das

<sup>6</sup> Um ponteiro de registro identifica exclusivamente um registro e fornece seu endereço no disco; logo, ele é chamado de **identificador de registro**, ou **id de registro**.

condições tiverem índices secundários, cada registro recuperado é testado ainda mais para determinar se ele satisfaz as condições restantes.<sup>7</sup> Em geral, o método S9 assume que cada um dos índices está em um *campo não chave* do arquivo, pois se uma das condições for uma condição de igualdade em um campo chave, somente um registro satisfará a condição inteira.

Sempre que uma única condição especifica a seleção — como em OP1, OP2 ou OP3 —, o SGBD só pode verificar se existe ou não um caminho de acesso no atributo envolvido nessa condição. Se existir um caminho de acesso (como uma chave de índice ou hash ou um arquivo ordenado), o método correspondente a esse caminho de acesso é utilizado. Caso contrário, a técnica de força bruta ou pesquisa linear de S1 pode ser usada. A otimização de consulta para uma operação SELEÇÃO é necessária principalmente para condições de seleção conjuntivas sempre que *mais de um* dos atributos envolvidos nas condições tiver um caminho de acesso. O otimizador deve escolher o caminho de acesso que *recupera menos registros* da maneira mais eficiente, estimando os diferentes custos (ver Seção 19.8) e escolhendo o método com o menor custo estimado.

**Seletividade de uma condição.** Quando o otimizador está escolhendo entre várias condições simples em uma condição de seleção conjuntiva, ele normalmente considera a *seletividade* de cada condição. A *seletividade* (*sl*) é definida como a razão entre o número de registros (tuplas) que satisfazem a condição e o número total de registros (tuplas) no arquivo (relação), e, por isso, é um número entre zero e um. *Seletividade zero* significa que nenhum dos registros no arquivo satisfaz a condição de seleção, e uma seletividade de um significa que todos os registros no arquivo satisfazem a condição. Em geral, a seletividade não será um desses dois extremos, mas uma fração que estima a porcentagem dos registros do arquivo a serem recuperados.

Embora as seletividades exatas de todas as condições possam não estar disponíveis, estimativas de seletividades costumam ser mantidas no catálogo do SGBD e são usadas pelo otimizador. Por exemplo, para uma condição de igualdade em um atributo chave da relação  $r(R)$ ,  $s = 1/|r(R)|$ , onde  $|r(R)|$  é o número de tuplas na relação  $r(R)$ . Para uma condição de igualdade em um atributo não chave com  $i$  valo-

res distintos,  $s$  pode ser estimado por  $(|r(R)|/i)/|r(R)|$  ou  $1/i$ , supondo que os registros sejam igual ou uniformemente distribuídos entre os valores distintos.<sup>8</sup> Sob essa suposição,  $|r(R)|/i$  registros satisfarão uma condição de igualdade nesse atributo. Em geral, o número de registros satisfazendo uma condição de seleção com seletividade *sl* é estimado como sendo  $|r(R)| * sl$ . Quanto menor for essa estimativa, maior o desejo de usar tal condição primeiro para recuperar registros. Em certos casos, a distribuição real dos registros entre os diversos valores distintos do atributo é mantida pelo SGBD na forma de um *histograma*, a fim de obter estimativas mais precisas do número de registros que satisfazem determinada condição.

**Condições de seleção disjuntivas.** Em comparação com uma condição de seleção conjuntiva, uma condição *disjuntiva* (em que condições simples são ligadas pelo conectivo lógico OR em vez de AND) é muito mais difícil de processar e otimizar. Por exemplo, considere a OP4':

$$\text{OP4': } \sigma_{\text{Dnr}=5 \text{ OR } \text{Salario} > 30.000 \text{ OR } \text{Sexo} = 'P'}(\text{FUNCIONARIO})$$

Com tal condição, pouca otimização pode ser feita, pois os registros que satisfazem a condição disjuntiva são a *união* dos registros que satisfazem as condições individuais. Logo, se qualquer *uma* das condições não tiver um caminho de acesso, somos atraídos a usar a técnica de força bruta, de pesquisa linear. Somente se houver um caminho de acesso em *cada* condição simples na disjunção é que podemos otimizar a seleção recuperando os registros que satisfazem cada condição — ou seus ids de registro — e depois aplicando a operação de *união* para eliminar duplicatas.

Um SGBD terá à disposição muitos dos métodos já discutidos, e geralmente diversos métodos adicionais. O otimizador de consulta precisa escolher o método apropriado para executar cada operação SELEÇÃO em uma consulta. Essa otimização usa fórmulas que estimam os custos para cada método de acesso disponível, conforme discutiremos na Seção 19.8. O otimizador escolhe o método de acesso com o menor custo estimado.

### 19.3.2 Implementando a operação JUNÇÃO

A operação JUNÇÃO é uma das operações mais demoradas no processamento da consulta. Muitas das operações de junção encontradas nas consultas são das variedades EQUIJUNÇÃO e JUNÇÃO NATURAL, de modo que consideramos apenas essas duas aqui, pois

<sup>7</sup> A técnica pode ter muitas variações — por exemplo, se os índices forem *índices lógicos* que armazenam valores de chave primária em vez de ponteiros de registro.

<sup>8</sup> Em otimizadores mais sofisticados, histogramas que representam a distribuição dos registros entre os diferentes valores de atributo podem ser mantidos no catálogo.

só estamos dando uma visão geral do processamento e otimização da consulta. Para o restante deste capítulo, o termo junção refere-se a uma EQUIJUNÇÃO (ou JUNÇÃO NATURAL).

Existem muitas maneiras possíveis de implementar uma junção de duas vias, que é uma junção em dois arquivos. As junções que envolvem mais de dois arquivos são denominadas junções multivias. O número de vias possíveis para executar junções multivias cresce muito rapidamente. Nesta seção, discutimos as técnicas para implementar *apenas junções de duas vias*. Para ilustrar nossa discussão, referimo-nos ao esquema relacional da Figura 3.5 mais uma vez — especificamente, às relações FUNCIONARIO, DEPARTAMENTO e PROJETO. Os algoritmos que discutimos em seguida são para uma operação de junção na forma:

$$R \bowtie_{A=B} S$$

onde  $A$  e  $B$  são os atributos de junção, que devem ser atributos compatíveis por domínio de  $R$  e  $S$ , respectivamente. Os métodos que discutimos podem ser estendidos para formas mais gerais de junção. Ilustramos quatro das técnicas mais comuns para realizar tal junção, usando as seguintes operações como exemplo:

$$\text{OP6: } \text{FUNCIONARIO} \bowtie_{\text{Div}=\text{Divnumero}} \text{DEPARTAMENTO}$$

$$\text{OP7: } \text{DEPARTAMENTO} \bowtie_{\text{Cpf\_ger}=\text{Cpf}} \text{FUNCIONARIO}$$

### Métodos para implementar junções

- **J1 — Junção de loop aninhado (ou junção de bloco aninhado).** Esse é o algoritmo padrão (força bruta), pois não exige quaisquer caminhos de acesso especiais em qualquer arquivo na junção. Para cada registro  $t$  em  $R$  (loop externo), recupere cada registro  $s$  de  $S$  (loop interno) e teste se os dois registros satisfazem a condição de junção  $t[A] = s[B]$ .<sup>9</sup>
- **J2 — Junção de único loop (usando uma estrutura de acesso para recuperar os registros correspondentes).** Se houver um índice (ou chave hash) para um dos dois atributos de junção — digamos, atributo  $B$  do arquivo  $S$  —, recupere cada registro  $t$  em  $R$  (loop no arquivo  $R$ ) e depois use a estrutura de acesso (como um índice ou uma chave hash) para recuperar diretamente todos os registros correspondentes  $s$  de  $S$  que satisfazem  $s[B] = t[A]$ .

■ **J3 — Junção ordenação-intercalação.** Se os registros de  $R$  e  $S$  estiverem *fisicamente ordenados* por valor dos atributos de junção  $A$  e  $B$ , respectivamente, podemos implementar a junção da maneira mais eficiente possível. Os dois arquivos são varridos simultaneamente na ordem dos atributos de junção, combinando os registros que têm os mesmos valores para  $A$  e  $B$ . Se os arquivos não estiverem ordenados, eles podem sê-lo, primeiro, usando a ordenação externa (ver Seção 19.2). Nesse método, pares de blocos de arquivo são copiados para buffers de memória na ordem e os registros de cada arquivo são varridos apenas uma vez cada, para combinar com o outro arquivo — a menos que  $A$  e  $B$  sejam atributos não chave, caso em que o método precisa ser ligeiramente modificado. Um esboço do algoritmo de junção ordenação-intercalação aparece na Figura 19.3(a). Usamos  $R(i)$  para nos referirmos ao  $i$ -ésimo registro no arquivo  $R$ . Uma variação da junção ordenação-intercalação pode ser usada quando houver índices secundários nos dois atributos de junção. Os índices oferecem a capacidade de acessar (varrer) os registros na ordem dos atributos de junção, mas os próprios registros estão fisicamente espalhados por todos os blocos do arquivo, de modo que esse método pode ser muito ineficaz, pois cada acesso a registro pode envolver o acesso a um bloco de disco diferente.

■ **J4 — Junção de partição-hash.** Os registros dos arquivos  $R$  e  $S$  são particionados em arquivos menores. O particionamento de cada arquivo é feito usando a mesma função hashing  $h$  no atributo de junção  $A$  de  $R$  (para o particionamento do arquivo  $R$ ) e  $B$  de  $S$  (para o particionamento do arquivo  $S$ ). Primeiro, um único passo pelo arquivo com menos registros (digamos,  $R$ ) cria hashes de seus registros para as diversas partições de  $R$ ; essa é chamada fase de particionamento, pois os registros de  $R$  são particionados nos buckets de hash. No caso mais simples, consideramos que o arquivo menor pode caber inteiramente na memória principal depois de ser particionado, de modo que os subarquivos particionados de  $R$  são todos mantidos na memória principal. A coleção de registros com o mesmo valor de  $h(A)$  é colocada na mesma partição, que é um bucket de hash em

<sup>9</sup> Para arquivos de disco, é óbvio que os loops serão sobre blocos de disco, de modo que essa técnica também tem sido chamada de *junção de bloco aninhado*.

uma tabela hash na memória principal. Na segunda fase, chamada fase de investigação, um único passo por outro arquivo ( $S$ ) cria então o hash de cada um de seus registros usando a mesma função de hash  $h(B)$  para investigar o bucket apropriado, e tal registro é combinado com todos os registros correspondentes de  $R$  nesse bucket. Essa descrição simplificada da junção de partição-hash pressupõe que o menor dos dois arquivos *cabe inteiramente nos buckets de memória* após a primeira fase. Mais adiante, discutiremos o caso geral da junção de partição-hash que não requer esse pressuposto. Na prática, as técnicas de J1 a J4 são implementadas ao acessar *blocos de disco inteiros* de um arquivo, em vez de registros individuais. Dependendo do número disponível de buffers na memória, o número de blocos lidos do arquivo pode ser ajustado.

**Como o espaço do buffer e a escolha do arquivo de loop externo afetam o desempenho da junção de loop aninhado.** O espaço disponível do buffer tem um efeito importante em alguns dos algoritmos de junção. Primeiro, vamos considerar a técnica de loop aninhado (J1). Examinando novamente a operação OP6, suponha que o número de buffers disponíveis na memória principal para implementar a junção seja  $n_b = 7$  blocos (buffers). Lembre-se de que assumimos que cada buffer da memória tem o mesmo tamanho de um bloco de disco. Por ilustração, suponha que o arquivo DEPARTAMENTO consista em  $r_D = 50$  registros armazenados em  $b_D = 10$  blocos de disco e que o arquivo FUNCIONARIO consista em  $r_F = 6.000$  registros armazenados em  $b_F = 2.000$  blocos de disco. É vantajoso ler o máximo de blocos possíveis de uma só vez para a memória com base no arquivo, cujos registros são usados para o loop externo (ou seja,  $n_b - 2$  blocos). O algoritmo pode então ler um bloco de cada vez para o arquivo de loop interno e usar seus registros

(a) ordena as tuplas em  $R$  sobre atributo  $A$ ;  
 ordena as tuplas em  $S$  sobre atributo  $B$ ;  
 atribua  $i \leftarrow 1, j \leftarrow 1$ ;  
 enquanto  $(i \leq n)$  and  $(j \leq m)$   
 faça { se  $R(i)[A] > S(j)[B]$   
 então atribua  $j \leftarrow j + 1$   
 se não se  $R(i)[A] < S(j)[B]$   
 então atribua  $i \leftarrow i + 1$   
 se não { (\*  $R(i)[A] = S(j)[B]$ , de modo que enviamos uma tupla combinada \*)  
 envia a tupla combinada  $\langle R(i), S(j) \rangle$  para  $T$ ;  
 (\* envia outras tuplas que combinam com  $R(i)$ , se houver \*)  
 atribua  $i \leftarrow j + 1$ ;  
 enquanto  $(i \leq m)$  and  $(R(i)[A] = S(j)[B])$   
 faça { envia a tupla combinada  $\langle R(i), S(j) \rangle$  para  $T$ ;  
 atribua  $i \leftarrow i + 1$   
 }  
 (\* envia outras tuplas que combinam com  $S(j)$ , se houver \*)  
 atribua  $k \leftarrow i + 1$ ;  
 enquanto  $(k \leq n)$  and  $(R(k)[A] = S(j)[B])$   
 faça { envia a tupla combinada  $\langle R(k), S(j) \rangle$  para  $T$ ;  
 atribua  $k \leftarrow k + 1$   
 }  
 atribua  $i \leftarrow k, j \leftarrow i$   
}

(continua)

**Figura 19.3**

Implementando JUNÇÃO, PROJEÇÃO, UNION, INTERSECÇÃO e DIFERENÇA DE CONJUNTO ao usar ordenação-intercalação, onde  $R$  tem  $n$  tuplas e  $S$  tem  $m$  tuplas. (a) Implementando a operação  $T \leftarrow R \bowtie_{A=B} S$ .

- (b) cria uma tupla  $t[<\text{lista atributos}>]$  em  $T'$  para cada tupla  $t$  em  $R$ ;  
 (\*  $T'$  contém os resultados da projeção antes da eliminação de duplicatas \*)  
 se  $<\text{lista atributos}>$  inclui uma chave de  $R$   
 então  $T \leftarrow T'$   
 se não { ordena as tuplas em  $T'$  ;  
 atribua  $i \leftarrow 1, j \leftarrow 2$ ;  
 enquanto  $i \leq n$   
 faça { envia a tupla  $T'[i]$  para  $T$ ;  
 enquanto  $T'[i] = T'[j]$  e  $j \leq n$  do  $j \leftarrow j + 1$ ; (\* elimina duplicatas \*)  
 $i \leftarrow j; j \leftarrow i + 1$   
}  
}  
(\*  $T$  contém o resultado da projeção após a eliminação de tuplas \*)
- (c) ordena as tuplas em  $R$  e  $S$  usando os mesmos atributos de ordenação únicos;  
 atribua  $i \leftarrow 1, j \leftarrow 1$ ;  
 enquanto ( $i \leq n$ ) and ( $j \leq m$ )  
 faça { se  $R(i) > S(j)$   
 então { envia  $S(j)$  para  $T$ ;  
 atribua  $j \leftarrow j + 1$   
}  
se não se  $R(i) < S(j)$   
então { envia  $R(i)$  para  $T$ ;  
atribua  $i \leftarrow i + 1$   
}  
se não atribua  $j \leftarrow j + 1$  (\*  $R(i) = S(j)$ , e pulamos uma das tuplas duplicadas \*)  
}  
se ( $i \leq n$ ) então acrescenta tuplas  $R(i)$  em  $R(n)$  para  $T$ ;  
se ( $j \leq m$ ) então acrescenta tuplas  $S(j)$  em  $S(m)$  para  $T$ ;
- (d) ordena as tuplas em  $R$  e  $S$  usando os mesmos atributos de ordenação únicos;  
 atribua  $i \leftarrow 1, j \leftarrow 1$ ;  
 enquanto ( $i \leq n$ ) and ( $j \leq m$ )  
 faça { se  $R(i) > S(j)$   
 então atribua  $j \leftarrow j + 1$   
 se não se  $R(i) < S(j)$   
 então atribua  $i \leftarrow i + 1$   
 se não { envia  $R(j)$  para  $T$ ; (\*  $R(i) = S(j)$ , de modo que enviamos a tupla \*)  
 atribua  $i \leftarrow i + 1, j \leftarrow j + 1$   
}  
}  
}  

(e) ordena as tuplas em  $R$  e  $S$  usando os mesmos atributos de ordenação únicos;  
 atribua  $i \leftarrow 1, j \leftarrow 1$ ;  
 enquanto ( $i \leq n$ ) and ( $j \leq m$ )  
 faça { se  $R(i) > S(j)$   
 então atribua  $j \leftarrow j + 1$   
 se não se  $R(i) < S(j)$   
 então { envia  $R(i)$  para  $T$ ; (\*  $R(i)$  não tem  $S(j)$  combinando, e enviamos  $R(i)$  \*)  
 atribua  $i \leftarrow i + 1$   
}  
se não atribua  $i \leftarrow i + 1, j \leftarrow j + 1$   
}  
se ( $i \leq n$ ) então acrescenta tuplas  $R(i)$  em  $R(n)$  para  $T$ ;

Figura 19.3 (continuação)

(b) Implementando a operação  $T \leftarrow \pi_{<\text{lista atributos}>} (R)$ . (c) Implementando a operação  $T \leftarrow R \cup S$ . (d) Implementando a operação  $T \leftarrow R \cap S$ . (e) Implementando a operação  $T \leftarrow R - S$ .

para investigar (ou seja, pesquisar) os blocos do loop externo que estão atualmente na memória principal para combinação dos registros. Isso reduz o número total de acessos a bloco. Um buffer extra na memória principal é necessário para conter os registros resultantes após serem juntados, e o conteúdo desse buffer de resultado pode ser anexado ao arquivo de resultado — o arquivo de disco que conterá o resultado da junção — sempre que ele for preenchido. Esse bloco de buffer de resultado é então reutilizado para manter registros adicionais de resultado de junção.

