Elmasri • Navathe

Sistemas de banco de dados

6ª edição

ALWAYS LEARNING

PEARSON

Elmasri • Navathe

Sistemas de banco de dados

6ª edição

Ramez Elmasri
Departamento de Ciência da Computação e Engenharia
Universidade do Texas em Arlington
Shamkant B. Navathe
Faculdade de Computação
Georgia Institute of Technology

Tradução:
Daniel Vieira
Revisão técnica:
Enzo Seraphim
Thatyana de Faria Piola Seraphim

Professores Doutores do Instituto de Engenharia de Sistemas e Tecnologias da Informação — Universidade Federal de Itajubá

PEARSON

Tradução autorizada a partir da edição original, em inglês, *Fundamentals of Database Systems*, 6th edition, publicada pela Pearson Education, Inc., sob o selo Addison-Wesley.

Diretor editorial: Roger Trimer
Gerente editorial: Sabrina Cairo
Supervisor de produção editorial: Marcelo Françozo
Editora plena: Thelma Babaoka
Editora de texto: Sabrina Levensteinas
Preparação: Paula Brandão Perez Mendes
Revisão: Elisa Andrade Buzzo
Capa: Thyago Santos sobre o projeto original de Lou Gibbs/Getty Images
Projeto gráfico e diagramação: Globaltec Artes Gráficas Ltda.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Elmasri, Ramez

Sistemas de banco de dados / Ramez Elmasri e Shamkant B. Navathe ; tradução Daniel Vieira ; revisão técnica Enzo Seraphim e Thatyana de Faria Piola Seraphim. -- 6. ed. -- São Paulo : Pearson Addison Wesley, 2011.

Título original: Fundamentals of database systems.

ISBN 978-85-4301-381-7

1. Banco de dados I. Navathe, Shamkant B..II. Título.

10-11462 CDD-005.75

Índices para catálogo sistemático:
1. Banco de dados : Sistemas : Processamento de dados 005.75
2. Banco de dados : Fundamentos : Processamento de dados 005.75

3ª reimpressão – Julho 2014

Rua Nelson Francisco, 26
CEP 02712-100 – São Paulo – SP – Brasil
Fone: (11) 2178-8686 – Fax: (11) 2178-8688
vendas@pearson.com

capítulo 5

Mais SQL: Consultas complexas, triggers, views e modificação de esquema

Este capítulo descreve recursos mais avançados da linguagem SQL padrão para bancos de dados relacionais. Começamos na Seção 5.1 apresentando recursos mais complexos das consultas de recuperação SQL, como consultas aninhadas, tabelas com junções, junções externas, funções de agregação e agrupamento. Na Seção 5.2, descrevemos o comando CREATE ASSERTION, que permite a especificação de restrições mais gerais sobre o banco de dados. Também apresentamos o conceito de triggers (gatilhos) e o comando CREATE TRIGGER, que será descrito com mais detalhes na Seção 26.1, quando mostraremos os princípios dos bancos de dados ativos. Depois, na Seção 5.3, descrevemos a facilidade da SQL para definir views (visões) no banco de dados. As views também são chamadas de *tabelas virtuais* ou *derivadas*, pois apresentam ao usuário o que parecem ser tabelas; porém, a informação nessas tabelas é derivada de tabelas previamente definidas. A Seção 5.4 apresenta o comando SQL ALTER TABLE, que é usado para modificar as tabelas e restrições do banco de dados. No final há um resumo do capítulo.

Este capítulo é uma continuação do Capítulo 4. O leitor poderá pular partes deste capítulo se desejar uma introdução menos detalhada à linguagem SQL.

5.1 Consultas de recuperação SQL mais complexas

Na Seção 4.3, descrevemos alguns tipos básicos de consultas de recuperação em SQL. Por causa da generalidade e do poder expressivo da linguagem, existem muitos outros recursos adicionais que permitem que os usuários especifiquem recuperações mais complexas do banco de dados. Discutiremos vários desses recursos nesta seção.

5.1.1 Comparações envolvendo NULL e lógica de três valores

A SQL tem diversas regras para lidar com valores NULL. Lembre-se, da Seção 3.1.2, que NULL é usado para representar um valor que está faltando, mas que em geral tem uma de três interpretações diferentes — valor *desconhecido* (existe, mas não é conhecido), valor *não disponível* (existe, mas é propositadamente retido) ou valor *não aplicável* (o atributo é indefinido para essa tupla). Considere os seguintes exemplos para ilustrar cada um dos significados de NULL.

1. **Valor desconhecido.** A data de nascimento de uma pessoa não é conhecida, e por isso é representada por NULL no banco de dados.
2. **Valor indisponível ou retido.** Uma pessoa tem um telefone residencial, mas não deseja que ele seja listado, por isso ele é retido e representado como NULL no banco de dados.
3. **Atributo não aplicável.** Um atributo Conjuge seria NULL para uma pessoa que não fosse casada, pois ele não se aplica a essa pessoa.

Normalmente, não é possível determinar qual dos significados é intencionado; por exemplo, um NULL para o telefone residencial de uma pessoa pode ter qualquer um dos três significados. Logo, a SQL não distingue entre os diferentes significados de NULL.

Em geral, cada valor NULL individual é considerado diferente de qualquer outro valor NULL nos diversos registros do banco de dados. Quando um

NULL está envolvido em uma operação de comparação, o resultado é considerado UNKNOWN, ou desconhecido (e pode ser TRUE ou FALSE). Assim, a SQL usa uma lógica de três valores com os valores TRUE, FALSE e UNKNOWN em vez da lógica de dois valores (booleana) padrão, com os valores TRUE e FALSF. Portanto, é necessário definir os resultados (ou valores verdadeiros) das expressões lógicas de três valores quando os conectivos lógicos AND, OR e NOT forem usados. A Tabela 5.1 mostra os valores resultantes.

Nas Tabelas 5.1(a) e 5.1(b), as linhas e colunas representam os valores dos resultados das condições de comparação, que normalmente apareceriam na cláusula WHERE de uma consulta SQL. Cada resultado de expressão teria um valor TRUE, FALSE ou UNKNOWN. O resultado da combinação de dois valores usando o conectivo lógico AND é mostrado pelas entradas na Tabela 5.1(a). A Tabela 5.1(b) mostra o resultado do uso do conectivo lógico OR. Por exemplo, o resultado de (FALSE AND UNKNOWN) é FALSE, ao passo que o resultado de (FALSE OR UNKNOWN) é UNKNOWN. A Tabela 5.1(c) mostra o resultado da operação lógica NOT. Observe que, na lógica booleana padrão, somente valores TRUE e FALSE são permitidos; não existe um valor UNKNOWN.

Nas consultas seleção-projeção-junção, a regra geral é que somente as combinações de tuplas que avaliam a expressão lógica na cláusula WHERE da consulta como TRUE são selecionadas. As combinações de tupla que são avaliadas como FALSE ou UNKNOWN não são selecionadas. Porém, existem exceções a essa regra para certas operações, como junções externas (*outer joins*), conforme veremos na Seção 5.1.6.

Tabela 5.1
Conectivos lógicos na lógica de três valores.

(a)	**AND**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)	**OR**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)	**NOT**	
	TRUE	FALSE
	FALSE	TRUE
	UNKNOWN	UNKNOWN

A SQL permite consultas que verificam se o valor de um atributo é **NULL**. Em vez de usar = ou <> para comparar o valor de um atributo com NULL, a SQL usa os operadores de comparação **IS** ou **IS NOT**. Isso porque ela considera cada valor NULL sendo distinto de cada outro valor NULL, de modo que a comparação de igualdade não é apropriada. Acontece que, quando uma condição de junção é especificada, as tuplas com valores NULL para os atributos de junção não são incluídas no resultado (a menos que seja uma OUTER JOIN; ver Seção 5.1.6). A Consulta 18 ilustra isso.

Consulta 18. Recuperar os nomes de todos os funcionários que não possuem supervisores.

```
C18:  SELECT  Pnome, Unome
      FROM    FUNCIONARIO
      WHERE   Cpf_supervisor IS NULL;
```

5.1.2 Consultas aninhadas, tuplas e comparações de conjunto/multiconjunto

Algumas consultas precisam que os valores existentes no banco de dados sejam buscados e depois usados em uma condição de comparação. Essas consultas podem ser formuladas convenientemente usando **consultas aninhadas**, que são blocos select-from-where completos dentro da cláusula WHERE de outra consulta. Essa outra consulta é chamada de **consulta externa**. A Consulta 4 é formulada em C4 sem uma consulta aninhada, mas pode ser reformulada para usar consultas aninhadas, como mostramos em C4A. A C4A introduz o operador de comparação **IN**, que compara um valor v com um conjunto (ou multiconjunto) de valores V e avalia como **TRUE** se v for um dos elementos em V.

A primeira consulta aninhada seleciona os números dos projetos que possuem um funcionário com sobrenome 'Silva' envolvido como gerente, enquanto a segunda consulta aninhada seleciona os números dos projetos que possuem um funcionário com o sobrenome 'Silva' envolvido como trabalhador. Na consulta externa, usamos o conectivo lógico **OR** para recuperar uma tupla PROJETO se o valor de PROJNUMERO dessa tupla estiver no resultado de qualquer uma das consultas aninhadas.

```
C4A: SELECT DISTINCT Projnumero
     FROM   PROJETO
     WHERE  Projnumero IN
            ( SELECT    Projnumero
              FROM      PROJETO,
                        DEPARTAMENTO,
                        FUNCIONARIO
                        Dnum=Dnumero AND
              WHERE     Cpf_gerente=Cpf
                        AND Unome='Silva' )
          OR Projnumero IN
            ( SELECT    Pnr
              FROM      TRABALHA_EM,
                        FUNCIONARIO
              WHERE     Fcpf=Cpf AND
                        Unome='Silva' );
```

Se uma consulta aninhada retornar um único atributo *e* uma única tupla, o resultado da consulta será um único valor (escalar). Nesses casos, é permitido usar = em vez de IN para o operador de comparação. Em geral, a consulta aninhada retornará uma **tabela** (relação), que é um conjunto ou multiconjunto de tuplas.

A SQL permite o uso de **tuplas** de valores em comparações, colocando-os entre parênteses. Para ilustrar isso, considere a seguinte consulta:

```
SELECT DISTINCT Fcpf
FROM   TRABALHA_EM
WHERE  (Pnr, Horas) IN ( SELECT Pnr, Horas
                         FROM   TRABALHA_EM
                         WHERE  Fcpf='12345678966' );
```

Essa consulta selecionará os Fcpfs de todos os funcionários que trabalham na mesma combinação (projeto, horas) em algum projeto que o funcionário 'João Silva' (cujo Cpf = '12345678966') trabalha. Neste exemplo, o operador IN compara a subtupla de valores entre parênteses (Pnr, Horas) dentro de cada tupla em TRABALHA_EM com o conjunto de tuplas com tipos compatíveis produzidas pela consulta aninhada.

Além do operador IN, diversos outros operadores de comparação podem ser usados para comparar um único valor *v* (em geral, um nome de atributo) com um conjunto ou multiconjunto *v* (tipicamente, uma consulta aninhada). O operador = ANY (ou = SOME) retorna TRUE se o valor *v* for igual a *algum valor* no conjunto *V* e, portanto, é equivalente a IN. As duas palavras-chave ANY e SOME possuem o mesmo efeito. Outros operadores que podem ser combinados com ANY (ou SOME) incluem >, >=, <, <= e <>. A palavra-chave ALL também pode ser combinada com cada um desses operadores. Por exemplo, a condição de comparação (*v* > ALL *V*) retorna TRUE se o valor *v* é maior do que *todos* os valores no conjunto (ou multiconjunto) *V*. Um exemplo é a consulta a seguir, que retorna os nomes dos funcionários cujo salário é maior do que o salário de todos os funcionários no departamento 5:

```
SELECT Unome, Pnome
FROM   FUNCIONARIO
WHERE  Salario > ALL ( SELECT  Salario
                       FROM    FUNCIONARIO
                       WHERE   Dnr=5 );
```

Observe que essa consulta também pode ser especificada usando a função de agregação MAX (ver Seção 5.1.7).

Em geral, podemos ter vários níveis de consultas aninhadas. Podemos mais uma vez lidar com a possível ambiguidade entre nomes de atributo se existirem atributos com o mesmo nome — um em uma relação na cláusula FROM da *consulta exterior* e outro em uma relação na cláusula FROM da *consulta aninhada*. A regra é que uma referência a um *atributo não qualificado* refere-se à relação declarada na **consulta aninhada mais interna**. Por exemplo, nas cláusulas SELECT e WHERE da primeira consulta aninhada de C4A, uma referência a qualquer atributo não qualificado da relação PROJETO refere-se à relação PROJETO especificada na cláusula FROM da consulta aninhada. Para se referir a um atributo da relação PROJETO especificada na consulta externa, especificamos e nos referimos a um *apelido* (variável de tupla) para essa relação. Essas regras são semelhantes às regras de escopo para variáveis de programa na maioria das linguagens de programação que permitem procedimentos e funções aninhadas. Para ilustrar a ambiguidade em potencial dos nomes de atributo nas consultas aninhadas, considere a Consulta 16.

Consulta 16. Recuperar o nome de cada funcionário que tem um dependente com o mesmo nome e com o mesmo sexo do funcionário.

```
C16: SELECT F.Pnome, F.Unome
     FROM   FUNCIONARIO AS F
     WHERE  F.Cpf IN ( SELECT D.Fcpf
                       FROM   DEPENDENTE
                              AS D
                       WHERE  F.Pnome=
                              D.Nome_
                                dependente
                              AND
                              F.Sexo=
                              D.Sexo );
```

Na consulta aninhada de C16, temos de qualificar F.Sexo porque se refere ao atributo Sexo de FUNCIONARIO da consulta externa, e DEPENDENTE também tem um atributo chamado Sexo. Se houvesse quaisquer referências não qualificadas a Sexo na consulta aninhada, elas se refeririam ao atributo Sexo de DEPENDENTF. No entanto, não *teríamos de* qualificar os atributos Pnome e Cpf de FUNCIONARIO se eles aparecessem na consulta aninhada, pois a relação DEPENDENTE não tem atributos chamados Pnome e Cpf, de modo que não existe ambiguidade.

É aconselhável criar variáveis de tupla (*apelidos*) para *todas as tabelas referenciadas em uma consulta SQL*, para evitar erros e ambiguidades em potencial, conforme ilustrado em C16.

5.1.3 Consultas aninhadas correlacionadas

Sempre que uma condição na cláusula WHERE de uma consulta aninhada referencia algum atributo de uma relação declarada na consulta externa, as duas consultas são consideradas **correlacionadas**. Podemos entender melhor uma consulta correlacionada ao considerar que a *consulta aninhada é avaliada uma vez para cada tupla (ou combinação de tuplas) na consulta externa*. Por exemplo, podemos pensar em C16 da seguinte forma: para *cada* tupla FUNCIONARIO, avalie a consulta aninhada, que recupera os valores de Fcpf para todas as tuplas de DEPENDENTE com o mesmo sexo e nome da tupla em FUNCIONARIO; se o valor de Cpf da tupla FUNCIONARIO estiver *no* resultado da consulta aninhada, então selecione essa tupla FUNCIONARIO.

Em geral, uma consulta escrita com blocos aninhados select-from-where e usando os operadores de comparação = ou IN *sempre* pode ser expressa como uma única consulta em bloco. Por exemplo, C16 pode ser escrito como em C16A:

```
C16A: SELECT F.Pnome, F.Unome
      FROM   FUNCIONARIO AS F,
             DEPENDENTE AS D
      WHERE  F.Cpf=D.Fcpf AND F.Sexo=D.Sexo
             AND F.Pnome=D.Nome_dependente;
```

5.1.4 As funções EXISTS e UNIQUE em SQL

A função EXISTS em SQL é usada para verificar se o resultado de uma consulta aninhada correlacionada é *vazio* (não contém tuplas) ou não. O resultado de EXISTS é um valor booleano **TRUE** se o resultado da consulta aninhada tiver pelo menos uma tupla, ou **FALSE**, se o resultado da consulta aninhada não tiver tuplas. Ilustramos o uso de EXISTS — e NOT EXISTS — com alguns exemplos. Primeiro, formulamos a Consulta 16 de uma forma alternativa, que usa EXISTS, como em C16B:

```
C16B: SELECT F.Pnome, F.Unome
      FROM   FUNCIONARIO AS F
      WHERE  EXISTS
             ( SELECT *
             FROM   DEPENDENTE
                    AS D
             WHERE  F.Cpf=D.Fcpf
                    AND F.Sexo=
                    D.Sexo
                    AND F.Pnome=
                    D.Nome_
                    dependente);
```

EXISTS e NOT EXISTS costumam ser usados em conjunto com uma consulta aninhada correlacionada. Em C16B, a consulta aninhada referencia os atributos Cpf, Pnome e Sexo da relação FUNCIONARIO da consulta externa. Podemos pensar em C16B da seguinte forma: para cada tupla FUNCIONARIO, avalie a consulta aninhada, que recupera todas as tuplas DEPENDENTE com os mesmos Fcpf, Sexo e Nome_dependente que a tupla FUNCIONARIO; se houver pelo menos uma tupla EXISTS no resultado da consulta aninhada, então selecionar a tupla FUNCIONARIO. Em geral, EXISTS(C) retorna **TRUE** se existe *pelo menos uma tupla* no resultado da consulta aninhada C, e retorna **FALSE** em caso contrário. Por sua vez, NOT EXISTS(C) retorna **TRUE** se *não houver tuplas* no resultado da consulta aninhada C, e retorna **FALSE** caso contrário. Em seguida, ilustramos o uso de NOT EXISTS.

Consulta 6. Recuperar os nomes de funcionários que não possuem dependentes.

```
C6: SELECT Pnome, Unome
    FROM   FUNCIONARIO
    WHERE  NOT EXISTS ( SELECT*
                        FROM   DEPENDENTE
                        WHERECpf=Fcpf );
```

Em C6, a consulta aninhada correlacionada recupera todas as tuplas de DEPENDENTE relacionadas a uma tupla FUNCIONARIO em particular. Se *não existir nenhuma*, a tupla FUNCIONARIO é selecionada, porque a condição da cláusula **WHERE** será avaliada como **TRUE** nesse caso. Podemos explicar C6 da seguinte forma: para *cada* tupla FUNCIONARIO, a

consulta aninhada correlacionada seleciona todas as tuplas DEPENDENTE cujo valor de Fcpf combina com o Cpf de FUNCIONARIO; se o resultado for vazio, nenhum dependente estará relacionado ao funcionário, de modo que selecionamos essa tupla FUNCIONARIO e recuperamos seu Pnome e Unome.

Consulta 7. Listar os nomes dos gerentes que possuem pelo menos um dependente.

```
C7: SELECT Pnome, Unome
    FROM   FUNCIONARIO
    WHERE  EXISTS ( SELECT *
                    FROM   DEPENDENTE
                    WHERE  Cpf=Fcpf )
           AND
           EXISTS ( SELECT *
                    FROM   DEPARTAMENTO
                    WHERE  Cpf=Cpf_gerente );
```

Uma maneira de escrever essa consulta é mostrada em C7, onde especificamos duas consultas aninhadas correlacionadas; a primeira seleciona todas as tuplas de DEPENDENTE relacionadas a um FUNCIONARIO, e a segunda seleciona todas as tuplas de DEPARTAMENTO gerenciadas pelo FUNCIONARIO. Se pelo menos uma da primeira e pelo menos uma da segunda existir, selecionamos a tupla FUNCIONARIO. Você consegue reescrever essa consulta usando apenas uma única consulta aninhada ou nenhuma consulta aninhada?