Na junção de loop aninhado, faz diferença qual arquivo é escolhido para o loop externo e qual o é para o loop interno. Se FUNCIONARIO for usado para o loop externo, cada bloco de FUNCIONARIO é lido uma vez, e o arquivo DEPARTAMENTO inteiro (cada um de seus blocos) é lido uma vez para *cada vez* que lemos  $(n_B - 2)$  blocos do arquivo FUNCIONARIO. Obtemos as seguintes fórmulas para o número de blocos de disco que são lidos do disco para a memória principal:

Número total de blocos acessados (lidos) para o arquivo de loop externo =  $b_F$

Número de vezes  $(n_B - 2)$  que os blocos do arquivo externo são carregados para a memória principal =  $\lfloor b_F / (n_B - 2) \rfloor$

Número total de blocos acessados (lidos) para o arquivo de loop interno =  $b_D * \lfloor b_F / (n_B - 2) \rfloor$

Logo, obtemos o seguinte número total de acessos para leitura de bloco:

$$b_F + (\lfloor b_F / (n_B - 2) \rfloor * b_D) = 2.000 + (\lfloor (2.000 / 5) \rfloor * 10) = 6.000 \text{ acessos de bloco}$$

Por sua vez, se usarmos os registros de DEPARTAMENTO no loop externo, por simetria, obtemos o seguinte número total de acessos de bloco:

$$b_D + (\lfloor b_D / (n_B - 2) \rfloor * b_F) = 10 + (\lfloor (10 / 5) \rfloor * 2.000) = 4.010 \text{ acessos de bloco}$$

O algoritmo de junção usa um buffer para manter os registros juntados do arquivo de resultado. Quando o buffer estiver preenchido, ele será gravado no disco e seu conteúdo anexado ao arquivo de resultado, e depois preenchido novamente com os registros do resultado da junção.<sup>10</sup>

Se o arquivo de resultado da operação de junção tem  $b_{RES}$  blocos de disco, cada bloco é gravado uma vez no disco, de modo que  $b_{RES}$  acessos de bloco (gravações) adicionais devem ser acrescentados às fórmulas anteriores a fim de estimar o custo total da operação de junção. O mesmo vale para as fórmulas

desenvolvidas mais adiante para outros algoritmos de junção. Como este exemplo mostra, é vantajoso usar o arquivo *com menos blocos* como arquivo de loop externo na junção de loop aninhado.

**Como o fator de seleção de junção afeta o desempenho da junção.** Outro fator que afeta o desempenho de uma junção, particularmente o método de único loop J2, é a fração de registros em um arquivo que será juntada com registros no outro arquivo. Chamamos isso de fator de seleção de junção<sup>11</sup> de um arquivo em relação a uma condição de equijunção com outro arquivo. Esse fator depende da condição de equijunção em particular entre os dois arquivos. Para ilustrar isso, considere a operação OP7, que junta cada registro de DEPARTAMENTO com o registro de FUNCIONARIO para o gerente desse departamento. Aqui, cada registro de DEPARTAMENTO (existem 50 desses registros em nosso exemplo) será juntado a um *único* registro de FUNCIONARIO, mas muitos registros de FUNCIONARIO (os 5.950 deles, que não gerenciam um departamento) não serão juntados com qualquer registro de DEPARTAMENTO.

Suponha que existam índices secundários nos atributos Cpf de FUNCIONARIO e Cpf\_ger de DEPARTAMENTO, com o número de níveis de índice  $x_{Cpf} = 4$  e  $x_{Cpf\_ger} = 2$ , respectivamente. Temos duas opções para implementar o método J2. A primeira recupera cada registro de FUNCIONARIO e depois usa o índice em Cpf\_ger de DEPARTAMENTO para encontrar um registro de DEPARTAMENTO correspondente. Nesse caso, nenhum registro combinando será encontrado para funcionários que não gerenciam um departamento. O número de acessos de bloco para esse caso é aproximadamente:

$$b_F + (r_F * (x_{Cpf\_ger} + 1)) = 2.000 + (6.000 * 3) = 20.000 \text{ acessos de bloco}$$

A segunda opção recupera cada registro de DEPARTAMENTO e depois usa o índice em Cpf de FUNCIONARIO para encontrar um registro de FUNCIONARIO gerente correspondente. Nesse caso, cada registro de DEPARTAMENTO terá um registro de FUNCIONARIO correspondente. O número de acessos de bloco para esse caso é aproximadamente:

$$b_D + (r_D * (x_{Cpf} + 1)) = 10 + (50 * 5) = 260 \text{ acessos de bloco}$$

A segunda opção é mais eficiente porque o fator de seleção de junção de DEPARTAMENTO *em relação à condição de junção Cpf = Cpf\_ger* é 1 (cada registro em DEPARTAMENTO será juntado), enquanto o fator

<sup>10</sup> Se reservarmos dois buffers para o arquivo de resultado, o buffering duplo pode ser usado para agilizar o algoritmo (ver Seção 17.3).

<sup>11</sup> Isso é diferente da seletividade de junção, que discutiremos na Seção 19.8.

de seleção de junção de FUNCIONARIO em relação à mesma condição de junção é  $(50/6.000)$ , ou 0,008 (apenas 0,8 por cento dos registros em FUNCIONARIO serão juntados). Para o método J2, o menor arquivo ou o arquivo que tem uma correspondência para cada registro (ou seja, o arquivo com o fator de seleção de junção alto) deve ser usado no (único) loop de junção. Também é possível criar um índice especificamente para realizar a operação de junção se ainda não houver um.

A junção ordenação-intercalação J3 é muito eficiente se os dois arquivos já estiverem ordenados por seu atributo de junção. Somente um único passo é feito em cada arquivo. Logo, o número de blocos acessados é igual à soma dos números de blocos nos dois arquivos. Para esse método, tanto OP6 quanto OP7 precisariam de  $b_F + b_D = 2.000 + 10 = 2.010$  acessos de bloco. No entanto, os dois arquivos precisam estar ordenados pelos atributos de junção; se um ou ambos não estiverem, deve ser criada uma cópia ordenada de cada arquivo especificamente para realizar a operação de junção. Se estimarmos aproximadamente o custo de ordenação de um arquivo externo por  $(b \log_2 b)$  acessos de bloco, e se os dois arquivos precisarem ser ordenados, o custo total de uma junção ordenação-intercalação pode ser estimado por  $(b_F + b_D + b_F \log_2 b_F + b_D \log_2 b_D)$ .<sup>12</sup>

**Caso geral para a junção de partição-hash.** O método de junção de hash J4 também é bastante eficiente. Nesse caso, apenas um único passo é feito em cada arquivo, estejam os arquivos ordenados ou não. Se a tabela hash para o menor dos dois arquivos puder ser mantida na memória principal após o hashing (particionamento) em seu atributo de junção, a implementação é simples. Porém, se as partições de ambos os arquivos tiverem de ser armazenadas em disco, o método torna-se mais complexo, e diversas variações para melhorar a eficiência foram propostas. Discutimos duas técnicas: o caso geral de *junção de partição-hash* e uma variação chamada *algoritmo híbrido de junção de hash*, que demonstrou ser muito eficiente.

No caso geral da junção de partição-hash, cada arquivo é primeiro particionado em  $M$  partes usando a mesma função hash de particionamento nos atributos de junção. Depois, cada par de partições correspondentes é juntado. Por exemplo, suponha que estejamos juntando relações  $R$  e  $S$  nos atributos de junção  $R.A$  e  $S.B$ :

$$R \bowtie_{A=B} S$$

Na fase de particionamento,  $R$  é particionado nas  $M$  partições  $R_1, R_2, \dots, R_M$ , e  $S$  nas  $M$  partições  $S_1, S_2, \dots, S_M$ . A propriedade de cada par de partições correspondentes  $R_i, S_j$  em relação à operação de junção é que os registros em  $R_i$  só precisam ser juntados com registros em  $S_j$ , e vice-versa. Essa propriedade é garantida usando a mesma função hash para particionar os dois arquivos em seus atributos de junção — atributo  $A$  para  $R$  e atributo  $B$  para  $S$ . O número mínimo de buffers na memória necessários para a fase de particionamento é  $M + 1$ . Cada um dos arquivos  $R$  e  $S$  é particionado separadamente. Durante o particionamento de um arquivo,  $M$  buffers na memória são alocados para armazenar os registros que criam hash para cada partição, e um buffer adicional é necessário para manter um bloco de cada vez do arquivo de entrada que está sendo particionado. Sempre que o buffer na memória para uma partição é preenchido, seu conteúdo é anexado a um subarquivo de disco que armazena a partição. A fase de particionamento tem *duas iterações*. Após a primeira iteração, o primeiro arquivo  $R$  é particionado nos subarquivos  $R_1, R_2, \dots, R_M$ , nos quais todos os registros que tiveram hash para o mesmo buffer estão na mesma partição. Após a segunda iteração, o segundo arquivo  $S$  é particionado de modo semelhante.

Na segunda fase, chamada fase de junção ou investigação,  $M$  iterações são necessárias. Durante a iteração  $i$ , duas partições correspondentes  $R_i$  e  $S_i$  são juntadas. O número mínimo de buffers necessários para a iteração  $i$  é o número de blocos na menor das duas partições, digamos  $R_i$ , mais dois buffers adicionais. Se usarmos uma junção de loop aninhado durante a iteração  $i$ , os registros da menor das duas partições  $R_i$  são copiados para buffers da memória; depois, todos os blocos da outra partição  $S_i$  são lidos — um de cada vez — e cada registro é usado para investigar (ou seja, pesquisar) a partição  $R_i$  em busca de registro(s) correspondente(s). Quaisquer registros correspondentes são juntados e gravados no arquivo de resultado. Para melhorar a eficiência da investigação na memória, é comum usar uma *tabela hash na memória* para armazenar os registros na partição  $R_i$  usando uma função hash diferente da função hash de particionamento.<sup>13</sup>

Podemos aproximar o custo dessa junção de partição-hash como  $3 * (b_R + b_S) + b_{RES}$  para nosso exemplo, pois cada registro é lido uma vez e gravado de volta no disco uma vez durante a fase de particionamento. No decorrer da fase de junção (investigação),

<sup>12</sup> Podemos usar as fórmulas mais exatas da Seção 19.2 se soubermos o número de buffers disponíveis para ordenação.

<sup>13</sup> Se a função hash para o particionamento for usada novamente, todos os registros em uma partição criam hash para o mesmo bucket novamente.

cada registro é lido uma segunda vez para realizar a junção. A *principal dificuldade* desse algoritmo é garantir que a função hash de particionamento seja *uniforme* — ou seja, os tamanhos de partição são quase iguais em tamanho. Se a função de particionamento for *viesada* (não uniforme), então algumas partições podem ser muito grandes para caber no espaço de memória disponível para a segunda fase de junção.

Observe que, se o espaço de buffer disponível na memória  $n_B > (b_R + 2)$ , onde  $b_R$  é o número de blocos para o *menor* dos dois arquivos sendo juntados, digamos,  $R$ , então não existe motivo para realizar o particionamento, pois nesse caso a junção pode ser realizada inteiramente na memória, usando alguma variação da junção de loop aninhado baseada no hashing e na investigação.

Por exemplo, suponha que estejamos realizando a operação de junção OP6, repetida a seguir:

OP6: FUNCIONARIO  $\bowtie_{\text{Div}=\text{Departamento}} \text{DEPARTAMENTO}$

Nesse exemplo, o arquivo menor é o arquivo DEPARTAMENTO; logo, se o número de buffers de memória disponíveis  $n_B > (b_D + 2)$ , o arquivo DEPARTAMENTO inteiro poderá ser lido para a memória principal e organizado em uma tabela hash no atributo de junção. Cada bloco de FUNCIONARIO é então lido para um buffer, e cada registro de FUNCIONARIO no buffer tem um hash em seu atributo de junção e é usado para *investigar* o bucket correspondente na memória, na tabela hash DEPARTAMENTO. Se um registro correspondente for achado, os registros são juntados, e os registros do resultado são gravados no buffer de resultado e, por fim, no arquivo de resultado em disco. O custo em termos de acessos de bloco é, portanto,  $(b_D + b_F)$ , mais  $b_{\text{RES}}$  — o custo de gravar o arquivo de resultado.

**Junção de hash híbrida.** O algoritmo junção hash híbrido é uma variação de partição de junção hash, em que a fase de *junção* para *uma das partições* está incluída na fase de *particionamento*. Para ilustrar isso, vamos supor que o tamanho de um buffer de memória seja um bloco de disco; que  $n_B$  de tais buffers estejam *disponíveis*; e que a função hash de particionamento utilizada seja  $h(K) = K \bmod M$ , de modo que  $M$  partições estejam sendo criadas, onde  $M < n_B$ . Por exemplo, suponha que estejamos realizando a operação de junção OP6. No *primeiro passo* da fase de particionamento, quando o algoritmo híbrido de junção hash está particionando o menor dos dois arquivos (DEPARTAMENTO em OP6), o algoritmo divide o espaço do buffer entre as  $M$  partições de modo que todos os blocos da *primeira partição* de DEPARTAMENTO residam completamente na memória principal. Para cada uma das outras partições, somente

um único buffer na memória — cujo tamanho é de um bloco de disco — é alocado; o restante da partição é gravado em disco, como na junção normal de partição-hash. Logo, ao final do *primeiro passo da fase de particionamento*, a primeira partição de DEPARTAMENTO reside inteiramente na memória principal, enquanto cada uma das outras partições de DEPARTAMENTO reside em um subarquivo do disco.

Para o segundo passo da fase de particionamento, os registros do segundo arquivo sendo juntado — o arquivo maior, FUNCIONARIO em OP6 — estão sendo particionados. Se um registro gera um hash para a *primeira partição*, ele é juntado com o registro correspondente em DEPARTAMENTO e os registros juntados são gravados no buffer de resultado (e, por fim, no disco). Se um registro de FUNCIONARIO gera um hash para uma partição que não seja a primeira, ele é particionado normalmente e armazenado no disco. Logo, ao final do segundo passo da fase de particionamento, todos os registros que geram hash para a primeira partição foram juntados. Nesse ponto, existem  $M - 1$  pares de partições no disco. Portanto, durante a segunda fase de junção ou investigação,  $M - 1$  iterações são necessárias, em vez de  $M$ . O objetivo é juntar o máximo de registros durante a fase de particionamento de modo a economizar o custo de armazenar esses registros no disco e depois lê-los pela segunda vez durante a fase de junção.