A consulta C3, *recuperar o nome de cada funcionário que trabalha em* todos *os projetos controlados pelo departamento número 5*, pode ser escrita usando EXISTS e NOT EXISTS nos sistemas SQL. Mostramos duas maneiras de especificar essa consulta C3 em SQL como C3A e C3B. Este é um exemplo de certos tipos de consultas que exigem *quantificação universal*, conforme discutiremos na Seção 6.6.7. Um modo de escrever essa consulta é usar a construção (*S2* EXCEPT *S1*), conforme explicaremos a seguir, e verificar se o resultado é vazio.[1] Essa opção aparece como C3A.

```
C3A: SELECT  Pnome, Unome
     FROM    FUNCIONARIO
     WHERE   NOT EXISTS ( (
             SELECT      Projnumero
             FROM        PROJETO
             WHERE       Dnum=5)
             EXCEPT      (
             SELECT      Pnr
             FROM        TRABALHA_EM
             WHERE       Cpf=Fcpf) );
```

Em C3A, a primeira subconsulta (que não está correlacionada à consulta externa) seleciona todos os projetos controlados pelo departamento 5, e a segunda subconsulta (que está correlacionada) seleciona todos os projetos em que o funcionário em particular trabalha. Se a diferença de conjunto do resultado da primeira subconsulta menos (SUBTRAÇÃO) (EXCEPT) o resultado da segunda subconsulta for vazio, isso significa que o funcionário trabalha em todos os projetos e, portanto, é selecionado.

A segunda opção aparece como C3B. Observe que precisamos de aninhamento de dois níveis em C3B e que essa formulação é muito mais complexa do que C3A, que usa NOT EXISTS e EXCEPT.

```
C3B:  SELECT  Unome, Pnome
      FROM    FUNCIONARIO
      WHERE   NOT EXISTS
              ( SELECT  *
                FROM    TRABALHA_EM T1
                WHERE   ( T1.Pnr IN
                          ( SELECT  Projnumero
                            FROM    PROJETO
                            WHERE   Dnum=5 )
                AND NOT EXISTS
                ( SELECT  *
                FROM  TRABALHA_EM T2
                WHERE   T2.Fcpf=Cpf
                AND     T2.Pnr=B.Pnr )));
```

Em C3B, a consulta aninhada externa seleciona quaisquer tuplas de TRABALHA_EM (T1) cujo Pnr é de um projeto controlado pelo departamento 5, *se* não houver uma tupla em TRABALHA_EM (T2) com o mesmo Pnr e o mesmo Cpf daquele da tupla FUNCIONARIO em consideração na consulta externa. Se não existir tal tupla, selecionamos a tupla FUNCIONARIO. A forma de C3B combina com a reformulação da Consulta 3 a seguir: selecionar cada funcionário de modo que não exista um projeto controlado pelo departamento 5 em que o funcionário não trabalha. Isso corresponde ao modo como escreveremos essa consulta no cálculo relacional de tupla (ver Seção 6.6.7).

Existe outra função em SQL, UNIQUE(C), que retorna TRUE se não houver tuplas duplicadas no resultado da consulta C; caso contrário, ela retorna FALSF. Isso pode ser usado para testar se o resultado de uma consulta aninhada é um conjunto ou um multiconjunto.

5.1.5 Conjuntos explícitos e renomeação de atributos em SQL

Vimos várias consultas com uma consulta aninhada na cláusula WHERF. Também é possível usar um **conjunto explícito de valores** na cláusula WHERE, em vez de uma consulta aninhada. Esse conjunto é delimitado por parênteses em SQL.

[1] Lembre-se de que EXCEPT é um operador de diferença de conjunto. A palavra-chave MINUS às vezes é usada, por exemplo, no Oracle.

Consulta 17. Recuperar os números do CPF de todos os funcionários que trabalham nos projetos de números 1, 2 ou 3.

```
C17: SELECT   DISTINCT Fcpf
     FROM     TRABALHA_EM
     WHERE    Pnr IN (1, 2, 3);
```

Em SQL, é possível renomear qualquer atributo que apareça no resultado de uma consulta acrescentando o qualificador AS seguido pelo novo nome desejado. Logo, a construção AS pode ser usada para *apelidar* os nomes tanto do atributo quanto da relação, e ele pode ser usado nas cláusulas SELECT e FROM. Por exemplo, a C8A mostra como a consulta C8 da Seção 4.3.2 pode ser ligeiramente alterada para recuperar o último nome de cada funcionário e seu supervisor, enquanto renomeia os atributos resultantes como Nome_funcionario e Nome_supervisor. Os novos nomes aparecerão como cabeçalhos de coluna no resultado da consulta.

```
C8A: SELECT   F.Unome AS Nome_funcionario,
             S.Unome AS Nome_supervisor
     FROM     FUNCIONARIO AS F,
             FUNCIONARIO AS S
     WHERE    F.Cpf_supervisor=S.Cpf;
```

5.1.6 Tabelas de junção em SQL e junções externas (outer joins)

O conceito de uma **tabela de junção** (ou **relação de junção**) foi incorporado na SQL para permitir aos usuários especificar uma tabela resultante de uma operação de junção *na cláusula* FROM de uma consulta. Essa construção pode ser mais fácil de compreender do que misturar todas as condições de seleção e junção na cláusula WHERF. Por exemplo, considere a consulta C1, que recupera o nome e o endereço de todos os funcionários que trabalham para o departamento 'Pesquisa'. Pode ser mais fácil especificar a junção das relações FUNCIONARIO e DEPARTAMENTO primeiro, e depois selecionar as tuplas e atributos desejados. Isso pode ser escrito em SQL como em C1A:

```
C1A: SELECT   Pnome, Unome, Endereco
     FROM     (FUNCIONARIO JOIN
               DEPARTAMENTO
               ON Dnr=Dnumero)
     WHERE    Dnome='Pesquisa';
```

A cláusula FROM em C1A contém uma única *tabela de junção*. Os atributos dessa tabela são todos os atributos da primeira tabela, FUNCIONARIO, seguidos por todos os atributos da segunda tabela, DEPARTAMENTO. O conceito de uma tabela de junção também permite que o usuário especifique diferentes tipos de junção, como NATURAL JOIN (junção natural), e vários tipos de OUTER JOIN (junção externa). Em uma NATURAL JOIN sobre duas reações R e S, nenhuma condição de junção é especificada; cria-se uma *condição EQUIJOIN* implícita para *cada par de atributos com o mesmo nome* de R e S. Cada par de atributos desse tipo é incluído *apenas uma vez* na relação resultante (ver seções 6.3.2 e 6.4.4 para mais detalhes sobre os vários tipos de operações de junção na álgebra relacional).

Se os nomes dos atributos de junção não forem os mesmos nas relações da base, é possível renomear os atributos de modo que eles combinem, e depois aplicar a NATURAL JOIN. Nesse caso, a construção AS pode ser usada para renomear uma relação e todos os seus atributos na cláusula FROM. Isso é ilustrado em C1B, onde a relação DEPARTAMENTO é renomeada como DEP e seus atributos são renomeados como Dnome, Dnr (para combinar com o nome do atributo de junção desejado Dnr na tabela FUNCIONARIO), Cpf_gerente e Data_incio_gerente. O significado da condição de junção para esse NATURAL JOIN é FUNCIONARIO.Dnr=DEPT.Dnr, porque esse é o único par de atributos com o mesmo nome após a renomeação:

```
C1B: SELECT   Pnome, Unome, Endereco
     FROM     (FUNCIONARIO NATURAL JOIN
              (DEPARTAMENTO AS DEP
              (Dnome, Dnr, Cpf_gerente, Data_inicio_gerente)))
     WHERE    Dnome='Pesquisa';
```

O tipo padrão de junção em uma tabela de junção é chamado de **inner join**, onde uma tupla é incluída no resultado somente se uma tupla combinar na outra relação. Por exemplo, na consulta C8A, somente os funcionários que *possuem um supervisor* são incluídos no resultado; uma tupla FUNCIONARIO cujo valor para Cpf_supervisor é NULL é excluída. Se o usuário exigir que todos os funcionários sejam incluídos, uma OUTER JOIN precisa ser usada explicitamente (veja a definição de OUTER JOIN na Seção 6.4.4). Em SQL, isso é tratado especificando explicitamente a palavra-chave OUTER JOIN em uma tabela de junção, conforme ilustrado em C8B:

```
C8B:  SELECT   F.Unome AS Nome_funcionario,
              S.Unome AS Nome_supervisor
      FROM     (FUNCIONARIO AS F LEFT
               OUTER JOIN FUNCIONARIO
               AS S ON F.Cpf_supervisor
               =S.Cpf);
```

Existem diversas operações de junção externa, que discutiremos com mais detalhes na Seção 6.4.4. Em SQL, as opções disponíveis para especificar tabelas de junção incluem INNER JOIN (apenas pares de tuplas que combinam com a condição de junção são recuperadas, o mesmo que JOIN), LEFT OUTER JOIN (toda tupla na tabela da esquerda precisa aparecer no resultado; se ela não tiver uma tupla combinando, ela é preenchida com valores NULL para os atributos da tabela da direita), RIGHT OUTER JOIN (toda tupla na tabela da direita precisa aparecer no resultado; se ela não tiver uma tupla combinando, ela é preenchida com valores NULL para os atributos da tabela da esquerda) e FULL OUTER JOIN. Nas três últimas opções a palavra-chave OUTER pode ser omitida. Se os atributos de junção tiverem o mesmo nome, também é possível especificar a variação de junção natural das junções externas usando a palavra-chave NATURAL antes da operação (por exemplo, NATURAL LEFT OUTER JOIN). A palavra-chave CROSS JOIN é usada para especificar a operação PRODUTO CARTESIANO (ver Seção 6.2.2), embora isso só deva ser feito com o máximo de cuidado, pois gera todas as combinações de tuplas possíveis.

Também é possível *aninhar* especificações de junção; ou seja, uma das tabelas em uma junção pode ela mesma ser uma tabela de junção. Isso permite a especificação da junção de três ou mais tabelas como uma única tabela de junção, o que é chamado de **junção múltipla**. Por exemplo, a C2A é um modo diferente de especificar a consulta C2, da Seção 4.3.1, usando o conceito de uma tabela de junção:

```
C2A: SELECT  Projnumero, Dnum, Unome,
             Endereco, Datanasc
     FROM    ((PROJETO JOIN DEPARTAMENTO
             ON Dnum=Dnumero) JOIN
             FUNCIONARIO ON
             Cpf_gerente =Cpf)
     WHERE   Projlocal='Mauá';
```

Nem todas as implementações de SQL empregaram a nova sintaxe das tabelas de junção. Em alguns sistemas, uma sintaxe diferente foi usada para especificar junções externas usando os operadores de comparação +=, =+ e +=+ para a junção externa esquerda, direta e completa, respectivamente, ao especificar a condição de junção. Por exemplo, essa sintaxe está disponível no Oracle. Para especificar a junção externa esquerda na C8B usando essa sintaxe, poderíamos escrever a consulta C8C, da seguinte forma:

```
C8C: SELECT  F.Unome, S.Unome
     FROM    FUNCIONARIO F,
             FUNCIONARIO S
     WHERE   F.Cpf_supervisor += S.Cpf;
```

5.1.7 Funções de agregação em SQL

Na Seção 6.4.2, apresentaremos o conceito de uma função de agregação como uma operação da álgebra relacional. As **funções de agregação** são usadas para resumir informações de várias tuplas em uma síntese de tupla única. O **agrupamento** é usado para criar subgrupos de tuplas antes do resumo. O agrupamento e a agregação são exigidos em muitas aplicações de banco de dados, e apresentaremos seu uso na SQL por meio de exemplos. Existem diversas funções de agregação embutidas: **COUNT**, **SUM**, **MAX**, **MIN** e **AVG**.[2] A função COUNT retorna o número de tuplas ou valores, conforme especificado em uma consulta. As funções SUM, MAX, MIN e AVG podem ser aplicadas a um conjunto ou multiconjunto de valores numéricos e retornam, respectivamente, a soma, o valor máximo, o valor mínimo e a média desses valores. Essas funções podem ser usadas na cláusula SELECT ou em uma cláusula HAVING (que apresentaremos mais adiante). As funções MAX e MIN também podem ser usadas com atributos que possuem domínios não numéricos, se os valores do domínio tiverem uma *ordenação total* entre si.[3] Vamos ilustrar o uso dessas funções com algumas consultas.

Consulta 19. Achar a soma dos salários de todos os funcionários, o salário máximo, o salário mínimo e a média dos salários.

```
C19: SELECT  SUM (Salario), MAX (Salario),
             MIN (Salario), AVG (Salario)
     FROM    FUNCIONARIO;
```

Se quisermos obter os valores das funções para os funcionários de um departamento específico — digamos, o departamento 'Pesquisa' —, podemos escrever a Consulta 20, na qual as tuplas FUNCIONARIO são restringidas pela cláusula WHERE aos funcionários que trabalham para o departamento 'Pesquisa'.

[2] Funções de agregação adicionais para cálculo estatístico mais avançado foram acrescentadas na SQL-99.

[3] Ordenação total significa que, para dois valores quaisquer no domínio, pode ser determinado que um aparece antes do outro na ordem definida; por exemplo, os domínios DATE, TIME e TIMESTAMP possuem ordenações totais em seus valores, assim como as cadeias alfabéticas.

Consulta 20. Achar a soma dos salários de todos os funcionários do departamento 'Pesquisa', bem como o salário máximo, o salário mínimo e a média dos salários nesse departamento.

```
C20: SELECT  SUM (Salario), MAX (Salario),
             MIN (Salario), AVG (Salario)
     FROM    (FUNCIONARIO JOIN
             DEPARTAMENTO ON Dnr=Dnumero)
     WHERE   Dnome='Pesquisa';
```

Consultas 21 e 22. Recuperar o número total de funcionários na empresa (C21) e o número de funcionários no departamento 'Pesquisa' (C22).

```
C21: SELECT  COUNT (*)
     FROM    FUNCIONARIO;
C22: SELECT  COUNT (*)
     FROM    FUNCIONARIO, DEPARTAMENTO
     WHERE   Dnr=Dnumero
             AND Dnome='Pesquisa';
```

Aqui, o asterisco (*) refere-se às *linhas* (tuplas), de modo que COUNT (*) retorna o número de linhas no resultado da consulta. Também podemos usar a função COUNT para contar os valores em uma coluna, em vez de tuplas, como no exemplo a seguir.

Consulta 23. Contar o número de valores de salário distintos no banco de dados.

```
C23: SELECT  COUNT (DISTINCT Salario)
     FROM    FUNCIONARIO;
```

Se escrevermos COUNT(SALARIO) em vez de COUNT (DISTINCT SALARIO) na C23, então os valores duplicados não serão eliminados. Porém, quaisquer tuplas com NULL para SALARIO não serão contadas. Em geral, valores NULL são **descartados** quando as funções de agregação são aplicadas a determinada coluna (atributo).

Os exemplos anteriores resumem *uma relação inteira* (C19, C21, C23) ou um subconjunto selecionado de tuplas (C20, C22), e, portanto, todos produzem tuplas ou valores isolados. Eles ilustram como as funções são aplicadas para recuperar um valor de resumo ou uma tupla de resumo do banco de dados. Essas funções também podem ser usadas nas condições de seleção envolvendo consultas aninhadas. Podemos especificar uma consulta aninhada correlacionada com uma função de agregação, e depois usar a consulta aninhada na cláusula WHERE de uma consulta externa. Por exemplo, para recuperar os nomes de todos os funcionários que têm dois ou mais dependentes (Consulta 5), podemos escrever o seguinte:

```
C5:  SELECT  Unome, Pnome
     FROM    FUNCIONARIO
     WHERE   ( SELECT COUNT (*)
               FROM  DEPENDENTE
               WHERE Cpf=Fcpf ) >= 2;
```

A consulta aninhada correlacionada conta o número de dependentes que cada funcionário tem; se for maior ou igual a dois, a tupla do funcionário é selecionada.

5.1.8 Agrupamento: as cláusulas GROUP BY e HAVING

Em muitos casos, queremos aplicar as funções de agregação *a subgrupos de tuplas em uma relação*, na qual os subgrupos são baseados em alguns valores de atributo. Por exemplo, podemos querer achar o salário médio dos funcionários *em cada departamento* ou o número de funcionários que trabalham *em cada projeto*. Nesses casos, precisamos **particionar** a relação em subconjuntos de tuplas (ou **grupos**) não sobrepostos. Cada grupo (partição) consistirá nas tuplas que possuem o mesmo valor de algum(ns) atributo(s), chamado(s) **atributo(s) de agrupamento**. Podemos, então, aplicar a função a cada grupo desse tipo independentemente, para produzir informações de resumo sobre cada grupo. A SQL tem uma cláusula **GROUP BY** para essa finalidade. A cláusula GROUP BY especifica os atributos de agrupamento, que *também devem aparecer na cláusula* SELECT, de modo que o valor resultante da aplicação de cada função de agregação a um grupo de tuplas apareça junto com o valor do(s) atributo(s) de agrupamento.

Consulta 24. Para cada departamento, recuperar o número do departamento, o número de funcionários no departamento e seu salário médio.

```
C24: SELECT    Dnr, COUNT (*), AVG (Salario)
     FROM      FUNCIONARIO
     GROUP BY  Dnr;
```

Na C24, as tuplas FUNCIONARIO são divididas em grupos — cada grupo tendo o mesmo valor para o atributo de agrupamento Dnr. Logo, cada grupo contém os funcionários que trabalham no mesmo departamento. As funções COUNT e AVG são aplicadas a cada grupo de tuplas. Observe que a cláusula SELECT inclui apenas o atributo de agrupamento e as funções de agregação a serem aplicadas a cada grupo de tuplas. A Figura 5.1(a) ilustra como o agrupamento funciona na C24; ela também mostra o resultado da C24.

(a)

Pnome	Minicial	Unome	Cpf	. . .	Salario	Cpf_supervisor	Dnr
João	B	Silva	12345678966		30.000	33344555587	5
Fernando	T	Wong	33344555587		40.000	88866555576	5
Ronaldo	K	Lima	66688444476		38.000	33344555587	5
Joice	A	Leite	45345345376	. . .	25.000	33344555587	5
Alice	J	Zelaya	99988777767		25.000	98765432168	4
Jennifer	S	Souza	98765432168		43.000	88866555576	4
André	V	Pereira	98798798733		25.000	98765432168	4
Jorge	E	Brito	88866555576		55.000	NULL	1

Agrupamento de tuplas FUNCIONARIO pelo valor de Dnr

Dnr	Count (*)	Avg (Salario)
5	4	33.250
4	3	31.000
1	1	55.000

Resultado de C24

(b)

Projnome	Projnumero	. . .	Fcpf	Pnr	Horas
ProdutoX	1		12345678966	1	32,5
ProdutoX	1		45345345376	1	20,0
ProdutoY	2		12345678966	2	7,5
ProdutoY	2		45345345376	2	20,0
ProdutoY	2		33344555587	2	10,0
ProdutoZ	3		66688444476	3	40,0
ProdutoZ	3		33344555587	3	10,0
Informatização	10	. . .	33344555587	10	10,0
Informatização	10		99988777767	10	10,0
Informatização	10		98798798733	10	35,0
Reorganização	20		33344555587	20	10,0
Reorganização	20		98765432168	20	15,0
Reorganização	20		88866555576	20	NULL
Novos Benefícios	30		98798798733	30	5,0
Novos Benefícios	30		98765432168	30	20,0
Novos Benefícios	30		99988777767	30	30,0

Após aplicar a cláusula WHERE, mas antes de aplicar HAVING

Estes grupos não são selecionados pela condição de HAVING de C26.