## 19.4 Algoritmos para operações PROJEÇÃO e de conjunto

Uma operação PROJEÇÃO  $\pi_{\langle\text{lista atributos}\rangle}(R)$  é simples de se implementar se *<lista atributos>* incluir uma chave da relação  $R$ , pois nesse caso o resultado da operação terá o mesmo número de tuplas que  $R$ , mas com apenas os valores para os atributos em *<lista atributos>* em cada tupla. Se *<lista atributos>* não incluir uma chave de  $R$ , as *tuplas duplicadas devem ser eliminadas*. Isso pode ser feito ao ordenar o resultado da operação e depois ao eliminar tuplas duplicadas, que aparecem consecutivamente após a ordenação. Um esboço do algoritmo aparece na Figura 19.3(b). O hashing também pode ser usado para eliminar duplicatas: à medida que cada registro gera um hash e é inserido em um bucket do arquivo hash na memória, ele é comparado com os registros que já estão no bucket; se for uma duplicata, ele não é inserido no bucket. É útil lembrar aqui que, nas consultas SQL, o padrão é não eliminar duplicatas do resultado da consulta; estas só são eliminadas do resultado da consulta se a palavra-chave DISTINCT for incluída.

Operações de conjunto — UNIÃO, INTERSECÇÃO, DIFERENÇA DE CONJUNTO e PRODUTO CARTESIANO — às vezes são dispendiosas de se implementar. Em particular, a operação de PRODUTO CARTESIANO  $R \times S$  é muito dispendiosa porque seu resultado inclui um registro para cada combinação de registros de  $R$  e  $S$ . Além disso, cada registro no resultado inclui todos os atributos de  $R$  e  $S$ . Se  $R$  tem  $n$  registros e  $j$  atributos, e  $S$  tem  $m$  registros e  $k$  atributos, a relação de resultado para  $R \times S$  terá  $n * m$  registros e cada registro terá  $j + k$  atributos. Logo, é importante evitar a operação PRODUTO CARTESIANO e substituí-la por outras operações, como a junção, durante a otimização da consulta (ver Seção 19.7).

As outras três operações de conjunto — UNIÃO, INTERSECÇÃO e DIFERENÇA DE CONJUNTO<sup>14</sup> — só se aplicam a relações compatíveis no tipo (ou compatíveis na união), que têm o mesmo número de atributos e os mesmos domínios de atributo. O modo comum de implementar essas operações é usar variações da técnica de ordenação-intercalação: as duas relações são ordenadas nos mesmos atributos e, depois da ordenação, uma única varredura por cada relação é suficiente para produzir o resultado. Por exemplo, podemos implementar a operação UNIÃO,  $R \cup S$ , varrendo e intercalando os dois arquivos ordenados simultaneamente, e, sempre que a mesma tupla existir nas duas relações, apenas uma é mantida no resultado intercalado. Para a operação INTERSECÇÃO,  $R \cap S$ , mantemos no resultado intercalado somente as tuplas que aparecem nas *duas relações ordenadas*. Da Figura 19.3(c) até a (e) há um esboço da implementação dessas operações pela ordenação e intercalação. Alguns dos detalhes não estão incluídos nesses algoritmos.

O hashing também pode ser usado para implementar UNIÃO, INTERSECÇÃO e DIFERENÇA DE CONJUNTO. Primeiro, uma tabela é varrida e depois particionada em uma tabela hash na memória com buckets, e os registros na outra tabela são então varridos um de cada vez e usados para investigar a partição apropriada. Por exemplo, para implementar  $R \cup S$ , primeiro crie o hash (particione) dos registros de  $R$ ; depois, use o hash (investigue) dos registros de  $S$ , mas não insira registros duplicados nos buckets. Para implementar  $R \cap S$ , primeiro particione os registros de  $R$  para o arquivo hash. Depois, enquanto realiza o hashing de cada registro de  $S$ , investigue para verificar se um registro idêntico de  $R$  existe no bucket e, se houver, acrescente o registro no arquivo de resultado. Para implementar  $R - S$ , primeiro crie o hash dos registros de  $R$  para os buckets do arquivo hash. Ao realizar o hashing de (investigar) cada registro de

$S$ , se um registro idêntico for encontrado no bucket, remova esse registro do bucket.

Em SQL, existem duas variações dessas operações de conjunto. As operações UNION, INTERSECTION e EXCEPT (a palavra-chave SQL para a operação SET DIFFERENCE) se aplicam a conjuntos tradicionais, onde não existe registro duplicado no resultado. As operações UNION ALL, INTERSECTION ALL e EXCEPT ALL se aplicam a multiconjuntos (ou bags), e as duplicatas são totalmente consideradas. As variações dos algoritmos citados podem ser usadas para as operações de multiconjunto em SQL. Deixamos estas como um exercício para o leitor.

## 19.5 Implementando operações de agregação e JUNÇÃO EXTERNA

### 19.5.1 Implementando operações de agregação

Os operadores de agregação (MIN, MAX, COUNT, AVERAGE, SUM), quando aplicados a uma tabela inteira, podem ser calculados por uma varredura de tabela ou usando um índice apropriado, se houver. Por exemplo, considere a seguinte consulta SQL:

```
SELECT MAX(Salario)
FROM FUNCIONARIO;
```

Se houver um índice de B<sup>+</sup>-tree (crescente) em Salario para a relação FUNCIONARIO, então o otimizador pode decidir sobre o uso do índice Salario para procurar o maior valor de Salario no índice, seguindo o ponteiro *mais à direita* em cada nó índice da raiz até a folha mais à direita. Esse nó incluiria o maior valor de Salario como sua *última* entrada. Na maioria dos casos, isso seria mais eficiente do que uma varredura completa da tabela FUNCIONARIO, pois nenhum registro real precisa ser recuperado. A função MIN pode ser tratada de maneira semelhante, com a exceção de que o ponteiro *mais à esquerda* no índice é seguido da raiz até a folha mais à esquerda. Esse nó incluiria o menor valor de Salario como sua *primeira* entrada.

O índice também poderia ser usado para as funções de agregação AVERAGE e SUM, mas somente se for um índice denso — ou seja, se houver uma entrada de índice para cada registro no arquivo principal. Nesse caso, o cálculo associado seria aplicado aos valores no índice. Para um índice não denso, o número real de registros associados a cada valor de índice deve ser utilizado para um cálculo correto.

<sup>14</sup> DIFERENÇA DE CONJUNTO é chamada de EXCEPT em SQL.

Isso pode ser feito se o *número de registros associados a cada valor* no índice for armazenado em cada entrada de índice. Para a função de agregação COUNT, o número de valores também pode ser calculado com base no índice de modo semelhante. Se uma função COUNT(\*) for aplicada a uma relação inteira, o número de registros atualmente em cada relação costuma ser armazenado no catálogo e, portanto, o resultado pode ser recuperado diretamente do catálogo.

Quando uma cláusula GROUP BY é usada em uma consulta, o operador de agregação deve ser aplicado separadamente a cada grupo de tuplas, conforme particionado pelo atributo de agrupamento. Logo, a tabela precisa primeiro ser particionada em subconjuntos de tuplas, nos quais cada partição (grupo) tem o mesmo valor para os atributos de agrupamento. Nesse caso, o cálculo é mais complexo. Considere a seguinte consulta:

```
SELECT Dnr, AVG(Salario)
FROM FUNCIONARIO
GROUP BY Dnr;
```

A técnica comum para tais consultas é primeiro usar a ordenação ou o hashing nos atributos de agrupamento para particionar o arquivo nos grupos apropriados. Depois, o algoritmo calcula a função de agregação para as tuplas em cada grupo, que têm o mesmo valor de atributo(s) de agrupamento. Na consulta de exemplo, o conjunto de tuplas de FUNCIONARIO para cada número de departamento seria agrupado em uma partição e o salário médio, calculado para cada grupo.

Observe que, se houver um índice de agrupamento (ver Capítulo 18) no(s) atributo(s) de agrupamento, então os registros já estão particionados (agrupados) nos subconjuntos apropriados. Nesse caso, só é preciso aplicar o cálculo a cada grupo.

### 19.5.2 Implementando JUNÇÃO EXTERNA

Na Seção 6.4, a operação de junção externa foi discutida, com suas três variações: junção externa esquerda (*left outer join*), junção externa direita (*right outer join*) e junção externa completa (*full outer join*). Também discutimos, no Capítulo 5, como essas operações podem ser especificadas em SQL. A seguir vemos um exemplo de uma operação de junção externa esquerda em SQL:

```
SELECT Uname, Pname, Dname
FROM (FUNCIONARIO LEFT OUTER JOIN
DEPARTAMENTO ON Dnr=Dnumero);
```

O resultado dessa consulta é uma tabela de nomes de funcionário e seus departamentos associados. Ele é semelhante ao resultado de uma junção nor-

mal (interna), com a exceção de que, se uma tupla de FUNCIONARIO (uma tupla na relação *esquerda*) não tiver um departamento associado, o nome do funcionário ainda aparecerá na tabela resultante, mas o nome do departamento seria NULL para tais tuplas no resultado da consulta.

A junção externa pode ser calculada modificando-se um dos algoritmos de junção, como a junção de loop aninhado ou a junção de único loop. Por exemplo, para calcular uma junção externa *esquerda*, usamos a relação esquerda como loop externo ou único loop, pois cada tupla na relação esquerda deve aparecer no resultado. Se houver tuplas correspondentes na outra relação, as tuplas juntadas são produzidas e salvas no resultado. Contudo, se nenhuma tupla correspondente for encontrada, a tupla ainda é incluída no resultado, mas é preenchida com valor(es) NULL. Os algoritmos ordenação-intercalação e junção hash também podem ser estendidos para calcular junções externas.

Teoricamente, a junção externa também pode ser calculada ao executar uma combinação de operadores da álgebra relacional. Por exemplo, a operação de junção externa à esquerda, mostrada acima, é equivalente à seguinte sequência de operações relacionais:

1. Calcule a JUNÇÃO (interna) das tabelas FUNCIONARIO e DEPARTAMENTO.  
 $\text{TEMP1} \leftarrow \pi_{\text{Uname}, \text{Pname}, \text{Dname}} (\text{FUNCIONARIO} \bowtie_{\text{Dnr}=\text{Dnumero}} \text{DEPARTAMENTO})$
2. Ache as tuplas de FUNCIONARIO que não aparecem no resultado da JUNÇÃO (interna).  
 $\text{TEMP2} \leftarrow \pi_{\text{Uname}, \text{Pname}} (\text{FUNCIONARIO}) - \pi_{\text{Uname}, \text{Pname}} (\text{TEMP1})$
3. Preencha cada tupla em TEMP2 com um campo Dname NULL.  
 $\text{TEMP2} \leftarrow \text{TEMP2} \times \text{NULL}$
4. Aplique a operação UNION a TEMP1, TEMP2 para produzir o resultado JUNÇÃO EXTERNA A ESQUERDA.  
 $\text{RESULT} \leftarrow \text{TEMP1} \cup \text{TEMP2}$

O custo da junção externa, conforme calculado anteriormente, seria a soma dos custos das etapas associadas (junção interna, projeções, diferença de conjunto e união). Porém, observe que a etapa 3 pode ser feita enquanto a relação temporária está sendo construída na etapa 2; ou seja, podemos simplesmente preencher cada tupla resultante com um NULL. Além disso, na etapa 4, sabemos que os dois operandos da união são disjuntos (sem tuplas comuns), de modo que não há necessidade de eliminação de duplicatas.

## 19.6 Combinando operações com pipelining

Uma consulta especificada em SQL normalmente será traduzida para uma expressão da álgebra relacional que é *uma sequência de operações relacionais*. Se executarmos uma única operação de cada vez, temos de gerar arquivos temporários no disco para manter os resultados dessas operações temporárias, criando um overhead excessivo. Gerar e armazenar grandes arquivos temporários em disco é demorado e pode ser desnecessário em muitos casos, uma vez que esses arquivos serão imediatamente usados como entrada para a próxima operação. Para reduzir o número de arquivos temporários, é comum gerar um código de execução de consulta que corresponde a algoritmos para combinações de operações em uma consulta.

Por exemplo, em vez de ser implementada separadamente, uma JUNÇÃO pode ser combinada com duas operações SELEÇÃO nos arquivos de entrada e uma operação PROJEÇÃO final no arquivo resultante; tudo isso é implementado por um algoritmo com dois arquivos de entrada e um único arquivo de saída. Em vez de criar quatro arquivos temporários, aplicamos o algoritmo diretamente e recebemos apenas um arquivo de resultado. Na Seção 19.7.2, discutimos como a otimização da álgebra relacional heurística pode agrupar operações para execução. Isso é chamado de *pipelining ou processamento baseado em fluxo*.

É usual criar o código de execução de consulta de maneira dinâmica para implementar múltiplas operações. O código gerado para produzir a consulta combina vários algoritmos que correspondem a operações individuais. À medida que são produzidas tuplas de resultado de uma operação, elas são fornecidas como entrada para operações subsequentes. Por exemplo, se uma operação de junção segue duas operações de seleção em relações da base, as tuplas resultantes de cada seleção são fornecidas como entrada para o algoritmo de junção em um fluxo ou pipeline à medida que são produzidas.

## 19.7 Usando a heurística na otimização da consulta

Nesta seção, discutimos técnicas de otimização que aplicam regras heurísticas para modificar a representação interna de uma consulta — que normalmente está na forma de uma árvore de consulta ou uma estrutura de dados de grafo de consulta — para melhorar seu desempenho esperado. A varredura e o analisador de uma consulta SQL gera primeiro uma estrutura de dados que corresponde a uma *representação inicial de consulta*, que é então otimizada de

acordo com regras heurísticas. Isso leva a uma *representação de consulta otimizada*, que corresponde à estratégia de execução da consulta. Depois disso, um plano de execução de consulta é gerado para executar grupos de operações com base nos caminhos de acesso disponíveis nos arquivos envolvidos na consulta.

Uma das principais regras heurísticas é aplicar operações SELEÇÃO e PROJEÇÃO *antes* de aplicar a JUNÇÃO ou outras operações binárias, pois o tamanho do arquivo resultante de uma operação binária — como JUNÇÃO — normalmente é uma função multiplicativa dos tamanhos dos arquivos de entrada. As operações SELEÇÃO e PROJEÇÃO reduzem o tamanho de um arquivo e, portanto, devem ser aplicadas *antes* de uma junção ou outra operação binária.

Na Seção 19.7.1, reiteramos as notações de árvore de consulta e grafo de consulta já introduzidas no contexto da álgebra e cálculo relacional, nas seções 6.3.5 e 6.6.5, respectivamente. Estas podem ser usadas como base para as estruturas de dados que servem para a representação interna das consultas. Uma *árvore de consulta* é utilizada pra representar uma expressão da álgebra relacional ou da álgebra relacional estendida, enquanto um *grafo de consulta* o é para representar uma expressão do cálculo relacional. Depois, na Seção 19.7.2, mostramos como as regras de otimização heurísticas são aplicadas para converter uma árvore de consulta inicial em uma árvore de consulta equivalente, que representa uma expressão da álgebra relacional diferente, a qual é mais eficiente de ser executada, mas gera o mesmo resultado da árvore original. Também discutimos a equivalência de diversas expressões da álgebra relacional. Finalmente, a Seção 19.7.3 discute a geração de planos de execução de consulta.

### 19.7.1 Notação para árvores de consulta e grafos de consulta

Uma árvore de consulta é uma estrutura de dados de árvore que corresponde a uma expressão da álgebra relacional. Ela representa as relações de entrada da consulta como *nós folha* da árvore, e representa as operações da álgebra relacional como *nós internos*. Uma execução da árvore de consulta consiste na execução de uma operação de nó interno sempre que seus operandos estão disponíveis e depois na substituição desse nó interno pela relação que resulta da execução da operação. A ordem de execução das operações *começa nos nós folha*, que representa as relações do banco de dados de entrada para a consulta, e *termina no nó raiz*, que representa a operação final da consulta. A execução termina quando a operação do nó raiz é executada e produz a relação de resultado para a consulta.