Projnome	Projnumero	. . .	Fcpf	Pnr	Horas
ProdutoY	2		12345678966	2	7,5
ProdutoY	2		45345345376	2	20,0
ProdutoY	2		33344555587	2	10,0
Informatização	10		33344555587	10	10,0
Informatização	10	. . .	99988777767	10	10,0
Informatização	10		98798798733	10	35,0
Reorganização	20		33344555587	20	10,0
Reorganização	20		98765432168	20	15,0
Reorganização	20		88866555576	20	NULL
Novos Benefícios	30		98798798733	30	5,0
Novos Benefícios	30		98765432168	30	20,0
Novos Benefícios	30		99988777767	30	30,0

Após aplicar a condição da cláusula HAVING

Projnome	Count (*)
ProdutoY	3
Informatização	3
Reorganização	3
Novos Benefícios	3

Resultado de C26 (Projnumero não mostrado)

Figura 5.1
Resultados de GROUP BY e HAVING. (a) C24. (b) C26.

Se houver NULLs no atributo de agrupamento, então um **grupo separado** é criado para todas as tuplas com um *valor* NULL *no atributo de agrupamento*. Por exemplo, se a tabela FUNCIONARIO tivesse algumas tuplas com NULL para o atributo de agrupamento Dnr, haveria um grupo separado para essas tuplas no resultado da C24.

Consulta 25. Para cada projeto, recuperar o número do projeto, o nome do projeto e o número de funcionários que trabalham nesse projeto.

```
C25: SELECT   Projnumero, Projnome, COUNT (*)
     FROM     PROJETO, TRABALHA_EM
     WHERE    Projnumero=Pnr
     GROUP BY Projnumero, Projnome;
```

A C25 mostra como podemos usar uma condição de junção em conjunto com GROUP BY. Neste caso, o agrupamento e as funções são aplicados *após* a junção das duas relações. Às vezes, queremos recuperar os valores dessas funções somente para *grupos que satisfazem certas condições*. Por exemplo, suponha que queremos modificar a Consulta 25 de modo que apenas projetos com mais de dois funcionários apareçam no resultado. A SQL oferece uma cláusula **HAVING**, que pode aparecer em conjunto com uma cláusula GROUP BY, para essa finalidade. A cláusula HAVING oferece uma condição sobre a informação de resumo referente ao grupo de tuplas associado a cada valor dos atributos de agrupamento. Somente os grupos que satisfazem a condição são recuperados no resultado da consulta. Isso é ilustrado pela Consulta 26.

Consulta 26. Para cada projeto *em que mais de dois funcionários trabalham*, recupere o número e o nome do projeto e o número de funcionários que trabalham no projeto.

```
C26: SELECT   Projnumero, Projnome, COUNT (*)
     FROM     PROJETO, TRABALHA_EM
     WHERE    Projnumero=Pnr
     GROUP BY Projnumero, Projnome
     HAVING   COUNT (*) > 2;
```

Observe que, embora as condições de seleção na cláusula WHERE limitem as *tuplas* às quais as funções são aplicadas, a cláusula HAVING serve para escolher *grupos inteiros*. A Figura 5.1(b) ilustra o uso de HAVING e apresenta o resultado da C26.

Consulta 27. Para cada projeto, recupere o número e o nome do projeto e o número de funcionários do departamento 5 que trabalham no projeto.

```
C27: SELECT   Projnumero, Projnome, COUNT (*)
     FROM     PROJETO, TRABALHA_EM,
              FUNCIONARIO
     WHERE    Projnumero=Pnr AND
              Cpf=Fcpf AND Dnr=5
     GROUP BY Projnumero, Projnome;
```

Aqui, restringimos as tuplas na relação (e, portanto, as tuplas em cada grupo) àquelas que satisfazem a condição especificada na cláusula WHERE — a saber, que eles trabalham no departamento número 5. Observe que precisamos ter um cuidado extra quando duas condições diferentes se aplicam (uma para a função de agregação na cláusula SELECT e outra para a função na cláusula HAVING). Por exemplo, suponha que queremos contar o número *total* de funcionários cujos salários são superiores a R$ 40.000 em cada departamento, mas somente para os departamentos em que há mais de cinco funcionários trabalhando. Aqui, a condição (SALARIO > 40.000) se aplica apenas à função COUNT na cláusula SELECT. Suponha que escrevamos a seguinte consulta *incorreta*:

```
SELECT   Dnome, COUNT (*)
FROM     DEPARTAMENTO, FUNCIONARIO
WHERE    Dnumero=Dnr AND Salario>40.000
GROUP BY Dnome
HAVING   COUNT (*) > 5;
```

Ela está incorreta porque selecionará somente departamentos que tenham mais de cinco funcionários *que ganham cada um mais de R$ 40.000*. A regra é que a cláusula WHERE é executada primeiro, para selecionar as tuplas individuais ou tuplas de junção; a cláusula HAVING é aplicada depois, para selecionar grupos individuais de tuplas. Logo, as tuplas já estão restritas a funcionários que ganham mais de R$ 40.000 *antes* que a cláusula HAVING seja aplicada. Um modo de escrever essa consulta corretamente é usar uma consulta aninhada, como mostra a Consulta 28.

Consulta 28. Para cada departamento que tem mais de cinco funcionários, recuperar o número do departamento e o número de seus funcionários que estão ganhando mais de R$ 40.000.

```
C28: SELECT Dnumero, COUNT (*)
     FROM   DEPARTAMENTO, FUNCIONARIO
     WHERE  Dnumero=Dnr AND Salario>40.000 AND
            ( SELECT  Dnr IN
              FROM  FUNCIONARIO
              GROUP BY Dnr
              HAVING  COUNT (*) > 5)
```

5.1.9 Discussão e resumo das consultas em SQL

Uma consulta de recuperação em SQL pode consistir em até seis cláusulas, mas somente as duas primeiras — SELECT e FROM — são obrigatórias. A consulta pode se espalhar por várias linhas, e termina com um sinal de ponto e vírgula. Os termos da consulta são separados por espaços, e parênteses podem ser usados para agrupar partes relevantes de uma consulta na forma padrão. As cláusulas são especificadas na seguinte ordem, sendo que os colchetes [...] são opcionais:

SELECT <lista atributo e função>
FROM <lista tabela>
[**WHERE** <condição>]
[**GROUP BY** <atributo(s) de agrupamento>]
[**HAVING** <condição de grupo>]
[**ORDER BY** <lista atributos>];

A cláusula SELECT lista os atributos ou funções a serem recuperadas. A cláusula FROM especifica todas as relações (tabelas) necessárias na consulta, incluindo as relações, mas não aquelas nas consultas aninhadas. A cláusula WHERE especifica as condições para selecionar as tuplas dessas relações, incluindo as condições de junção, se necessário. A GROUP BY especifica atributos de agrupamento, enquanto a HAVING especifica uma condição sobre os grupos selecionados, em vez das tuplas individuais. As funções de agregação embutidas COUNT, SUM, MIN, MAX e AVG são usadas em conjunto com o agrupamento, mas também podem ser aplicadas a todas as tuplas selecionadas em uma consulta sem uma cláusula GROUP BY. Por fim, ORDER BY especifica uma ordem para exibir o resultado de uma consulta.

Para formular consultas de maneira correta, é útil considerar as etapas que definem o *significado* ou a *semântica* de cada consulta. Uma consulta é avaliada *conceitualmente*[4] aplicando primeiro a cláusula FROM (para identificar todas as tabelas envolvidas na consulta ou materializar quaisquer tabelas de junção), seguida pela cláusula WHERE para selecionar e juntar tuplas, e depois por GROUP BY e HAVING. Conceitualmente, ORDER BY é aplicado no final para classificar o resultado da consulta. Se nenhuma das três últimas cláusulas (GROUP BY, HAVING e ORDER BY) for especificada, podemos *pensar conceitualmente* em uma consulta como sendo executada da seguinte forma: para *cada combinação de tuplas* — uma de cada uma das relações especificadas na cláusula FROM —, avaliar a cláusula WHERE; se ela for avaliada como TRUE, colocar os valores dos atributos especificados na cláusula SELECT dessa combinação de tuplas no resultado da consulta. É óbvio que esse não é um modo eficiente de implementar a consulta em um sistema real, e cada SGBD possui rotinas especiais de otimização de consulta para decidir sobre um plano de execução que seja eficiente para ser executado. Discutiremos sobre o processamento e a otimização da consulta no Capítulo 19.

Em geral, existem várias maneiras de especificar a mesma consulta em SQL. Essa flexibilidade na especificação de consultas possui vantagens e desvantagens. A principal vantagem é que os usuários podem escolher a técnica com a qual estão mais acostumados ao especificar uma consulta. Por exemplo, muitas consultas podem ser especificadas com condições de junção na cláusula WHERE, ou usando relações de junção na cláusula FROM, ou com alguma forma de consultas aninhadas e o operador de comparação IN. Alguns usuários podem se sentir mais confiantes usando uma técnica, enquanto outros podem estar mais acostumados à outra. Do ponto de vista do programador e do sistema com relação à otimização da consulta, é preferível escrever uma consulta com o mínimo de aninhamento e de ordenação possível.

A desvantagem de ter várias maneiras de especificar a mesma consulta é que isso pode confundir o usuário, que pode não saber qual técnica usar para especificar tipos particulares de consultas. Outro problema é que pode ser mais eficiente executar uma consulta especificada de uma maneira do que a mesma consulta especificada de uma maneira alternativa. O ideal é que isso não aconteça: o SGBD deve processar a mesma consulta da mesma maneira, independentemente de como ela é especificada. Mas isso é muito difícil na prática, pois cada SGBD possui diferentes métodos para processar consultas específicas de diversas maneiras. Assim, um fardo adicional sobre o usuário é determinar qual das especificações alternativas é a mais eficiente de se executar. O ideal é que o usuário se preocupe apenas em especificar a consulta corretamente, enquanto o SGBD determinaria como executar a consulta de forma eficiente. Na prática, contudo, ajuda se o usuário souber quais tipos de construções em uma consulta são mais dispendiosas para processar do que outras (ver Capítulo 20).