A Figura 19.4(a) mostra uma árvore de consulta (a mesma mostrada na Figura 6.9) para a consulta C2 dos capítulos 4 a 6: para cada projeto localizado em ‘Mauá’, recupere o número do projeto, o número do departamento de controle, o sobrenome, o endereço e a data de nascimento do gerente do departamento. Essa consulta é especificada no esquema relacional EMPRESA da Figura 3.5 e corresponde à seguinte expressão da álgebra relacional:

$$\pi_{\text{Projnumero}, \text{Dnum}, \text{Unome}, \text{Endereco}, \text{Data_nasc}}((\sigma_{\text{Projlocal} = \text{'Mauá'}}(\text{PROJETO})) \bowtie_{\text{Dnum}=\text{Dnumero}} (\text{DEPARTAMENTO}) \bowtie_{\text{Cpf\_ger}=\text{Cpf}} (\text{FUNCIONARIO}))$$

Isso corresponde à seguinte consulta SQL:

**C2: SELECT** P.Projnumero, P.Dnum, F.Unome, F.Endereco, F.Data\_nasc

**FROM** PROJETO AS P, DEPARTAMENTO AS D, FUNCIONARIO AS F  
**WHERE** P.Dnum=D.Dnumero AND D.Cpf\_ger=F.Cpf AND P.Projlocal=‘Mauá’;

Na Figura 19.4(a), os nós folha P, D e F representam as três relações PROJETO, DEPARTAMENTO e FUNCIONARIO, respectivamente, e os nós da árvore interna representam as *operações da álgebra relacional* da expressão. Quando essa árvore de consulta é executada, o nó marcado com (1) na Figura 19.4(a) deve iniciar a execução antes do nó (2), porque algumas tuplas resultantes da operação (1) devem estar disponíveis antes de podermos iniciar a execução da operação (2). De modo semelhante, o nó (2) deve começar a executar e produzir resultados antes que o nó (3) possa iniciar a execução, e assim por diante.

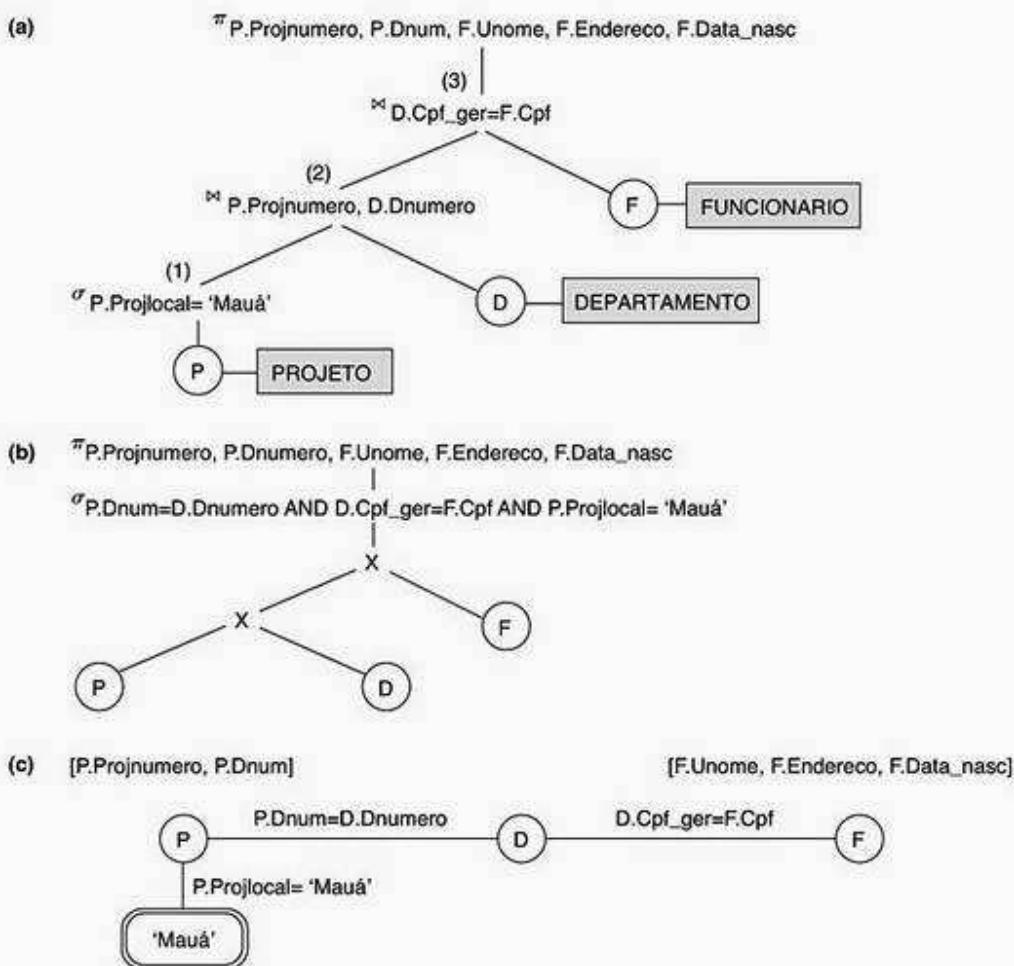


Figura 19.4

Duas árvores de consulta para a consulta C2. (a) Árvore de consulta correspondente à expressão da álgebra relacional para C2. (b) Árvore de consulta inicial (canônica) para a consulta SQL C2. (c) Grafo de consulta para C2.

Como podemos ver, a árvore de consulta representa uma ordem de operações específica para a execução de uma consulta. Uma estrutura de dados mais neutra para a representação de uma consulta é a notação do grafo de consulta. A Figura 19.4(c) (a mesma mostrada na Figura 6.13) traz o grafo de consulta para a consulta C2. As relações na consulta são representadas por nós de relação, que são exibidos como círculos isolados. Os valores de constantes, normalmente das condições de seleção de consulta, são representados por nós de constante, que são exibidos como círculos duplos ou ovais. As condições de seleção e junção são representadas pelas arestas do grafo, como mostra a Figura 19.4(c). Por fim, os atributos a serem recuperados de cada relação são exibidos entre colchetes, acima dela.

A representação do grafo de consulta não indica uma ordem sobre quais operações realizar primeiro. Existe apenas um único grafo correspondente a cada consulta.<sup>15</sup> Embora algumas técnicas de otimização fossem baseadas em grafos de consulta, agora costuma-se aceitar que as árvores de consulta são preferíveis porque, na prática, o otimizador de consulta precisa mostrar a ordem das operações para a execução da consulta, o que não é possível nos grafos de consulta.

### 19.7.2 Otimização heurística das árvores de consulta

Em geral, muitas expressões diferentes da álgebra relacional — e, portanto, muitas árvores de consulta diferentes — podem ser equivalentes; ou seja, elas podem representar a *mesma consulta*.<sup>16</sup>

O analisador de consulta normalmente gerará uma árvore de consulta inicial padrão para corresponder a uma consulta SQL, sem realizar qualquer otimização. Por exemplo, para uma consulta SELEÇÃO-PROJEÇÃO-JUNÇÃO, como C2, a árvore inicial aparece na Figura 19.4(b). O PRODUTO CARTESIANO das relações especificadas na cláusula FROM é aplicado primeiro; depois, as condições de seleção e junção da cláusula WHERE são aplicadas, seguidas pela projeção nos atributos da cláusula SELEÇÃO. Essa árvore de consulta canônica representa uma expressão da álgebra relacional que é *muito ineficaz se executada diretamente*, por causa das operações do PRODUTO CARTESIANO (X). Por exemplo, se as relações PROJETO, DEPARTAMENTO e FUNCIONARIO tivessem tamanhos de registro de 100, 50 e 150 bytes, e tivessem 100, 20 e 5.000 tuplas, respectivamente, o resultado do PRODUTO CARTESIANO conteria 10

milhões de tuplas com tamanho de registro de 300 bytes cada. Porém, a árvore de consulta inicial na Figura 19.4(b) está em uma forma padrão simples, que pode ser facilmente criada com base na consulta SQL. Ela nunca será executada. O otimizador de consulta heurística transformará essa árvore de consulta inicial em uma árvore de consulta final equivalente, que é eficiente para se executar.

O otimizador deve incluir regras para *equivalência entre expressões da álgebra relacional* que podem ser aplicadas para transformar a árvore inicial na árvore de consulta otimizada final. Primeiro, discutimos informalmente como uma árvore de consulta é transformada pelo uso de heurísticas, e depois discutimos as regras de transformação gerais e mostramos como elas podem ser usadas em um otimizador heurístico algébrico.

**Exemplo de transformação de uma consulta.** Considere a seguinte consulta C no banco de dados da Figura 3.5: *ache os sobrenomes dos funcionários nascidos após 1957 que trabalham em um projeto chamado 'Aquarius'*. Essa consulta pode ser especificada em SQL da seguinte forma:

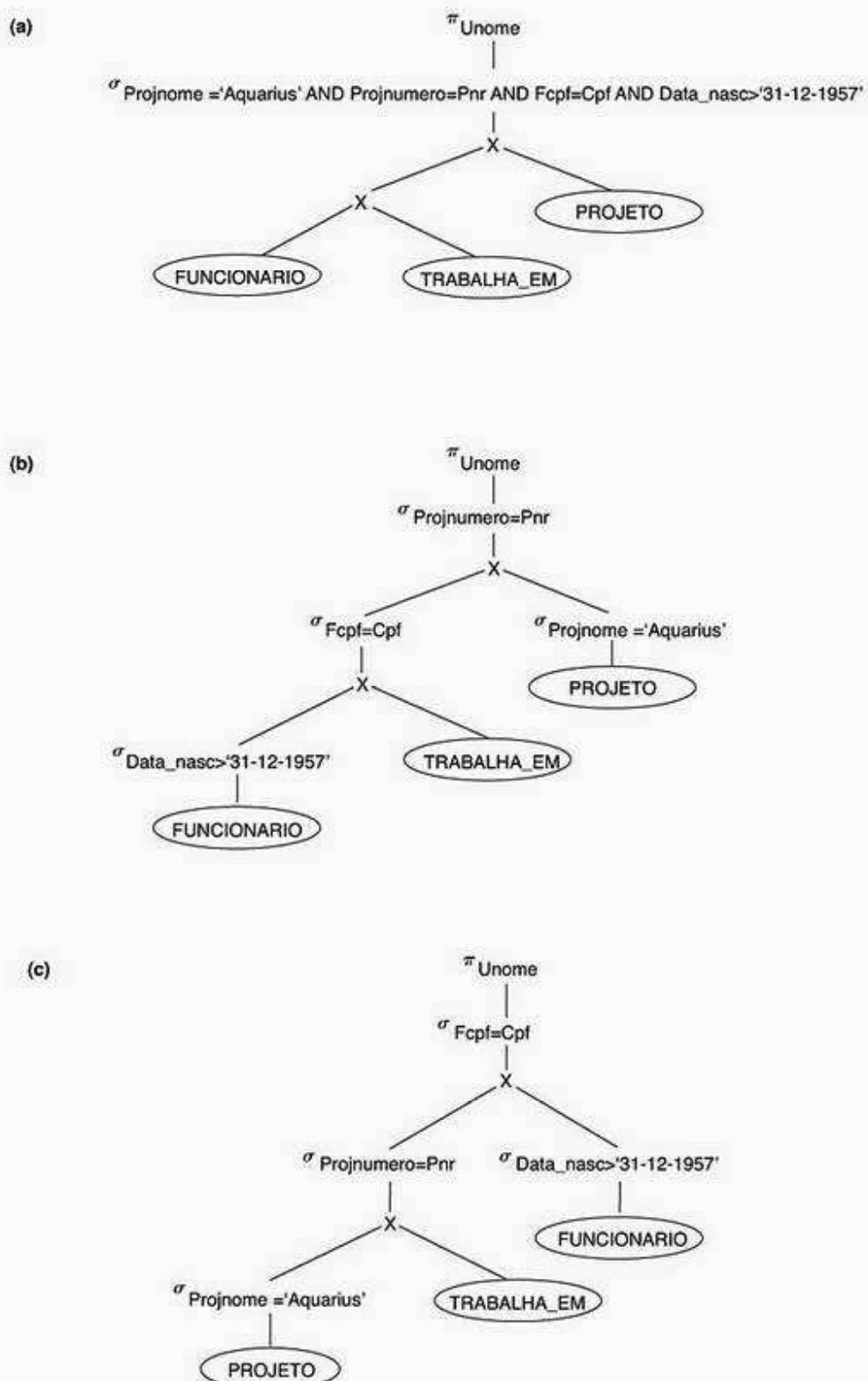
```
C: SELECT Unome
  FROM FUNCIONARIO, TRABALHA_EM,
        PROJETO
 WHERE Projnome='Aquarius' AND
       Projnumero=Pnr AND FCpf=Cpf AND
       Data_nasc > '31-12-1957';
```

A árvore de consulta inicial para C aparece na Figura 19.5(a). A execução dessa árvore primeiro cria um arquivo muito grande contendo o PRODUTO CARTESIANO dos arquivos FUNCIONARIO, TRABALHA\_EM e PROJETO inteiros. É por isso que a árvore de consulta inicial nunca é executada, mas sim, transformada em outra árvore equivalente, que é eficiente para se executar. Essa consulta em particular só precisa de um registro da relação PROJETO — para o projeto 'Aquarius' — e apenas os registros de FUNCIONARIO para aqueles cuja data de nascimento se dá após '31-12-1957'. A Figura 19.5(b) mostra uma árvore de consulta melhorada, que primeiro aplica as operações SELEÇÃO para reduzir o número de tuplas que aparecem no PRODUTO CARTESIANO.

Outra melhoria é alcançada trocando as posições das relações FUNCIONARIO e PROJETO na árvore, como mostra a Figura 19.5(c). Isso usa a informação de que Projnumero é um atributo de chave da relação PROJETO, e, portanto, a operação SELEÇÃO

<sup>15</sup> Portanto, um grafo de consulta corresponde a uma expressão do cálculo relacional, como mostramos na Seção 6.6.5.

<sup>16</sup> A mesma consulta também pode ser declarada de várias maneiras em uma linguagem de consulta de alto nível, como a SQL (ver capítulos 4 e 5).



(continua)

**Figura 19.5**

Etapas na conversão de uma árvore de consulta durante a otimização heurística. (a) Árvore de consulta inicial (canônica) para a consulta SQL C. (b) Movendo as operações SELEÇÃO mais para baixo na árvore de consulta. (c) Aplicando a operação SELEÇÃO mais restritiva primeiro.

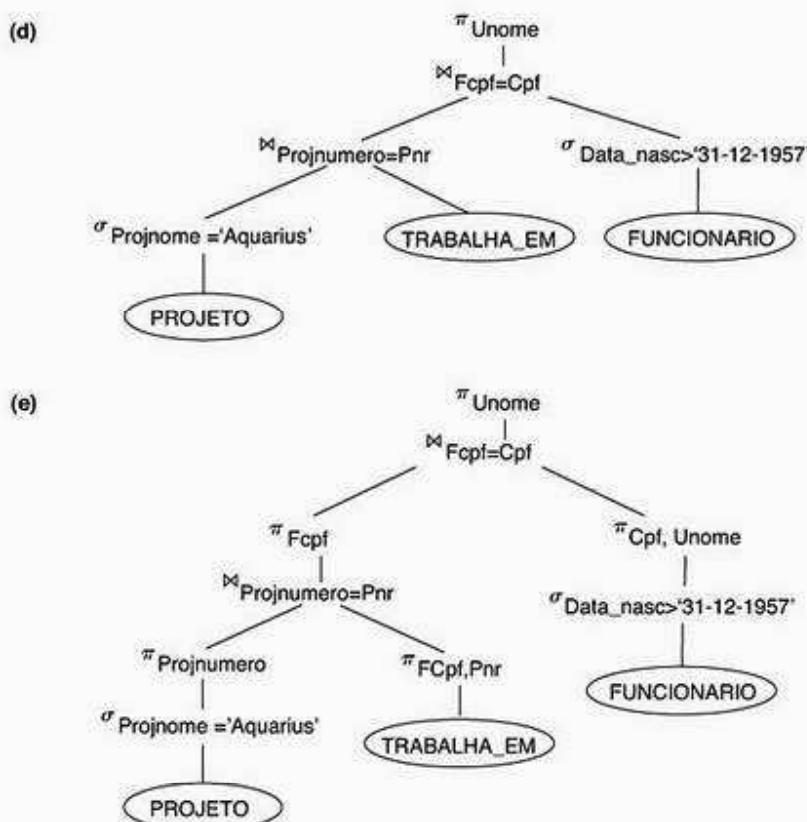


Figura 19.5 (continuação)

Etapas na conversão de uma árvore de consulta durante a otimização heurística. (d) Substituindo PRODUTO CARTESIANO e SELEÇÃO por operações JUNÇÃO. (e) Movendo operações PROJEÇÃO mais para baixo na árvore de consulta.

na relação PROJETO recuperará um único registro. Podemos melhorar ainda mais a árvore de consulta ao substituir qualquer operação de PRODUTO CARTESIANO que seja acompanhada por uma condição de junção por uma operação JUNÇÃO, como mostra a Figura 19.5(d). Outra melhoria é manter apenas os atributos necessários pelas operações subsequentes nas relações intermediárias, incluindo operações PROJEÇÃO ( $\pi$ ) o mais cedo possível na árvore de consulta, como mostra a Figura 19.5(e). Isso reduz os atributos (colunas) das relações intermediárias, enquanto as operações SELEÇÃO reduzem o número de tuplas (registros).