[4] A ordem real da avaliação de consulta depende da implementação; esse é apenas um modo de visualizar conceitualmente uma consulta a fim de formulá-la de maneira correta.

5.2 Especificando restrições como asserções e ações como triggers

Nesta seção, apresentamos dois recursos adicionais da SQL: o comando **CREATE ASSERTION** e o comando **CREATE TRIGGER**. A Seção 5.2.1 discute o CREATE ASSERTION, que pode ser usado para especificar tipos adicionais de restrições que estão fora do escopo das *restrições embutidas do modelo relacional* (chaves primária e única, integridade de entidade e integridade referencial), que apresentamos na Seção 3.2. Essas restrições embutidas podem ser especificadas dentro do comando **CREATE TABLE** da SQL (ver seções 4.1 e 4.2).

Depois, na Seção 5.2.2, apresentamos CREATE TRIGGER, que pode ser usado para especificar ações automáticas que o sistema de banco de dados realizará quando certos eventos e condições ocorrerem. Esse tipo de funcionalidade costuma ser conhecido como **bancos de dados ativos**. Só apresentamos os fundamentos básicos dos **triggers** neste capítulo, e uma discussão mais completa sobre os bancos de dados ativos pode ser encontrada na Seção 26.1.

5.2.1 Especificando restrições gerais como asserções em SQL

Em SQL, os usuários podem especificar restrições gerais — aquelas que não se encaixam em nenhuma das categorias descritas nas seções 4.1 e 4.2 — por meio de **asserções declarativas**, usando o comando CREATE ASSERTION da DDL. Cada asserção recebe um nome de restrição e é especificada por uma condição semelhante à cláusula WHERE de uma consulta SQL. Por exemplo, para especificar a restrição de que *o salário de um funcionário não pode ser maior que o salário do gerente do departamento para o qual o funcionário trabalha* em SQL, podemos escrever a seguinte asserção:

```
CREATE ASSERTION RESTRICAO_SALARIAL
CHECK ( NOT EXISTS
        ( SELECT        *
        FROM    FUNCIONARIO F,
                FUNCIONARIO G,
                DEPARTAMENTO D
        WHERE   F.Salario>G.Salario
                AND F.Dnr=D.Dnumero
                AND D. Cpf_gerente =G.Cpf ) );
```

O nome de restrição RESTRICAO_SALARIAL é seguido pela palavra-chave CHECK, que é seguida por uma **condição** entre parênteses que precisa ser verdadeira em cada estado do banco de dados para que a asserção seja satisfeita. O nome da restrição pode ser usado mais tarde para se referir à restrição ou para modificá-la ou excluí-la. O SGBD é responsável por garantir que a condição não seja violada. Qualquer condição de cláusula WHERE pode ser usada, mas muitas restrições podem ser especificadas usando o estilo EXISTS e NOT EXISTS das condições em SQL. Sempre que alguma tupla no banco de dados fizer que a condição de um comando ASSERTION seja avaliada como FALSE, a restrição é **violada**. A restrição é **satisfeita** por um estado do banco de dados se *nenhuma combinação de tuplas* nesse estado do banco de dados violar a restrição.

A técnica de uso comum para escrever essas asserções é especificar uma consulta que seleciona quaisquer tuplas *que violam a condição desejada*. Ao incluir essa consulta em uma cláusula NOT EXISTS, a asserção especificará que o resultado dessa consulta precisa ser vazio para que a condição seja sempre TRUF. Assim, uma asserção é violada se o resultado da consulta não for vazio. No exemplo anterior, a consulta seleciona todos os funcionários cujos salários são maiores que o salário do gerente de seu departamento. Se o resultado da consulta não for vazio, a asserção é violada.

Observe que a cláusula CHECK e a condição de restrição também podem ser utilizadas para especificar restrições sobre atributos e domínios *individuais* (ver Seção 4.2.1) e sobre tuplas *individuais* (ver Seção 4.2.4). A principal diferença entre CREATE ASSERTION e as restrições de domínio e de tupla individuais é que as cláusulas CHECK sobre atributos, domínios e tuplas individuais são verificadas na SQL *somente quando as tuplas são inseridas ou atualizadas*. Logo, a verificação de restrição pode ser implementada de maneira mais eficiente pelo SGBD nesses casos. O projetista do esquema deve usar CHECK sobre atributos, domínios e tuplas apenas quando estiver certo de que a restrição *só pode ser violada pela inserção ou atualização de tuplas*. Além disso, o projetista do esquema deve usar CREATE ASSERTION somente em casos em que não é possível usar CHECK sobre atributos, domínios ou tuplas, de modo que verificações simples são implementadas de modo mais eficiente pelo SGBD.

5.2.2 Introdução às triggers em SQL

Outro comando importante em SQL é o CREATE TRIGGER. Em muitos casos, é conveniente especificar um tipo de ação a ser tomada quando certos eventos ocorrem e quando certas condições são satisfeitas. Por exemplo, pode ser útil especificar uma condição que, se violada, faz que algum usuário seja informado dela. Um gerente pode querer ser informado se as despesas de viagem de um funcionário excederem certo limite, recebendo uma mensagem sempre que isso acontecer. A ação que o SGBD deve tomar nesse caso é enviar uma mensagem apropriada a esse usuário. A condição, portanto, é usada para **monitorar**

o banco de dados. Outras ações podem ser especificadas, como executar um procedimento armazenado (*stored procedure*) específico ou disparar outras atualizações. A instrução CREATE TRIGGER é utilizada para implementar essas ações em SQL. Discutiremos sobre triggers (gatilhos) com detalhes na Seção 26.1, quando descreveremos os *bancos de dados ativos*. Aqui, vamos apenas dar um exemplo simples de como os triggers podem ser usadas.

Suponha que queiramos verificar se o salário de um funcionário é maior que o salário de seu supervisor direto no banco de dados EMPRESA (ver figuras 3.5 e 3.6). Vários eventos podem disparar essa regra: inserir um novo registro de funcionário, alterar o salário de um funcionário ou alterar o supervisor de um funcionário. Suponha que a ação a ser tomada seria chamar o procedimento armazenado,[5] que notificará o supervisor. O trigger poderia então ser escrita como em R5, a seguir. Aqui, estamos usando a sintaxe do sistema de banco de dados Oracle.

```
R5: CREATE TRIGGER VIOLACAO_SALARIAL
    BEFORE INSERT OR UPDATE OF SALARIO,
    CPF_SUPERVISOR ON FUNCIONARIO
FOR EACH ROW
     WHEN ( NEW.SALARIO >
          ( SELECT SALARIO FROM FUNCIONARIO
          WHERE CPF = NEW.CPF_SUPERVISOR ) )
          INFORMAR_SUPERVISOR
          (NEW.Cpf_supervisor,
          NEW.Cpf );
```

O trigger recebe o nome VIOLACAO_SALARIAL, que pode ser usada para remover ou desativar o trigger mais tarde. Um trigger típico tem três componentes:

1. O(s) **evento(s):** estes em geral são operações de atualização no banco de dados, aplicadas explicitamente a ele. Neste exemplo, os eventos são: inserir um novo registro de funcionário, alterar o salário de um funcionário ou alterar o supervisor de um funcionário. A pessoa que escreve o trigger precisa garantir que todos os eventos possíveis sejam considerados. Em alguns casos, pode ser preciso escrever mais de um trigger para cobrir todos os casos possíveis. Esses eventos são especificados após a palavra-chave **BEFORE** em nosso exemplo, o que significa que o trigger deve ser executada antes que a operação de disparo seja executada. Uma alternativa é usar a palavra-chave **AFTER**, que especifica que o trigger deve ser executada após a operação especificada no evento ser concluída.
2. A **condição** que determina se a ação da regra deve ser executada: depois que o evento de disparo tiver ocorrido, uma condição *opcional* pode ser avaliada. Se *nenhuma condição* for especificada, a ação será executada uma vez que o evento ocorra. Se uma condição for especificada, ela primeiro é avaliada e, somente *se for avaliada como verdadeira*, a ação da regra será executada. A condição é especificada na cláusula WHEN do trigger.
3. A **ação** a ser tomada: a ação normalmente é uma sequência de instruções em SQL, mas também poderia ser uma transação de banco de dados ou um programa externo que será executado automaticamente. Neste exemplo, a ação é executar o procedimento armazenado INFORMAR_SUPERVISOR.

Os triggers podem ser usados em várias aplicações, como na manutenção da coerência do banco de dados, no monitoramento de atualizações do banco de dados e na atualização de dados derivados automaticamente. Uma discussão mais completa pode ser vista na Seção 26.1.

5.3 Visões (views) — Tabelas virtuais em SQL

Nesta seção, apresentamos o conceito de uma view (visão) em SQL. Mostraremos como as views são especificadas, depois discutiremos o problema de atualizá-las e como elas podem ser implementadas pelo SGBD.

5.3.1 Conceito de uma view em SQL

Uma **view** em terminologia SQL é uma única tabela que é derivada de outras tabelas.[6] Essas outras tabelas podem ser *tabelas da base* ou views previamente definidas. Uma view não necessariamente existe em forma física; ela é considerada uma **tabela virtual**, ao contrário das **tabelas da base**, cujas tuplas sempre estão armazenadas fisicamente no banco de dados. Isso limita as possíveis operações de atualização que podem ser aplicadas às views, mas não oferece quaisquer limitações sobre a consulta de uma view.

[5] Supondo que um procedimento externo tenha sido declarado. Discutiremos os procedimentos armazenados no Capítulo 13.

[6] Conforme usado em SQL, o termo *view* é mais limitado do que o termo *view do usuário* discutido nos capítulos 1 e 2, pois esta última possivelmente incluiria muitas relações.