Conforme mostra o exemplo anterior, uma árvore de consulta pode ser transformada passo a passo em uma árvore de consulta equivalente que é mais eficiente de se executar. Porém, temos de garantir que as etapas de transformação sempre levem a uma árvore de consulta equivalente. Para fazer isso, o otimizador de consulta precisa saber quais regras de transformação *preservam essa equivalência*. Discutiremos algumas dessas regras de transformação a seguir.

**Regras de transformação gerais para operações da álgebra relacional.** Existem muitas regras para transformar operações da álgebra relacional em equivalentes. Para fins de otimização de consulta, estamos interessados no significado das operações e das relações resultantes. Logo, se duas relações tiverem o mesmo conjunto de atributos em uma *ordem diferente*, mas ambas representarem a mesma informação, consideramos que as relações são equivalentes. Na Seção 3.1.2, demos uma definição alternativa da *relação* que torna a ordem dos atributos não importante; usaremos essa definição aqui. Vamos expressar algumas regras de transformação que são úteis na otimização da consulta, sem prová-las:

1. **Cascata de  $\sigma$ .** Uma condição de seleção conjuntiva pode ser desmembrada em uma cascata (ou seja, uma sequência) de operações  $\sigma$  individuais:  

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$
2. **Comutatividade de  $\sigma$ .** A operação  $\sigma$  é comutativa:  

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

3. **Cascata de  $\pi$ .** Em uma cascata (sequência) de operações  $\pi$ , todas podem ser ignoradas, menos a última:

$$\pi_{List_1}(\pi_{List_2}(\dots(\pi_{List_n}(R))\dots)) \equiv \pi_{List_1}(R)$$

4. **Comutação de  $\sigma$  com  $\pi$ .** Se a condição de seleção  $c$  envolve apenas os atributos  $A_1, \dots, A_n$  na lista de projeção, as duas operações podem ser comutadas:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

5. **Comutatividade de  $\bowtie$  (e  $\times$ ).** A operação de junção é comutativa, assim como a operação  $\times$ :

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Observe que, embora a ordem dos atributos possa não ser a mesma nas relações resultantes das duas junções (ou dois produtos cartesianos), o significado é o mesmo porque a ordem dos atributos não é importante na definição alternativa da relação.

6. **Comutação de  $\sigma$  com  $\bowtie$  (ou  $\times$ ).** Se todos os atributos na condição de seleção  $c$  envolvem apenas os atributos de uma das relações sendo juntadas — digamos,  $R$  —, as duas operações podem ser comutadas da seguinte forma:

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

Como alternativa, se a condição de seleção  $c$  puder ser escrita como  $(c_1 \text{ AND } c_2)$ , onde a condição  $c_1$  envolve apenas os atributos de  $R$  e a condição  $c_2$  envolve apenas os atributos de  $S$ , as operações são comutadas da seguinte forma:

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$$

As mesmas regras se aplicam se o  $\bowtie$  for substituído por uma operação  $\times$ .

7. **Comutação de  $\pi$  com  $\bowtie$  (ou  $\times$ ).** Suponha que a lista de projeção seja  $L = [A_1, \dots, A_n, B_1, \dots, B_m]$ , onde  $A_1, \dots, A_n$  são os atributos de  $R$  e  $B_1, \dots, B_m$  são atributos de  $S$ . Se a condição de junção  $c$  envolver apenas atributos em  $L$ , as duas operações podem ser comutadas da seguinte forma:

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \dots, A_n}(R)) \bowtie_c (\pi_{B_1, \dots, B_m}(S))$$

Se a condição de junção  $c$  contiver atributos adicionais não em  $L$ , estes devem ser acrescentados à lista de projeção, e uma operação  $\pi$  final é necessária. Por exemplo, se os atributos  $A_{n+1}, \dots, A_{n+k}$  de  $R$  e  $B_{m+1}, \dots, B_{m+p}$  de  $S$  estiverem envolvidos na condição de junção  $c$ , mas não estiverem na lista de projeção  $L$ , as operações são comutadas da seguinte forma:

$$\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A_1, \dots, A_n, A_{n+1}, \dots, A_{n+k}}(R)) \bowtie_c (\pi_{B_1, \dots, B_m, B_{m+1}, \dots, B_{m+p}}(S)))$$

Para  $\times$ , não existe condição  $c$ , de modo que a primeira regra de transformação sempre se aplica substituindo  $\bowtie_c$  por  $\times$ .

8. **Comutatividade das operações de conjunto.** As operações de conjunto  $\cup$  e  $\cap$  são comutativas, mas — não é.

9. **Associatividade de  $\bowtie$ ,  $\times$ ,  $\cup$  e  $\cap$ .** Essas quatro operações são associativas individualmente; ou seja, se  $\theta$  indicar qualquer uma dessas quatro operações (por toda a expressão), temos:  $(R \theta S) \theta T \equiv R \theta (S \theta T)$

10. **Comutação de  $\sigma$  com operações de conjunto.** A operação  $\sigma$  comuta com  $\cup$ ,  $\cap$  e  $-$ . Se  $\theta$  indicar qualquer uma dessas três operações (por toda a expressão), temos:

$$\sigma_c(R \theta S) \equiv (\sigma_c(R)) \theta (\sigma_c(S))$$

11. **A operação  $\pi$  comuta com  $\cup$ .**

$$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$$

12. **Convertendo uma sequência  $(\sigma, \times)$  em  $\bowtie$ .** Se a condição  $c$  de um  $\sigma$  que segue um  $\times$  corresponde a uma condição de junção, converta a sequência  $(\sigma, \times)$  em um  $\bowtie$ , da seguinte forma:

$$(\sigma_c(R \times S)) \equiv (R \bowtie_c S)$$

Existem outras transformações possíveis. Por exemplo, uma condição de seleção ou junção  $c$  pode ser convertida para uma condição equivalente usando as seguintes regras padrão da álgebra booleana (leis de DeMorgan):

$$\text{NOT}(c_1 \text{ AND } c_2) \equiv (\text{NOT } c_1) \text{ OR } (\text{NOT } c_2)$$

$$\text{NOT}(c_1 \text{ OR } c_2) \equiv (\text{NOT } c_1) \text{ AND } (\text{NOT } c_2)$$

Transformações adicionais discutidas nos capítulos 4, 5 e 6 não são repetidas aqui. Discutimos em seguida como as transformações podem ser usadas na otimização heurística.

**Esboço de um algoritmo de otimização algébrica heurística.** Agora, podemos esboçar as etapas de um algoritmo que utiliza algumas das regras acima para transformar uma árvore de consulta inicial em uma árvore final que seja mais eficiente de executar (na maioria dos casos). O algoritmo levará a transformações semelhantes às aquelas discutidas em nosso exemplo da Figura 19.5. As etapas do algoritmo são as seguintes:

1. Usando a Regra 1, quebre quaisquer operações SELEÇÃO com condições conjuntivas em uma cascata de operações SELEÇÃO. Isso permite um maior grau de liberdade na movi-

- mentação para baixo de operações SELEÇÃO por diferentes ramos da árvore.
2. Usando as regras 2, 4, 6 e 10 referentes à comutatividade de SELEÇÃO com outras operações, move cada operação SELEÇÃO o mais para baixo possível na árvore de consulta que for permitido pelos atributos envolvidos na condição de seleção. Se a condição envolver atributos de *apenas uma tabela*, o que significa que ela representa uma *condição de seleção*, a operação é movida até o nó folha que representa essa tabela. Se a condição envolver atributos de *duas tabelas*, o que significa que ela representa uma *condição de junção*, a condição é movida para um local mais abaixo na árvore, após as duas tabelas serem combinadas.
  3. Usando as regras 5 e 9 referentes à comutatividade e associatividade de operações binárias, reorganize os nós folha da árvore usando os critérios a seguir. Primeiro, posicione as relações do nó folha com as operações SELEÇÃO mais restritivas, de modo que sejam executadas primeiro na representação da árvore de consulta. A definição da SELEÇÃO *mais restritiva* pode significar aquelas que produzem uma relação com menos tuplas ou com o menor tamanho absoluto.<sup>17</sup> Outra possibilidade é definir a SELEÇÃO mais restritiva como aquela com a menor seletividade; isso é mais prático, pois as estimativas de seletividades normalmente estão disponíveis no catálogo do SGBD. Em segundo lugar, garanta que a ordenação dos nós folha não cause operações de PRODUTO CARTESIANO; por exemplo, se as duas relações com a SELEÇÃO mais restritiva não tiverem uma condição de junção direta entre elas, pode ser desejável mudar a ordem dos nós folha para evitar produtos cartesianos.<sup>18</sup>
  4. Usando a Regra 12, combine uma operação PRODUTO CARTESIANO com uma operação SELEÇÃO subsequente na árvore para uma operação JUNÇÃO, se a condição representar uma condição de junção.
  5. Usando as regras 3, 4, 7 e 11 referentes à cascata de PROJEÇÃO e a comutação de PROJEÇÃO com outras operações, desmembre e move as listas de atributos de projeção para baixo na árvore ao máximo possível, criando novas operações PROJEÇÃO, conforme a necessidade. Somente os atributos necessários no resultado da consulta e nas operações subsequentes na árvore de consulta devem ser mantidas após cada operação PROJEÇÃO.
  6. Identifique subárvores que representam grupos de operações que podem ser executados por um único algoritmo.

Em nosso exemplo, a Figura 19.5(b) mostra a árvore da Figura 19.5(a) após aplicar as etapas 1 e 2 do algoritmo; a Figura 19.5(c) mostra a árvore após a etapa 3; a Figura 19.5(d), após a etapa 4; e a Figura 19.5(e), após a etapa 5. Na etapa 6, podemos agrupar as operações na subárvore cuja raiz é a operação  $\pi_{FC_{Opf}}$  em um único algoritmo. Também podemos agrupar as operações restantes em outra subárvore, na qual as tuplas resultantes do primeiro algoritmo substituem a subárvore cuja raiz é a operação  $\pi_{FC_{Opf}}$ , pois o primeiro agrupamento significa que essa subárvore é executada primeiro.

**Resumo da heurística para otimização algébrica.** A heurística principal é aplicar primeiro as operações que reduzem o tamanho dos resultados intermediários. Isso inclui a realização o mais cedo possível de operações SELEÇÃO para reduzir o número de tuplas e operações PROJEÇÃO para reduzir o número de atributos — ao mover as operações SELEÇÃO e PROJEÇÃO o mais para baixo na árvore possível. Além disso, as operações SELEÇÃO e JUNÇÃO que são mais restritivas — ou seja, que resultam em relações com menos tuplas ou com o menor tamanho absoluto — devem ser executadas antes de outras operações semelhantes. A última regra é realizada por meio da reordenação dos nós folha da árvore entre eles mesmos, enquanto evita produtos cartesianos, e do ajuste do restante da árvore adequadamente.

### 19.7.3 Convertendo árvores de consulta em planos de execução de consulta

Um plano de execução para uma expressão da álgebra relacional representada como uma árvore de consulta inclui informações sobre os métodos de acesso disponíveis para cada relação, bem como os algoritmos a serem usados na computação dos operadores relacionais representados na árvore. Como um exemplo simples, considere a consulta C1 do Capítulo 4, cuja expressão da álgebra relacional correspondente é

$$\begin{aligned} &\pi_{\text{Nome}, \text{Unome}, \text{Endereco}}(\sigma_{\text{Dname} = \text{'Pesquisa'}}(\text{DEPARTAMENTO}) \\ &\bowtie_{\text{Dnumero} = \text{Dnr}} \text{FUNCIONARIO}) \end{aligned}$$

<sup>17</sup> Qualquer definição pode ser usada, pois essas regras são heurísticas.

<sup>18</sup> Observe que um PRODUTO CARTESIANO é aceitável em alguns casos — por exemplo, se cada relação só tiver uma única tupla, pois cada uma teve uma condição de seleção anterior em um campo chave.

A árvore de consulta é mostrada na Figura 19.6. Para converter isso em um plano de execução, o otimizador poderia escolher uma busca de índice para a operação SELEÇÃO em DEPARTAMENTO (supondo que exista uma), um algoritmo de junção de único loop que percorra os registros no resultado da operação SELEÇÃO em DEPARTAMENTO para a operação de junção (supondo que exista um índice no atributo Dnr de FUNCIONARIO), e uma varredura do resultado de JUNÇÃO para a entrada do operador PROJEÇÃO. Além disso, a técnica para executar a consulta pode especificar uma avaliação materializada ou canalizada, embora em geral uma avaliação canalizada seja preferida sempre que viável.

Com a avaliação materializada, o resultado de uma operação é armazenado como uma relação temporária (ou seja, o resultado é *fisicamente materializado*). Por exemplo, a operação JUNÇÃO pode ser calculada e o resultado inteiro, armazenado como uma relação temporária, que é então lida como entrada pelo algoritmo que calcula a operação PROJEÇÃO, que produziria a tabela de resultado da consulta. Por sua vez, com a avaliação em pipeline, à medida que as tuplas resultantes de uma operação são produzidas, elas são encaminhadas diretamente à próxima operação na sequência de consulta. Por exemplo, à medida que as tuplas selecionadas de DEPARTAMENTO são produzidas pela operação SELEÇÃO, elas são colocadas em um buffer; o algoritmo da operação JUNÇÃO consumiria então as tuplas do buffer, e as tuplas que resultam da operação JUNÇÃO são pipeline para o algoritmo da operação de projeção. A vantagem da pipeline é a economia de custo por não ter de gravar os resultados imediatos em disco e não ter de lê-los de volta para a próxima operação.



**Figura 19.6**  
Uma árvore de consulta para a consulta C1.

## 19.8 Usando seletividade e estimativas de custo na otimização da consulta

Um otimizador de consulta não depende unicamente de regras heurísticas; ele também estima e compara os custos da execução de uma consulta utilizando diferentes estratégias de execução e algoritmos, e depois escolhe a estratégia com a *estimativa de custo mais baixa*. Para que essa técnica funcione, *estimativas de custo* precisas são exigidas, de modo que diferentes estratégias possam ser comparadas justa e realisticamente. Além disso, o otimizador precisa limitar o número de estratégias de execução a serem consideradas; caso contrário, muito tempo será gasto ao fazer estimativas de custo para as muitas estratégias de execução possíveis. Logo, essa técnica é mais adequada para consultas compiladas, nas quais a otimização é feita na hora da compilação e o código da estratégia de execução resultante é armazenado e executado diretamente em tempo de execução. Para consultas interpretadas, nas quais o processo inteiro mostrado na Figura 19.1 ocorre em tempo de execução, uma otimização em escala total pode atrasar o tempo de resposta. Uma otimização mais elaborada é indicada para consultas compiladas, enquanto uma otimização parcial, menos demorada, funciona melhor para consultas interpretadas.

Essa técnica geralmente é denominada **otimização de consulta baseada em custo**.<sup>19</sup> Ela usa técnicas de otimização tradicionais que pesquisam o *espaço da solução* para um problema em busca de uma solução que minimize uma função de objetivo (custo). As funções de custo usadas na otimização de consulta são estimativas e não funções de custo exatas, de modo que a otimização pode selecionar uma estratégia de execução de consulta que não é a ideal (melhor absoluta). Na Seção 19.8.1, discutimos os componentes do custo de execução da consulta. Na Seção 19.8.2, discutimos o tipo de informação necessária em funções de custo. Essa informação é mantida no catálogo do SGBD. Na Seção 19.8.3, damos exemplos de funções de custo para a operação SELEÇÃO, e, na Seção 19.8.4, discutimos as funções de custo para operações JUNÇÃO de duas vias. A Seção 19.8.5 aborda as junções em múltiplas vias, e a Seção 19.8.6 fornece um exemplo.