Pensamos em uma view como um modo de especificar uma tabela que precisamos referenciar com frequência, embora ela possa não existir fisicamente. Por exemplo, em relação ao banco de dados EMPRESA da Figura 3.5, podemos emitir frequentemente consultas que recuperam o nome do funcionário e os nomes dos projetos em que o funcionário trabalha. Em vez de ter que especificar a junção das três tabelas FUNCIONARIO, TRABALHA_EM e PROJETO toda vez que emitirmos essa consulta, podemos definir uma view que é especificada como o resultado dessas junções. Depois, podemos emitir consultas sobre a view, que são especificadas como leituras de uma única tabela, em vez de leituras envolvendo duas junções sobre três tabelas. Chamamos as tabelas FUNCIONARIO, TRABALHA_EM e PROJETO de **tabelas de definição** da view.

5.3.2 Especificação das views em SQL

Em SQL, o comando para especificar uma view é CREATE VIEW. A view recebe um nome de tabela (virtual), ou nome de view, uma lista de nomes de atributo e uma consulta para especificar o conteúdo da view. Se nenhum dos atributos da view resultar da aplicação de funções ou operações aritméticas, não temos de especificar novos nomes de atributo para a view, pois eles seriam iguais aos nomes dos atributos das tabelas de definição no caso default. As views em V1 e V2 criam tabelas virtuais, cujos esquemas são ilustrados na Figura 5.2, quando aplicadas ao esquema de banco de dados da Figura 3.5.

```
V1: CREATE VIEW TRABALHA_EM1
    AS SELECT   Pnome, Unome, Projnome,
                Horas
       FROM     FUNCIONARIO, PROJETO,
                TRABALHA_EM
       WHERE    Cpf=Fcpf AND Pnr=Projnumero;
V2: CREATE VIEW DEP_INFO(Dep_nome, Qtd_
                func, Total_sal)
    AS SELECT   Dnome, COUNT (*), SUM
                (Salario)
       FROM     DEPARTAMENTO, FUNCIONARIO
       WHERE    Dnumero=Dnr
       GROUP BY Dnome;
```

Em V1, não especificamos quaisquer novos nomes de atributo para a view TRABALHA_EM1 (embora pudéssemos tê-lo feito); nesse caso, TRABALHA_EM1 *herda* os nomes dos atributos de view das tabelas de definição FUNCIONARIO, PROJETO e TRABALHA_EM. A view V2 especifica explicitamente novos nomes de atributo para a view DEP_INFO, usando uma correspondência um para um entre os atributos especificados na cláusula CREATE VIEW e aqueles especificados na cláusula SELECT da consulta que define a view.

TRABALHA_EM1

Pnome	Unome	Projnome	Horas

DEP_INFO

Dep_nome	Qtd_func	Total_sal

Figura 5.2
Duas views especificadas sobre o esquema de banco de dados da Figura 3.5.

Agora, podemos especificar consultas SQL em uma view — ou tabela virtual — da mesma forma como fazemos consultas envolvendo tabelas da base. Por exemplo, para recuperar o primeiro e o último nome de todos os funcionários que trabalham no projeto 'ProdutoX', podemos utilizar a view TRABALHA_EM1 e especificar a consulta como na CV1:

```
CV1: SELECT   Pnome, Unome
     FROM     TRABALHA_EM1
     WHERE    Projnome='ProdutoX';
```

A mesma consulta exigiria a especificação de duas junções se fosse realizada sobre as relações da base diretamente; uma das principais vantagens de uma view é simplificar a especificação de certas consultas. As views também são usadas como um mecanismo de segurança e autorização (ver Capítulo 24).

Supõe-se que uma view esteja *sempre atualizada*; se modificarmos as tuplas nas tabelas da base sobre as quais a view é definida, esta precisa refletir automaticamente essas mudanças. Logo, a view não é realizada ou materializada no momento de sua *definição*, mas quando *especificamos uma consulta* na view. É responsabilidade do SGBD, e não do usuário, cuidar para que a view mantenha-se atualizada. Discutiremos várias maneiras como o SGBD pode manter uma view atualizada na próxima subseção.

Se não precisarmos mais de uma view, podemos usar o comando DROP VIEW para descartá-la. Por exemplo, para descartarmos a view V1, podemos usar o comando SQL em V1A:

```
V1A:    DROP VIEW   TRABALHA_EM1;
```

5.3.3 Implementação e atualização de view e views em linha

O problema de implementar uma view de forma eficiente para consulta é muito complexo. Duas

técnicas principais foram sugeridas. Uma estratégia, chamada **modificação de consulta**, envolve modificar ou transformar a consulta da view (submetida pelo usuário) em uma consulta nas tabelas da base. Por exemplo, a consulta CV1 seria automaticamente modificada para a seguinte consulta pelo SGBD:

```
SELECT  Pnome, Unome
FROM    FUNCIONARIO, PROJETO, TRABALHA_EM
WHERE   Cpf=Fcpf AND Pnr=Projnumero
        AND Projnome='ProdutoX';
```

A desvantagem dessa técnica é que ela é ineficaz para views definidas por consultas complexas, que são demoradas de se executar, especialmente se várias delas tiverem de ser aplicadas à mesma view em um curto período. A segunda estratégia, chamada **materialização de view**, envolve criar fisicamente uma tabela de view temporária quando a view for consultada pela primeira vez e manter essa tabela na suposição de que outras consultas a view acontecerão em seguida. Nesse caso, uma estratégia eficiente para atualizar automaticamente a tabela da view quando as tabelas de base forem atualizadas deverá ser desenvolvida para que a view esteja sempre atualizada. As técnicas que usam o conceito de **atualização incremental** têm sido desenvolvidas para essa finalidade, nas quais o SGBD pode determinar quais novas tuplas devem ser inseridas, excluídas ou modificadas em uma *tabela de view materializada* quando uma atualização de banco de dados é aplicada *a uma das tabelas da base definidas*. A view geralmente é mantida como uma tabela materializada (armazenada fisicamente), desde que esteja sendo consultada. Se a view não for consultada por certo período, o sistema pode então remover automaticamente a tabela física e recalculá-la do zero quando consultas futuras referenciarem a view.

A atualização das views é complicada e pode ser ambígua. Em geral, uma atualização em uma view definida sobre uma *única tabela* sem quaisquer *funções de agregação* pode ser mapeada para uma atualização sobre a tabela da base sob certas condições. Para uma view que envolve junções (*joins*), uma operação de atualização pode ser mapeada para operações de atualização sobre as relações da base de *múltiplas maneiras*. Logo, com frequência não é possível que o SGBD determine qual das atualizações é intencionada. Para ilustrar os problemas em potencial com a atualização de uma view definida sobre múltiplas tabelas, considere a view TRABALHA_EM1 e suponha que emitamos o comando para atualizar o atributo PROJNOME de 'João Silva' de 'ProdutoX' para 'ProdutoY'. Essa atualização de view aparece em UV1:

```
UV1:  UPDATE TRABALHA_EM1
      SET    Projnome = 'ProdutoY'
      WHERE  Unome='Silva' AND Pnome='João'
             AND Projnome='ProdutoX';
```

Essa consulta pode ser mapeada para várias atualizações sobre as relações da base para gerar o efeito de atualização desejado sobre a view. Além disso, algumas das atualizações criarão efeitos colaterais adicionais, que afetam o resultado de outras consultas. Por exemplo, aqui estão duas atualizações possíveis, (a) e (b), sobre as relações da base correspondentes à operação de atualização de view em UV1:

```
(a): UPDATE TRABALHA_EM
     SET    Pnr =    ( SELECT  Projnumero
                      FROM    PROJETO
                      WHERE   Projnome=
                              'ProdutoY' )
     WHERE  Fcpf IN  ( SELECT  Cpf
                      FROM    FUNCIONARIO
                      WHERE   Unome='Silva'
                              AND
                              Pnome='João' )
            AND
            Pnr =    ( SELECT  Projnumero
                      FROM    PROJETO
                      WHERE   Projnome=
                              'ProdutoX' );
(b): UPDATE PROJETO  SET   Projnome =
                           'ProdutoY'
     WHERE  Projnome = 'ProdutoX';
```

A atualização (a) relaciona 'João Silva' à tupla 'ProdutoY' de PROJETO em vez da tupla 'ProdutoX' de PROJETO e é a atualização provavelmente mais desejada. Porém, (b) também daria o efeito de atualização desejado sobre a view, mas realiza isso alterando o nome da tupla 'ProdutoX' na relação PROJETO para 'ProdutoY'. É muito pouco provável que o usuário que especificou a atualização de view UV1 queira que ela seja interpretada como em (b), pois isso também tem o efeito colateral de alterar todas as tuplas de view com Projnome = 'ProdutoX'.

Algumas atualizações de view podem não fazer muito sentido; por exemplo, modificar o atributo Total_sal da view DEP_INFO não faz sentido porque Total_sal é definido como sendo a soma dos salários de funcionário individuais. Esta solicitação aparece como em UV2:

```
UV2:  UPDATE  DEP_INFO
      SET     Total_sal=100.000
      WHERE   Dnome='Pesquisa';
```

Um grande número de atualizações sobre as relações da base pode satisfazer essa atualização de view.

Em geral, uma atualização de view é viável quando somente *uma atualização possível* sobre as relações da base pode realizar seu efeito desejado sobre a view. Sempre que uma atualização sobre a view pode ser mapeada para *mais de uma atualização* sobre as relações da base, precisamos ter um certo procedimento para escolher uma das atualizações possíveis como a mais provável. Alguns pesquisadores desenvolveram métodos para escolher a atualização mais provável, enquanto outros preferem deixar que o usuário escolha o mapeamento de atualização desejado durante a definição da view.

Resumindo, podemos fazer as seguintes observações:

- Uma view com uma única tabela de definição é atualizável se seus atributos tiverem a chave primária da relação da base, bem como todos os atributos com a restrição NOT NULL que *não tem* valor default especificado.
- As views definidas sobre múltiplas tabelas usando junções geralmente não são atualizáveis.
- As views definidas usando funções de agrupamento e agregação não são atualizáveis.

Em SQL, a cláusula **WITH CHECK OPTION** precisa ser acrescentada ao final da definição de view se uma view *tiver de ser atualizada*. Isso permite que o sistema verifique a possibilidade de atualização da view e planeje uma estratégia de execução para ela.