<sup>19</sup> Essa técnica foi usada inicialmente no otimizador para o SYSTEM R em um SGBD experimental desenvolvido na IBM (Selinger et al., 1979).

### 19.8.1 Componentes de custo para execução da consulta

O custo da execução de uma consulta inclui os seguintes componentes:

- Custo de acesso ao armazenamento secundário.** Esse é o custo de transferir (ler e gravar) blocos de dados entre o armazenamento de disco secundário e os buffers da memória principal. Isso também é conhecido como *custo de E/S (entrada/saída) de disco*. O custo de procurar registros em um arquivo de disco depende do tipo de estruturas de acesso nesse arquivo, como ordenação, hashing e índices primário ou secundário. Além disso, fatores como se os blocos de arquivo estão alocados consecutivamente no mesmo cilindro de disco ou espalhados pelo disco afetam o custo de acesso.
- Custo de armazenamento em disco.** Esse é o custo de armazenar no disco quaisquer arquivos intermediários que sejam gerados por uma estratégia de execução para a consulta.
- Custo de computação.** Esse é o custo de realizar operações na memória em registros dentro dos buffers de dados durante a execução da consulta. Essas operações incluem procurar e ordenar registros, mesclar registros para uma operação de junção ou ordenação, e realizar cálculos em valores de campo. Isso também é conhecido como *custo de CPU (unidade central de processamento)*.
- Custo de uso da memória.** Esse é o custo que diz respeito ao número de buffers da memória principal necessários durante a execução da consulta.
- Custo de comunicação.** Esse é o custo de envio da consulta e seus resultados do local do banco de dados até o local ou terminal onde a consulta foi originada. Nos bancos de dados distribuídos (ver Capítulo 25), isso também incluiria o custo de transferência de tabelas e resultados entre vários computadores durante a avaliação da consulta.

Para bancos de dados grandes, a ênfase principal costuma estar na minimização do custo de acesso para armazenamento secundário. Funções de custo simples ignoram outros fatores e compararam diferentes estratégias de execução de consulta em relação ao número de transferências de bloco entre buffers do disco e da memória principal. Para bancos de dados menores, nos quais a maioria dos dados nos arquivos envolvidos

na consulta pode ser armazenada completamente na memória, a ênfase é na minimização do custo de computação. Em bancos de dados distribuídos, nos quais muitos locais estão envolvidos (ver Capítulo 25), o custo da comunicação também precisa ser minimizado. É difícil incluir todos os componentes de custo em uma função de custo (ponderada), devido à dificuldade de atribuir pesos adequados aos componentes de custo. É por isso que algumas funções de custo consideram apenas um único fator — acesso de disco. Na próxima seção, vamos tratar de algumas informações necessárias para formular funções de custo.

### 19.8.2 Informação de catálogo usada nas funções de custo

Para estimar os custos de diversas estratégias de execução, temos de registrar qualquer informação necessária para as funções de custo. Essa informação pode ser armazenada no catálogo do SGBD, onde é acessada pelo otimizador de consulta. Primeiro, temos de saber o tamanho de cada arquivo. Para um arquivo cujos registros são do mesmo tipo, o número de registros (tuplas) ( $r$ ), o tamanho do registro (médio) ( $R$ ) e o número de blocos de arquivo ( $b$ ) (ou estimativas próximas deles) são necessários. O fator de bloco ( $b_{fr}$ ) para o arquivo também pode ser necessário. Também devemos registrar a organização de arquivo primária para cada arquivo. Os registros da organização de arquivo primária podem ser *desordenados*, *ordenados* por um atributo com ou sem um índice primário ou de agrupamento, ou *hashed* (hashing estático ou um dos métodos de hashing dinâmico) em um atributo chave. A informação também é mantida em todos os índices primários, secundários ou de agrupamento e seus atributos de indexação. O número de níveis ( $x$ ) de cada índice multi-nível (primário, secundário ou de agrupamento) é necessário para funções de custo que estimam o número de acessos de bloco que ocorrem durante a execução da consulta. Em algumas funções de custo, o número de blocos de índice de primeiro nível ( $b_{n1}$ ) é necessário.

Outro parâmetro importante é o número de valores distintos ( $d$ ) de um atributo e sua seletividade ( $sl$ ), que é a fração de registros que satisfazem uma condição de igualdade no atributo. Isso permite a estimativa da cardinalidade de seleção ( $s = sl * r$ ) de um atributo, que é o número médio de registros que satisfarão uma condição de seleção de igualdade nesse atributo. Para um *atributo chave*,  $d = r$ ,  $sl = 1/r$  e  $s = 1$ . Para um *atributo não chave*, ao fazer a suposição de que os  $d$  valores distintos são distribuídos uniformemente entre os registros, estimamos  $sl = (1/d)$  e, portanto,  $s = (r/d)$ .<sup>20</sup>

<sup>20</sup> Otimizadores mais precisos armazenam *histogramas* da distribuição dos registros nos valores de dados para um atributo.

Informações como o número de níveis de índice são fáceis de se manter porque não mudam com muita frequência. Porém, outras informações podem mudar frequentemente; por exemplo, o número de registros  $r$  em um arquivo muda toda vez que um registro é inserido ou excluído. O otimizador de consulta precisará de valores bem próximos, mas não necessariamente atualizados até o último minuto desses parâmetros para uso na estimativa do custo de diversas estratégias de execução.

Para um atributo não chave com  $d$  valores distintos, é comum acontecer de os registros não serem distribuídos uniformemente entre esses valores. Por exemplo, suponha que uma empresa tenha cinco departamentos numerados de 1 a 5, e 200 funcionários que estão distribuídos entre os departamentos da seguinte forma: (1, 5), (2, 25), (3, 70), (4, 40), (5, 60). Nesses casos, o otimizador pode armazenar um *histograma* que reflete a distribuição dos registros de funcionários em diferentes departamentos em uma tabela com os dois atributos (Dnr, Seletividade), que teriam os seguintes valores para nosso exemplo: (1, 0,025), (2, 0,125), (3, 0,35), (4, 0,2), (5, 0,3). Os valores de seletividade armazenados no histograma também podem ser estimativas se a tabela de funcionários mudar com frequência.

Nas próximas duas seções, examinamos como alguns desses parâmetros são usados em funções de custo para um otimizador de consulta baseado em custo.

### 19.8.3 Exemplos de funções de custo para SELEÇÃO

Agora, damos as funções de custo para os algoritmos de seleção S1 a S8 discutidos na Seção 19.3.1 em relação ao *número de transferências de bloco* entre a memória e o disco. O algoritmo S9 envolve uma interseção de ponteiros de registro após eles terem sido recuperados por algum outro meio, como o algoritmo S6, e, portanto, a função de custo será baseada no custo para S6. Essas funções de custo são estimativas que ignoraram o tempo de computação, o custo de armazenamento e outros fatores. O custo para o método Si é indicado como  $C_{S_i}$  acessos de bloco.

- **S1 — Técnica de pesquisa linear (força bruta).** Pesquisamos todos os blocos do arquivo para recuperar todos os registros que satisfazem a condição de seleção; logo,  $C_{S1} = b$ . Para uma *condição de igualdade em um atributo chave*, somente metade dos blocos de arquivo são pesquisados *na média* antes de encontrar o registro, de modo que uma estimativa aproximada para  $C_{S1b} = (b/2)$  se o registro for encontrado; se nenhum registro for encontrado satisfazendo a condição,  $C_{S1b} = b$ .

- **S2 — Pesquisa binária.** Essa pesquisa acessa aproximadamente  $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$  blocos de arquivo. Isso reduz para  $\log_2 b$  se a condição de igualdade estiver em um atributo exclusivo (chave), pois  $s = 1$  nesse caso.
- **S3a — Usando um índice de chave primária para recuperar um único registro.** Para um índice primário, recupere um bloco de disco em cada nível de índice, mais um bloco de disco do arquivo de dados. Logo, o custo é um bloco de disco a mais que o número de níveis de índice:  $C_{S3a} = x + 1$ .
- **S3b — Usando uma chave hash para recuperar um único registro.** Para o hashing, somente um bloco de disco precisa ser acessado na maioria dos casos. A função de custo é aproximadamente  $C_{S3b} = 1$  para o hashing estático ou o hashing linear, e é dois acessos de bloco de disco para o hashing extensível (ver Seção 17.8).
- **S4 — Usando um índice de ordenação para recuperar múltiplos registros.** Se a condição de comparação for  $>$ ,  $\geq$ ,  $<$  ou  $\leq$  em um campo chave com um índice de ordenação, aproximadamente metade dos registros do arquivo satisfarão a condição. Isso produz uma função de custo de  $C_{S4} = x + (b/2)$ . Essa é uma estimativa muito aproximada e, embora possa estar correta na média, pode ser muito imprecisa em casos individuais. Uma estimativa mais precisa é possível se a distribuição de registros for armazenada em um histograma.
- **S5 — Usando um índice de agrupamento para recuperar múltiplos registros.** Um bloco de disco é acessado em cada nível de índice, que oferece o endereço do primeiro bloco de disco do arquivo no cluster. Dada uma condição de igualdade no atributo de indexação,  $s$  registros satisfarão a condição, onde  $s$  é a cardinalidade de seleção do atributo de indexação. Isso significa que  $\lceil (s/bfr) \rceil$  blocos de arquivo estarão no cluster de blocos de arquivo que mantêm todos os registros selecionados, gerando  $C_{S5} = x + \lceil (s/bfr) \rceil$ .
- **S6 — Usando um índice secundário ( $B^+$ -tree).** Para um índice secundário em um atributo chave (exclusivo), o custo é  $x + 1$  acessos de bloco de disco. Para um índice secundário em um atributo não chave (não exclusivo),  $s$  registros satisfarão uma condição de igualdade, onde  $s$  é a cardinalidade de seleção do atributo de indexação. Porém, como o índice é não de agrupamento, cada um dos registros pode residir em

um bloco de disco diferente, de modo que a estimativa de custo (pior caso) é  $C_{S6b} = x + 1 + s$ . O 1 adicional é levado em conta para o bloco de disco que contém os ponteiros de registro após o índice ser pesquisado (ver Figura 18.5). Se a condição de comparação for  $>$ ,  $\geq$ ,  $<$  ou  $\leq$  e metade dos registros de arquivo forem considerados para satisfazer a condição, então (muito aproximadamente) metade dos blocos de índice de primeiro nível são acessados, mais metade dos registros de arquivo por meio do índice. A estimativa de custo para este caso, de maneira aproximada, é  $C_{S6b} = x + (b_f/2) + (r_f/2)$ . O fator  $r_f/2$  pode ser refinado se existirem melhores estimativas de seletividade por meio de um histograma. O último método  $C_{S6b}$  pode ser muito dispendioso.

- **S7 — Seleção conjuntiva.** Podemos usar S1 ou um dos métodos de S2 a S6 já discutidos. Nesse último caso, usamos uma condição para recuperar os registros e depois verificamos nos buffers da memória principal se cada registro recuperado satisfaz as condições restantes na conjunção. Se existirem vários índices, a pesquisa de cada índice pode produzir um conjunto de ponteiros de registro (ids de registro) nos buffers da memória principal. A interseção dos conjuntos de ponteiros de registro (indicados em S9) pode ser calculada na memória principal e, então, os registros resultantes são lidos com base em seus ids de registro.
- **S8 — Seleção conjuntiva usando um índice composto.** O mesmo que S3a, S5 ou S6a, dependendo do tipo de índice.

**Exemplo de uso das funções de custo.** Em um otimizador de consulta, é comum enumerar as diversas estratégias possíveis para execução de uma consulta e estimar seus custos para diferentes estágios. Uma técnica de otimização, como a programação dinâmica, pode ser usada para encontrar a estimativa de custo ideal (menor) com eficiência, sem ter de considerar todas as estratégias de execução possíveis. Não discutimos os algoritmos de otimização aqui; em vez disso, usamos um exemplo simples para ilustrar como as estimativas de custo podem ser usadas. Suponha que o arquivo FUNCIONARIO da Figura 3.5 tenha  $r_f = 10.000$  registros armazenados em  $b_f = 2.000$  blocos de disco com fator de bloco  $bfr_f = 5$  registros/bloco e os seguintes caminhos de acesso:

1. Um índice de agrupamento em Salario, com níveis  $x_{Salario} = 3$  e cardinalidade de seleção média  $s_{Salario} = 20$ . (Isso corresponde a uma seletividade de  $sl_{Salario} = 0,002$ ).
2. Um índice secundário em um atributo chave Cpf, com  $x_{Cpf} = 4$  ( $sl_{Cpf} = 1$ ,  $s_{Cpf} = 0,0001$ ).
3. Um índice secundário no atributo não chave Dnr, com  $x_{Dnr} = 2$  e blocos de índice de primeiro nível  $b_{IDnr} = 4$ . Existem  $d_{Dnr} = 125$  valores distintos para Dnr, de modo que a seletividade de Dnr é  $sl_{Dnr} = (1/d_{Dnr}) = 0,008$ , e a cardinalidade de seleção é  $s_{Dnr} = (r_f * sl_{Dnr}) = (r_f / d_{Dnr}) = 80$ .
4. Um índice secundário em Sexo, com  $x_{Sexo} = 1$ . Existem  $d_{Sexo} = 2$  valores para o atributo Sexo, de modo que a cardinalidade de seleção média é  $s_{Sexo} = (r_f / d_{Sexo}) = 5.000$ . (Observe que, neste caso, um histograma que dá a porcentagem dos funcionários do sexo masculino e feminino pode ser útil, a menos que eles sejam aproximadamente iguais.)

Ilustramos o uso das funções de custo com os seguintes exemplos:

OP1:  $\sigma_{Cpf=12345678966}(\text{FUNCIONARIO})$

OP2:  $\sigma_{Dnr>5}(\text{FUNCIONARIO})$

OP3:  $\sigma_{Dnr=5}(\text{FUNCIONARIO})$

OP4:  $\sigma_{Dnr=5 \text{ AND } SALARIO > 30.000 \text{ AND } Sexo='F'}(\text{FUNCIONARIO})$

O custo da opção de força bruta (pesquisa linear ou varredura de arquivo) S1 será estimado como  $C_{S1a} = b_f = 2.000$  (para uma seleção em um atributo não chave) ou  $C_{S1b} = (b_f/2) = 1.000$  (custo médio para uma seleção em um atributo chave). Para OP1, podemos usar o método S1 ou o método S6a; a estimativa de custo para S6a é  $C_{S6a} = x_{Cpf} + 1 = 4 + 1 = 5$ , e ele é escolhido em relação ao método S1, cujo custo médio é  $C_{S1b} = 1000$ . Para OP2, podemos usar ou o método S1 (com custo estimado  $C_{S1a} = 2.000$ ) ou o método S6b (com custo estimado  $C_{S6b} = x_{Dnr} + (b_{IDnr}/2) + (r_f/2) = 2 + (4/2) + (10.000/2) = 5.004$ ), de modo que escolhemos a técnica de pesquisa linear para OP2. Para OP3, podemos utilizar o método S1 (com custo estimado  $C_{S1a} = 2.000$ ) ou o método S6a (com custo estimado  $C_{S6a} = x_{Dnr} + s_{Dnr} = 2 + 80 = 82$ ), de modo que escolhemos o método S6a.

Finalmente, considere OP4, que tem uma condição de seleção conjuntiva. Precisamos estimar o custo de usar qualquer um dos três componentes da condição de seleção para recuperar os registros, mais a técnica de pesquisa linear. A última gera a estimativa de custo  $C_{S1a} = 2.000$ . O uso da condição (Dnr = 5) primeiro gera a estimativa de custo  $C_{S6a} = 82$ . O uso da condição (Salario > 30.000) primeiro gera a estimativa de custo  $C_{S4} = x_{Salario} + (b_f/2) = 3 + (2.000/2) = 1.003$ . O uso da condição (Sexo = 'F') primeiro gera a estimativa de custo  $C_{S6a} = x_{Sexo} + s_{Sexo} = 1 + 5.000 = 5.001$ . O otimizador, então, escolheria o mé-

todo  $S_{6a}$  no índice secundário em  $Dnr$ , pois ele tem a menor estimativa de custo. A condição ( $Dnr = 5$ ) serve para recuperar os registros, e a parte restante da condição conjuntiva ( $Salario > 30.000$  AND  $Sexo = 'F'$ ) é verificada para cada registro selecionado depois que ele for recuperado para a memória. Apenas os registros que satisfazem essas condições adicionais são incluídos no resultado da operação.

#### 19.8.4 Exemplos de funções de custo para JUNÇÃO

Para desenvolver funções de custo razoavelmente precisas para operações JUNÇÃO, precisamos ter uma estimativa do tamanho (número de tuplas) do arquivo resultante *após* a operações JUNÇÃO. Isso costuma ser mantido como uma razão entre o tamanho (número de tuplas) do arquivo de junção resultante e o tamanho do arquivo de PRODUTO CARTESIANO, se ambos forem aplicados aos mesmos arquivos de entrada, e é chamado de *seletividade de junção* ( $js$ ). Se indicarmos o número de tuplas de uma relação  $R$  como  $|R|$ , temos:

$$js = |(R \bowtie_c S)| / |(R \times S)| = |(R \bowtie_c S)| / (|R| * |S|)$$

Se não houver condição de junção  $c$ , então  $js = 1$  e a junção é igual ao PRODUTO CARTESIANO. Se nenhuma tupla das relações satisfizer a condição de junção, então  $js = 0$ . Em geral,  $0 \leq js \leq 1$ . Para uma junção em que a condição  $c$  é uma comparação de igualdade  $R.A = S.B$ , chegamos aos dois casos especiais a seguir:

1. Se  $A$  é uma chave de  $R$ , então  $|(R \bowtie_c S)| \leq |S|$ , de modo que  $js \leq (1/|R|)$ . Isso porque cada registro no arquivo  $S$  será juntado com no máximo um registro no arquivo  $R$ , pois  $A$  é uma chave de  $R$ . Um caso especial dessa condição é quando o atributo  $B$  é uma *chave estrangeira* de  $S$  que referencia a *chave primária*  $A$  de  $R$ . Além disso, se a chave estrangeira  $B$  tiver a restrição NOT NULL, então  $js = (1/|R|)$ , e o arquivo de resultado da junção terá  $|S|$  registros.
2. Se  $B$  é uma chave de  $S$ , então  $|(R \bowtie_c S)| \leq |R|$ , de modo que  $js \leq (1/|S|)$ .

Ter uma estimativa da seletividade de junção para condições de junção que ocorrem normalmente permite que o otimizador de consulta estime o tamanho do arquivo resultante após a operação de junção, dados os tamanhos dos dois arquivos de entrada, usando a fórmula  $|(R \bowtie_c S)| = js * |R| * |S|$ . Agora, podemos dar alguns exemplos de funções de custo *aproximado* para estimar o custo de alguns dos

algoritmos de junção dados na Seção 19.3.2. As operações de junção têm a forma:

$$R \bowtie_{A=B} S$$

onde  $A$  e  $B$  são atributos compatíveis em domínio de  $R$  e  $S$ , respectivamente. Suponha que  $R$  tenha  $b_R$  blocos e que  $S$  tenha  $b_S$  blocos:

- **J1 — Junção de loop aninhado.** Suponha que usemos  $R$  para o loop externo; depois obtemos a seguinte função de custo para estimar o número de blocos para este método, considerando *três buffers de memória*. Consideramos que o fator de bloco para o arquivo resultante seja  $bfr_{RS}$  e que a seletividade de junção seja conhecida:

$$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$$

A última parte da fórmula é o custo de gravar o arquivo resultante em disco. Esta fórmula de custo pode ser modificada para levar em conta diferentes números de buffers de memória, conforme apresentados na Seção 19.3.2. Se  $n_B$  buffers da memória principal estiverem disponíveis para realizar a junção, a fórmula de custo torna-se:

$$C_{J1} = b_R + (|b_R/(n_B - 2)| * b_S) + ((js * |R| * |S|)/bfr_{RS})$$

- **J2 — Junção de único loop (usando uma estrutura de acesso para recuperar os registros que combinam).** Se existir um índice para o atributo de junção  $B$  de  $S$  com níveis de índice  $x_B$ , podemos recuperar cada registro  $s$  em  $R$  e depois usar o índice para recuperar todos os registros que combinam  $t$  de  $S$  que satisfazem  $t[B] = s[A]$ . O custo depende do tipo de índice. Para um índice secundário onde  $s_B$  é a cardinalidade de seleção para o atributo de junção  $B$  de  $S$ ,<sup>21</sup> obtemos:

$$C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|)/bfr_{RS})$$

Para um índice de agrupamento onde  $s_B$  é a cardinalidade de seleção de  $B$ , obtemos

$$C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$$

Para um índice primário, obtemos

$$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$$

Se houver uma chave de hash para um dos dois atributos de junção — digamos,  $B$  de  $S$  —, obtemos

$$C_{J2d} = b_R + (|R| * b) + ((js * |R| * |S|)/bfr_{RS})$$

<sup>21</sup> A cardinalidade de seleção foi definida como o número médio de registros que satisfazem uma condição de igualdade em um atributo, que é o número médio de registros que têm o mesmo valor para o atributo e, portanto, serão juntados a um único registro no outro arquivo.

onde  $b \geq 1$  é o número médio de acessos de bloco para recuperar um registro, dado seu valor de chave hash. Normalmente,  $b$  é estimado como sendo 1 para o hashing estático e linear e 2 para o hashing estensível.

- J3 — Junção ordenação-intercalação. Se os arquivos já estiverem ordenados nos atributos de junção, a função de custo para esse método é

$$C_{J3} = b_k + b_s + ((js * |RI| * |SI|) / bfr_{RS})$$

Se tivermos de ordenar os arquivos, o custo de ordenação precisa ser acrescentado. Podemos usar as fórmulas da Seção 19.2 para estimar esse custo.

**Exemplo de uso das funções de custo.** Suponha que tenhamos o arquivo FUNCIONARIO descrito no exemplo da seção anterior, e suponha que o arquivo DEPARTAMENTO da Figura 3.5 consista em  $r_D = 125$  registros armazenados em  $b_D = 13$  blocos de disco. Considere as duas operações de junção:

- OP6: FUNCIONARIO  $\bowtie_{Dnum=Dnumero}$  DEPARTAMENTO  
 OP7: DEPARTAMENTO  $\bowtie_{Cpf\_ger=Cpf}$  FUNCIONARIO

Suponha que tenhamos um índice primário em Dnumero de DEPARTAMENTO com  $x_{Dnumero} = 1$  nível e um índice secundário em Cpf\_ger de DEPARTAMENTO com cardinalidade de seleção  $s_{Cpf\_ger} = 1$  e níveis  $x_{Cpf\_ger} = 2$ . Suponha que a seletividade de junção para OP6 seja  $js_{OP6} = (1/|DEPARTAMENTO|) = 1/125$ , pois Dnumero é uma chave de DEPARTAMENTO. Suponha também que o fator de bloco para o arquivo de junção resultante seja  $bfr_{FD} = 4$  registros por bloco. Podemos estimar os custos no pior caso para a operação JUNÇÃO OP6 usando os métodos aplicáveis J1 e J2, da seguinte forma:

1. Usando o método J1 com FUNCIONARIO como loop externo:

$$\begin{aligned} C_{J1} &= b_F + (b_F * b_D) + ((js_{OP6} * r_F * r_D) / bfr_{FD}) \\ &= 2.000 + (2.000 * 13) + ((1/125) * 10.000 * 125/4) = 30.500 \end{aligned}$$

2. Usando o método J1 com DEPARTAMENTO como loop externo:

$$\begin{aligned} C_{J1} &= b_D + (b_F * b_D) + ((js_{OP6} * r_F * r_D) / bfr_{FD}) \\ &= 13 + (13 * 2.000) + ((1/125) * 10.000 * 125/4) = 28.513 \end{aligned}$$

3. Usando o método J2 com FUNCIONARIO como loop externo:

$$\begin{aligned} C_{J2c} &= b_F + (r_F * (x_{Dnumero} + 1)) + ((js_{OP6} * r_F * r_D) / bfr_{FD}) \\ &= 2.000 + (10.000 * 2) + ((1/125) * 10.000 * 125/4) = 24.500 \end{aligned}$$

4. Usando o método J2 com DEPARTAMENTO

como loop externo:

$$\begin{aligned} C_{J2a} &= b_D + (r_D * (x_{Dnumero} + s_{Dnumero})) + ((js_{OP6} * r_F * r_D) / bfr_{FD}) \\ &= 13 + (125 * (2 + 80)) + ((1/125) * 10.000 * 125/4) = 12.763 \end{aligned}$$

O caso 4 tem a menor estimativa de custo e será escolhido. Observe que, no caso 2, se 15 buffers de memória (ou mais) estivessem disponíveis para executar a junção em vez de apenas três, 13 deles poderiam ser usados para manter a relação DEPARTAMENTO inteira (relação de loop externo) na memória, um poderia ser usado como buffer para o resultado e um seria usado para manter um bloco de cada vez do arquivo FUNCIONARIO (arquivo de loop interno), e o custo para o caso 2 seria drasticamente reduzido para apenas  $b_F + b_D + ((js_{OP6} * r_F * r_D) / bfr_{FD})$ , ou 4.513, conforme discutimos na Seção 19.3.2. Se algum outro número de buffers da memória principal estivesse disponível, digamos,  $n_B = 10$ , então o custo para o caso 2 seria calculado da seguinte forma, que também daria um desempenho melhor do que o caso 4:

$$\begin{aligned} C_{J2} &= b_D + ([b_D / (n_B - 2)] * b_F) + ((js * |RI| * |SI|) / bfr_{RS}) \\ &= 13 + ([13/8] * 2.000) + ((1/125) * 10.000 * 125/4) = 28.513 \\ &= 13 + (2 * 2.000) + 2.500 = 6.513 \end{aligned}$$

Como um exercício, o leitor deverá realizar uma análise semelhante para OP7.

### 19.8.5 Consultas de múltiplas relações e ordenação de JUNÇÃO

As regras de transformação algébrica na Seção 19.7.2 incluem uma regra comutativa e uma regra associativa para a operação de junção. Com essas regras, muitas expressões de junção equivalentes podem ser produzidas. Como resultado, o número de árvores de consulta alternativas cresce muito rapidamente à medida que o número de junções em uma consulta aumenta. Uma consulta que junta  $n$  relações normalmente terá  $n - 1$  operações de junção e, portanto, pode ter um grande número de diferentes ordens de junção. A estimativa do custo de cada árvore de junção possível para uma consulta com um grande número de junções exigirá um tempo substancial do otimizador de consulta. Logo, é preciso realizar alguma poda das árvores de consulta possíveis. Os otimizadores de consulta em geral limitam a estrutura de uma árvore de consulta (junção) à das árvores com profundidade esquerda (ou profundidade direita). Uma árvore com profundidade esquerda é uma árvore binária em que o filho da direita de

cada nó não folha é sempre uma relação da base. O otimizador escolheria a árvore com profundidade esquerda em particular com o custo estimado mais baixo. Dois exemplos de árvores com profundidade esquerda aparecem na Figura 19.7. (Observe que as árvores na Figura 19.5 também são árvores com profundidade esquerda.)

Nas árvores com profundidade esquerda, o filho da direita é considerado a relação interna quando se executa uma junção de loop aninhado, ou a relação de investigação quando se executa uma junção de único loop. Uma vantagem das árvores com profundidade esquerda (ou direita) é que elas são receptivas em pipeline, conforme discutimos na Seção 19.6. Por exemplo, considere a primeira árvore com profundidade esquerda na Figura 19.7 e que o algoritmo de junção é o método de único loop; nesse caso, uma página de disco de tuplas da relação externa é usada para investigar a relação interna em busca de tuplas correspondentes. À medida que tuplas (registros) resultantes são produzidas com base na junção de R1 e R2, elas podem ser utilizadas para investigar R3 para localizar seus registros correspondentes para junção. De modo semelhante, à medida que as tuplas resultantes são produzidas dessa junção, elas poderiam ser usadas para investigar R4. Outra vantagem das árvores com profundidade esquerda (ou direita) é que ter uma relação da base como uma das entradas de cada junção permite que o otimizador utilize quaisquer caminhos de acesso nessa relação que possam ser úteis na execução da junção.

Se a materialização for usada no lugar de pipeline (ver seções 19.6 e 19.7.3), os resultados da junção podem ser materializados e armazenados como relações temporárias. A ideia-chave do ponto de vista do otimizador em relação à ordenação de junção é encontrar uma ordenação que reduza o tamanho dos resultados temporários, pois estes (pipeline ou mate-

rializados) são usados por operadores subsequentes e, portanto, afetam o custo de execução desses operadores.

### 19.8.6 Exemplo para ilustrar a otimização de consulta baseada em custo

Vamos considerar a consulta C2 e sua árvore de consulta mostrada na Figura 19.4(a) para ilustrar a otimização de consulta baseada em custo:

**C2: SELECT** Projnumero, Dnum, Unome, Endereco, Data\_nasc  
**FROM** PROJETO, DEPARTAMENTO,  
**FUNCIONARIO**  
**WHERE** Dnum=Dnumero **AND** Cpf\_ger=Cpf  
**AND** Projlocal='Mauá';

Suponha que tenhamos a informação sobre as relações mostradas na Figura 19.8. As estatísticas de VALOR\_MINIMO e VALOR\_MAXIMO foram normalizadas por clareza. A árvore da Figura 19.4(a) é considerada para representar o resultado do processo algébrico de otimização heurística e o início da otimização baseada em custo (neste exemplo, consideraremos que o otimizador heurístico não leva as operações de projeção para baixo na árvore).

A primeira otimização baseada em custo a considerar é a ordenação de junção. Conforme mencionado anteriormente, pressupomos que o otimizador considera apenas árvores com profundidade à esquerda, de modo que as ordens de junção em potencial — sem PRODUTO CARTESIANO — são:

1. PROJETO  $\bowtie$  DEPARTAMENTO  $\bowtie$  FUNCIONARIO
2. DEPARTAMENTO  $\bowtie$  PROJETO  $\bowtie$  FUNCIONARIO
3. DEPARTAMENTO  $\bowtie$  FUNCIONARIO  $\bowtie$  PROJETO
4. FUNCIONARIO  $\bowtie$  DEPARTAMENTO  $\bowtie$  PROJETO

Suponha que a operação de seleção já tenha sido aplicada à relação PROJETO. Se considerarmos uma técnica materializada, então uma nova relação temporária é criada após cada operação de junção. Para examinar o custo da ordem de junção (1), a primeira junção é entre PROJETO e DEPARTAMENTO. Tanto o método de junção quanto os métodos de acesso para as relações de entrada precisam ser determinados. Como DEPARTAMENTO não tem índice, de acordo com a Figura 19.8, o único método de acesso disponível é uma varredura de tabela (ou seja, uma pesquisa linear). A relação PROJETO terá a operação de seleção realizada antes da junção, de modo que existem duas opções: varredura de tabela (pesquisa linear) ou utilização de seu índice PROJ\_PLOC, de modo que o otimizador deve comparar seus custos estimados.

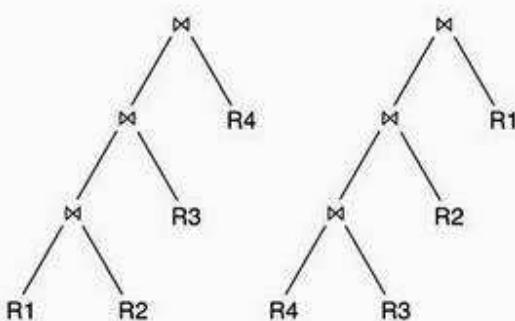


Figura 19.7

Dois árvores de consulta (JUNÇÃO) com profundidade esquerda.

(a)

Nome_tabela	Nome_coluna	Num_distinto	Valor_minimo	Valor_maximo
PROJETO	Projlocal	200	1	200
PROJETO	Projnumero	2.000	1	2.000
PROJETO	Dnum	50	1	50
DEPARTAMENTO	Dnumero	50	1	50
DEPARTAMENTO	Cpf_ger	50	1	50
FUNCIONARIO	Cpf	10.000	1	10.000
FUNCIONARIO	Dnr	50	1	50
FUNCIONARIO	Salario	500	1	500

(b)

Nome_tabela	Num_linhas	Blocos
PROJETO	2.000	100
DEPARTAMENTO	50	5
FUNCIONARIO	10.000	2.000

(c)

Nome_indexe	Exclusividade	Bnível*	Blocos_folha	Chaves_distintas
PROJ_PLOC	NAO UNICA	1	4	200
CPF_FUNC	UNICA	1	50	10.000
SAL_FUNC	NÃO UNICA	1	50	500

\*Bnível é o número de níveis sem o nível folha.

Figura 19.8

Informações estatísticas de exemplo para as relações em C2. (a) Informação de coluna. (b) Informação de tabela. (c) Informação de índice.

A informação estatística no índice PROJ\_PLOC (ver Figura 19.8) mostra o número de níveis de índice  $x = 2$  (raiz mais níveis folha). O índice é não único (porque Projlocal não é uma chave de PROJETO), de modo que o otimizador assume uma distribuição de dados uniforme e estima o número de ponteiros de registro para cada valor de Projlocal como sendo 10. Isso é calculado com base nas tabelas da Figura 19.8 ao multiplicar Seletividade \* Num\_linhas, em que Seletividade é estimado por  $1/\text{Num\_distinto}$ . Assim, o custo do uso do índice e acessos dos registros é estimado como sendo 12 acessos de bloco (2 para o índice e 10 para os blocos de dados). O custo de uma varredura de tabela é estimado como sendo de 100 acessos de bloco, de modo que o acesso ao índice é mais eficiente, conforme esperado.

Em uma técnica materializada, um arquivo temporário TEMP1 com tamanho de 1 bloco é criado para manter o resultado da operação de seleção. O tamanho do arquivo é calculado ao determinar o fator de bloco, usando a fórmula  $\text{Num\_linhas}/\text{Blocos}$ , que gera  $2.000/100$  ou 20 linhas por bloco. Portanto, os 10 re-

gistros selecionados da relação PROJETO caberão em um único bloco. Agora, podemos calcular o custo estimado da primeira junção. Vamos considerar apenas o método de junção de loop aninhado, no qual a relação externa é o arquivo temporário, TEMP1, e a relação interna é DEPARTAMENTO. Como o arquivo TEMP1 inteiro cabe no espaço de buffer disponível, precisamos ler cada um dos cinco blocos da tabela DEPARTAMENTO apenas uma vez, de modo que o custo da junção é de seis acessos de bloco mais o custo de gravar o arquivo de resultado temporário, TEMP2. O otimizador teria de determinar o tamanho de TEMP2. Como o atributo de junção Dnumero é a chave para DEPARTAMENTO, qualquer valor de Dnum de TEMP1 se juntará com no máximo um registro de DEPARTAMENTO, de modo que o número de linhas em TEMP2 será igual ao número de linhas em TEMP1, que é 10. O otimizador determinaria o tamanho do registro para TEMP2 e o número de blocos necessários para armazenar essas 10 linhas. Para simplificar, considere que o fator de bloco para TEMP2 seja cinco linhas por bloco, de modo que um total de dois blocos é necessário para armazenar TEMP2.

Finalmente, o custo da última junção precisa ser estimado. Podemos usar uma junção de único loop em TEMP2, pois nesse caso o índice CPF\_FUNC (ver Figura 19.8) pode ser usado para sondar e localizar registros correspondentes de FUNCIONARIO. Logo, o método de junção envolveria a leitura em cada bloco de TEMP2 e a pesquisa de cada um dos cinco valores de Cpf\_ger usando o índice CPF\_FUNC. Cada pesquisa de índice exigiria um acesso raiz, um acesso de folha e um acesso de bloco de dados ( $x + 1$ , onde o número de níveis  $x$  é 2). Assim, 10 pesquisas exigem 30 acessos de bloco. Somando os dois acessos de bloco para TEMP2, temos um total de 32 acessos de bloco para essa junção.

Para a projeção final, considere que a canalização seja usada para produzir o resultado final, que não exige acessos de bloco adicionais, de modo que o custo total para a ordem de junção (1) é estimado como a soma dos custos anteriores. O otimizador, então, estimaria os custos de uma maneira semelhante para outras três ordens de junção e escolheria aquela com a estimativa mais baixa. Deixamos isso como um exercício para o leitor.

## 19.9 Visão geral da otimização da consulta no Oracle

O SGBD Oracle<sup>22</sup> oferece duas técnicas diferentes para otimização de consulta: baseada em regra e baseada em custo. Com a técnica baseada em regra, o otimizador escolhe planos de execução com base em operações heuristicamente classificadas. O Oracle mantém uma tabela de 15 caminhos de acesso posicionados, em que uma posição mais baixa implica uma técnica mais eficiente. Os caminhos de acesso variam de acesso à tabela por ROWID (o mais eficiente) — em que ROWID especifica o endereço físico do registro, que inclui o arquivo de dados, bloco de dados e deslocamento de linha dentro do bloco — até uma varredura completa da tabela (o menos eficiente) — em que todas as linhas da tabela são pesquisadas ao realizar leituras de múltiplos blocos. No entanto, a técnica baseada em regra está sendo retirada em favor da técnica baseada em custo, na qual o otimizador examina os caminhos de acesso alternativos e os algoritmos de operador, e escolhe o plano de execução com o custo estimado mais baixo. O custo de consulta estimado é proporcional ao tempo gasto esperado para executar a consulta com o plano de execução indicado.

O otimizador do Oracle calcula esse custo com base no uso estimado dos recursos, como E/S, tempo

de CPU e memória necessária. O objetivo da otimização baseada em custo no Oracle é minimizar o tempo decorrido para processar a consulta inteira.

Um acréscimo interessante ao otimizador de consulta do Oracle é a capacidade para um desenvolvedor de aplicação especificar dicas ao otimizador.<sup>23</sup> A ideia é que um desenvolvedor de aplicação possa saber mais informações sobre os dados do que o otimizador. Por exemplo, considere a tabela FUNCIONARIO mostrada na Figura 3.6. A coluna Sexo dessa tabela tem apenas dois valores distintos. Se houver 10.000 funcionários, então o otimizador estimaria que metade é do sexo masculino e metade é do sexo feminino, considerando uma distribuição de dados uniforme. Se existir um índice secundário, será mais do que provável que não seja usado. Porém, se o desenvolvedor da aplicação souber que há apenas 100 funcionários do sexo masculino, uma dica poderia ser especificada em uma consulta SQL cuja condição da cláusula WHERE fosse Sexo = 'M', de modo que o índice associado seja usado no processamento da consulta. Diversas dicas podem ser especificadas, como:

- A técnica de otimização para uma instrução SQL.
- O caminho de acesso para uma tabela acessada pela instrução.
- A ordem de junção para uma instrução de junção.
- Uma operação de junção em particular em uma instrução de junção.

A otimização baseada em custo do Oracle 8 e versões mais recentes são um bom exemplo da técnica sofisticada utilizada para otimizar as consultas SQL em SGBDRs comerciais.

## 19.10 Otimização de consulta semântica

Uma técnica diferente para a otimização de consulta, chamada otimização de consulta semântica, foi sugerida. Essa técnica, que pode ser utilizada em combinação com as técnicas discutidas anteriormente, usa restrições especificadas no esquema de banco de dados — como atributos exclusivos e outras restrições mais complexas — a fim de modificar uma consulta para outra que seja mais eficiente de executar. Não discutiremos essa técnica com detalhes, mas a ilustraremos com um exemplo simples. Considere a consulta SQL:

<sup>22</sup> A discussão nesta seção é baseada principalmente na versão 7 do Oracle. Mais técnicas de otimização foram acrescentadas a versões subsequentes.

<sup>23</sup> Essas dicas também são chamadas de anotações da consulta.

```
SELECT F.Uname, G.Uname
FROM FUNCIONARIO AS F, FUNCIONARIO AS G
WHERE F.Cpf_supervisor=G.Cpf AND F.Salario >
G.Salario
```

Essa consulta recupera os nomes dos funcionários que ganham mais do que seus supervisores. Suponha que tivéssemos uma restrição no esquema de banco de dados que indicasse que nenhum funcionário pode ganhar mais do que seu supervisor direto. Se o otimizador de consulta semântica verificar a existência dessa restrição, ele não precisa executar a consulta de forma alguma, pois sabe que seu resultado será vazio. Isso pode economizar bastante tempo se a verificação de restrição puder ser feita de modo eficiente. Contudo, a pesquisa por meio de muitas restrições para encontrar aquelas que se aplicam a determinada consulta e que podem otimizá-la semanticamente também pode ser muito demorada. Com a inclusão de regras ativas e metadados adicionais nos sistemas de banco de dados (ver Capítulo 26), as técnicas de otimização semântica estão sendo gradualmente incorporadas nos SGBDs.

## Resumo

Neste capítulo, demos uma visão geral das técnicas usadas pelos SGBDs no processamento e otimização de consultas de alto nível. Primeiro, discutimos como as consultas SQL são traduzidas em álgebra relacional e, depois, como diversas operações da álgebra relacional podem ser executadas por um SGBD. Vimos que algumas operações, particularmente SELEÇÃO e JUNÇÃO, podem ter muitas opções de execução. Também abordamos como as operações podem ser combinadas durante o processamento da consulta para criar execução canalizada ou baseada em fluxo, em vez da execução materializada.

Depois disso, descrevemos as técnicas heurísticas para a otimização de consulta, que usam regras heurísticas e técnicas algébricas para melhorar a eficiência da execução da consulta. Mostramos como uma árvore de consulta que representa uma expressão da álgebra relacional pode ser heuristicamente otimizada ao reorganizar os nós de árvore e transformá-los em outra árvore de consulta equivalente, que é mais eficiente de executar. Também demos regras de transformação de preservação da equivalência, que podem ser aplicadas a uma árvore de consulta. Então, introduzimos os planos de execução de consulta para consultas SQL, que acrescentam planos de execução de método às operações da árvore de consulta.

Discutimos a técnica baseada em custo para a otimização da consulta. Mostramos como as funções de custo são desenvolvidas para algoritmos de acesso ao banco de dados e como essas funções de custo são usadas para estimar os custos de diferentes estratégias de execução. Apresentamos uma visão geral do otimizador de consulta do Oracle e mencionamos a técnica da otimização de consulta semântica.

## Perguntas de revisão

- 19.1. Discuta os motivos para converter consultas SQL em consultas da álgebra relacional antes que a otimização seja feita.
- 19.2. Discuta os diferentes algoritmos para implementação de cada um dos seguintes operadores relacionais e as circunstâncias sob as quais cada algoritmo pode ser usado: SELEÇÃO, JUNÇÃO, PROJEÇÃO, UNIÃO, INTERSECÇÃO, DIFERENÇA DE CONJUNTO, PRODUTO CARTESIANO.
- 19.3. O que é um plano de execução de consulta?
- 19.4. O que significa o termo *otimização heurística*? Discuta as principais heurísticas que são aplicadas durante a otimização da consulta.
- 19.5. Como uma árvore de consulta representa uma expressão da álgebra relacional? O que significa a execução de uma árvore de consulta? Discuta as regras para transformação de árvores de consulta e identifique quando cada regra deve ser aplicada durante a otimização.
- 19.6. Quantas ordens de junção diferentes existem para uma consulta que junta 10 relações?
- 19.7. O que significa *otimização de consulta baseada em custo*?
- 19.8. Qual é a diferença entre *pipelining* e *materialização*?
- 19.9. Discuta os componentes de custo para uma função de custo que é usada para estimar o custo de execução da consulta. Quais componentes de custo são utilizados com mais frequência como base para as funções de custo?
- 19.10. Discuta os diferentes tipos de parâmetros que são usados nas funções de custo. Onde essa informação é mantida?
- 19.11. Liste as funções de custo para os métodos SELEÇÃO e JUNÇÃO discutidos na Seção 19.8.
- 19.12. O que significa otimização de consulta semântica? Como ela difere das outras técnicas de otimização de consulta?

## Exercícios

- 19.13. Considere as consultas SQL C1, C8, C1B e C4 do Capítulo 4 e C27 do Capítulo 5.
  - a. Desenhe pelo menos duas árvores de consulta que podem representar *cada uma* dessas consultas. Sob que circunstâncias você usaria *cada uma* de suas árvores de consulta?
  - b. Desenhe a árvore de consulta inicial para *cada uma* dessas consultas e depois mostre como a árvore de consulta é otimizada pelo algoritmo esboçado na Seção 19.7.
  - c. Para *cada* consulta, compare suas árvores de consulta da parte (a) e as árvores de consulta inicial e final da parte (b).

- 19.14. Um arquivo com 4.096 blocos deve ser ordenado com um espaço de buffer disponível de 64 blocos. Quantas passadas serão necessárias na fase de intercalação do algoritmo ordenação-intercalação externo?
- 19.15. Desenvolva funções de custo para os algoritmos PROJEÇÃO, UNIÃO, INTERSECÇÃO, DIFERENÇA DE CONJUNTO e PRODUTO CARTESIAN discutidos na Seção 19.4.
- 19.16. Desenvolva funções de custo para um algoritmo que consiste em duas SELEÇÕES, uma JUNÇÃO e uma PROJEÇÃO final, em relação às funções de custo para as operações individuais.
- 19.17. Um índice não denso pode ser usado na implementação de um operador de agregação? Por quê?
- 19.18. Calcule as funções de custo para diferentes opções de execução da operação JUNÇÃO OPT discutida na Seção 19.3.2.
- 19.19. Desenvolva fórmulas para o algoritmo junção hash híbrido para calcular o tamanho do buffer para o primeiro bucket. Desenvolva fórmulas de estimativa de custo mais precisas para o algoritmo.
- 19.20. Estime o custo das operações OP6 e OP7, usando as fórmulas desenvolvidas no Exercício 19.9.
- 19.21. Estenda o algoritmo de junção ordenação-intercalação para implementar a operação JUNÇÃO EXTERNA À ESQUERDA.
- 19.22. Compare o custo de dois planos de consulta diferentes para a consulta a seguir:

$$\sigma_{\text{Salario} > 40.000}(\text{FUNCIONARIO} \bowtie_{\text{Dnr} = \text{Dnumero}} \text{DEPARTAMENTO})$$

Use as estatísticas de banco de dados da Figura 19.8.

## Bibliografia selecionada

Um algoritmo detalhado para otimização da álgebra relacional é dado por Smith e Chang (1975). A tese de doutorado de Kooi (1980) oferece uma base para as técnicas de processamento de consulta. Um estudo de

Jarke e Koch (1984) contém uma taxonomia da otimização de consulta e inclui uma bibliografia do trabalho nessa área. Um estudo de Graefe (1993) discute a execução da consulta nos sistemas de banco de dados e inclui uma extensa bibliografia.

Whang (1985) discute a otimização de consulta no OBE (Office-By-Example), que é um sistema baseado na linguagem QBE. A otimização baseada em custo foi introduzida no SGBD experimental SYSTEM R e é discutida em Astrahan et al. (1976). Selinger et al. (1979) é um artigo clássico que discutiu a otimização baseada em custo das junções de múltiplas vias no SYSTEM R. Algoritmos de junção são discutidos em Gotlieb (1975), Blasgen e Eswaran (1976) e Whang et al. (1982). Os algoritmos de hashing para implementar junções são descritos e analisados em DeWitt et al. (1984), Bratbergsengen (1984), Shapiro (1986), Kitsuregawa et al. (1989), e Blakeley e Martin (1990), entre outros. Técnicas para encontrar uma boa ordem de junção são apresentadas em Ioannidis e Kang (1990) e em Swami e Gupta (1989). Uma discussão das implicações das árvores de junção com profundidade à esquerda e bushy (frondosas) é apresentada em Ioannidis e Kang (1991). Kim (1982) discute as transformações de consultas SQL aninhadas para representações canônicas. A otimização das funções de agregação é discutida em Klug (1982) e Muralikrishna (1992). Salzberg et al. (1990) descrevem um algoritmo de ordenação externa rápida. A estimativa do tamanho das relações temporárias é crucial para a otimização de consulta. Os esquemas de estimativa baseados em amostragem são apresentados em Haas et al. (1995) e em Haas e Swami (1995). Lipton et al. (1990) também discutem a estimativa de seletividade. Fazer que o sistema de banco de dados armazene e use estatísticas detalhadas na forma de histogramas é o assunto de Muralikrishna e DeWitt (1988) e Poosala et al. (1996).

Kim et al. (1985) discutem tópicos avançados na otimização de consulta. A otimização de consulta semântica é discutida em King (1981) e Malley e Zdonick (1986). O trabalho sobre otimização de consulta semântica é relatado em Chakravarthy et al. (1990), Shenoy e Ozsoyoglu (1989) e Siegel et al. (1992).