Também é possível definir uma tabela de view na **cláusula FROM** de uma consulta em SQL. Isso é conhecido como uma **view em linha**. Nesse caso, a view é definida na própria consulta.

5.4 Instruções de alteração de esquema em SQL

Nesta seção, oferecemos uma visão geral dos **comandos de evolução de esquema** disponíveis em SQL, que podem ser usados para alterar um esquema, acrescentando ou removendo tabelas, atributos, restrições e outros elementos dele. Isso pode ser feito enquanto o banco de dados está operando e não exige recompilação do esquema. Certas verificações precisam ser feitas pelo SGBD para garantir que as mudanças não afetarão o restante do banco de dados, tornando-o inconsistente.

5.4.1 O comando DROP

O comando DROP pode ser usado para remover elementos *nomeados* do esquema, como tabelas, domínios ou restrições. Também é possível remover um esquema. Por exemplo, se todo um esquema não for mais necessário, o comando DROP SCHEMA pode ser utilizado. Existem duas opções de *comportamento de drop*: CASCADE e RESTRICT. Por exemplo, para remover o esquema de banco de dados EMPRESA e todas as suas tabelas, domínios e outros elementos, a opção CASCADE é usada da seguinte forma:

```
DROP SCHEMA EMPRESA CASCADE;
```

Se a opção RESTRICT for escolhida no lugar da CASCADE, o esquema é removido somente se ele *não tiver elementos*; caso contrário, o comando DROP não será executado. Para usar a opção RESTRICT, o usuário deve primeiro remover individualmente cada elemento no esquema, depois remover o próprio esquema.

Se uma relação da base dentro de um esquema não for mais necessária, a relação e sua definição podem ser excluídas usando o comando DROP TABLF. Por exemplo, se não quisermos mais manter os dependentes dos funcionários no banco de dados EMPRESA da Figura 4.1, podemos descartar a relação DEPENDENTE emitindo o seguinte comando:

```
DROP TABLE DEPENDENTE CASCADE;
```

Se a opção RESTRICT for escolhida em vez da CASCADE, uma tabela é removida somente se ela *não for referenciada* em quaisquer restrições (por exemplo, por definições de chave estrangeira em outra relação) ou views (ver Seção 5.3), ou por quaisquer outros elementos. Com a opção CASCADE, todas essas restrições, views e outros elementos que referenciam a tabela sendo removida também são excluídos automaticamente do esquema, junto com a própria tabela.

Observe que o comando DROP TABLE não apenas exclui todos os registros na tabela se tiver sucesso, mas também remove a *definição de tabela* do catálogo. Se for desejado excluir apenas os registros, mas deixar a definição de tabela para uso futuro, então o comando DELETE (ver Seção 4.4.2) deve ser usado no lugar de DROP TABLF.

O comando DROP também pode ser empregado para descartar outros tipos de elementos de esquema nomeados, como restrições ou domínios.

5.4.2 O comando ALTER

A definição de uma tabela da base ou de outros elementos de esquema nomeados pode ser alterada usando o comando ALTER. Para as tabelas

da base, as possíveis **ações de alteração de tabela** incluem acrescentar ou remover uma coluna (atributo), alterar uma definição de coluna e acrescentar ou remover restrições de tabela. Por exemplo, para incluir um atributo que mantém as tarefas dos funcionários na relação da base FUNCIONARIO do esquema EMPRESA (ver Figura 4.1), podemos usar o comando

ALTER TABLE EMPRESA.FUNCIONARIO **ADD COLUMN** Tarefa VARCHAR(12);

Ainda podemos inserir um valor para o novo atributo Tarefa para cada tupla individual de FUNCIONARIO. Isso pode ser feito especificando uma cláusula default ou usando o comando UPDATE individualmente sobre cada tupla (ver Seção 4.4.3). Se nenhuma cláusula default for especificada, o novo atributo receberá o valor NULL em todas as tuplas da relação imediatamente após o comando ser executado; logo, a restrição NOT NULL *não é permitida* nesse caso.

Para remover uma coluna, temos de escolher CASCADE ou RESTRICT para o comportamento de remoção. Se CASCADE for escolhido, todas as restrições e views que referenciam a coluna são removidas automaticamente do esquema, junto com a coluna. Se RESTRICT for escolhido, o comando só tem sucesso se nenhuma view ou restrição (o outro elemento do esquema) referenciar a coluna. Por exemplo, o comando a seguir remove o atributo Endereco da tabela FUNCIONARIO:

ALTER TABLE EMPRESA.FUNCIONARIO **DROP COLUMN** Endereco **CASCADE**;

Também é possível alterar uma definição de coluna removendo uma cláusula default existente ou definindo uma nova cláusula default. Os exemplos a seguir ilustram essa cláusula:

ALTER TABLE EMPRESA.DEPARTAMENTO **ALTER COLUMN** Cpf_gerente **DROP DEFAULT**;

ALTER TABLE EMPRESA.DEPARTAMENTO **ALTER COLUMN** Cpf_gerente **SET DEFAULT** '33344555587';

Também é possível alterar as restrições especificadas sobre uma tabela ao acrescentar ou remover uma restrição nomeada. Para ser removida, uma restrição precisa ter recebido um nome quando foi especificada. Por exemplo, para descartar a restrição chamada CHESUPERFUNC da Figura 4.2 da relação FUNCIONARIO, escrevemos:

ALTER TABLE EMPRESA.FUNCIONARIO **DROP CONSTRAINT** CHESUPERFUNC **CASCADE**;

Quando isso é feito, podemos redefinir uma restrição substituída acrescentando uma nova restrição à relação, se necessário. Isso é especificado usando a palavra-chave **ADD** na instrução ALTER TABLE seguida pela nova restrição, que pode ser nomeada ou não, e pode ser de qualquer um dos tipos de restrição de tabela discutidos.

As subseções anteriores deram uma visão geral dos comandos de evolução de esquema da SQL. Também é possível criar tabelas e views em um esquema de banco de dados usando os comandos apropriados. Existem muitos outros detalhes e opções; o leitor interessado deverá consultar os documentos sobre SQL listados na 'Bibliografia selecionada', ao final deste capítulo.

Resumo

Neste capítulo, apresentamos recursos adicionais da linguagem de banco de dados em SQL. Começamos na Seção 5.1 apresentando recursos mais complexos das consultas de atualização de SQL, incluindo consultas aninhadas, tabelas de junção, junções externas, funções agregadas e agrupamento. Na Seção 5.2, descrevemos o comando CREATE ASSERTION, que permite a especificação de restrições mais gerais sobre o banco de dados, e apresentamos o conceito de triggers e o comando CREATE TRIGGER. Depois, na Seção 5.3, descrevemos a facilidade da SQL para definir views no banco de dados. As views também são chamadas de *tabelas virtuais* ou *derivadas*, pois apresentam ao usuário o que parecem ser tabelas; no entanto, as informações nessas tabelas são derivadas de outras definidas anteriormente. A Seção 5.4 introduziu o comando ALTER TABLE da SQL, que é usado para modificar as tabelas e restrições do banco de dados.

A Tabela 5.2 resume a sintaxe (ou estrutura) de diversos comandos SQL. Esse resumo não é abrangente, nem descreve cada construção SQL possível; em vez disso, ele serve como uma referência rápida para os principais tipos de construções disponíveis em SQL. Usamos a notação BNF, onde os símbolos não terminais aparecem entre sinais de <...>, as partes opcionais aparecem entre colchetes [...], as repetições aparecem entre chaves {...} e as alternativas aparecem entre parênteses (... | ... | ...).[7]

[7] A sintaxe completa da SQL é descrita em muitos documentos volumosos, com centenas de páginas.

Tabela 5.2
Resumo da sintaxe da SQL.

CREATE TABLE <nome tabela> (<nome coluna> <tipo coluna> [<restrição atributo>] { , <nome coluna> <tipo coluna> [<restrição atributo>] } [<restrição tabela> { , <restrição tabela> }])
DROP TABLE <nome tabela>
ALTER TABLE <nome tabela> ADD <nome coluna> <tipo coluna>
SELECT [DISTINCT] <lista atributos>
FROM (<nome tabela> { <apelido> } | <tabela de junção>) { , (<nome tabela> { <apelido> } | <tabela de junção>) }
[WHERE <condição>]
[GROUP BY <atributos agrupamento> [HAVING <condição seleção grupo>]]
[ORDER BY <nome coluna> [<ordem>] { , <nome coluna> [<ordem>] }]
<lista atributos> ::= (* | (<nome coluna> | <função> (([DISTINCT] <nome coluna> | *))) { , (<nome coluna> | <função> (([DISTINCT] <nome coluna> | *)) }))
<atributos agrupamento> ::= <nome coluna> { , <nome coluna> }
<ordem> ::= (ASC | DESC)
INSERT INTO <nome tabela> [(<nome coluna> { , <nome coluna> })]
(VALUES (<valor constante> , { <valor constante> }) { , (<valor constante> { , <valor constante> }) }
| <instrução seleção>)
DELETE FROM <nome tabela>
[WHERE <condição seleção>]
UPDATE <nome tabela>
SET <nome coluna> = < expressão valor> { , <nome coluna> = <expressão valor> }
[WHERE <condição seleção>]
CREATE [UNIQUE] INDEX <nome índice>
ON <nome tabela> (<nome coluna> [<ordem>] { , <nome coluna> [<ordem>] })
[CLUSTER]
DROP INDEX <nome índice>
CREATE VIEW <nome view> [(<nome coluna> { , <nome coluna> })]
AS <instrução seleção>
DROP VIEW <nome view>

NOTA: Os comandos para criar e excluir índices não fazem parte do padrão SQL.

Perguntas de revisão

5.1. Descreva as seis cláusulas na sintaxe de uma consulta de recuperação SQL. Mostre que tipos de construções podem ser especificados em cada uma das seis cláusulas. Quais das seis cláusulas são obrigatórias e quais são opcionais?

5.2. Descreva conceitualmente como uma consulta de recuperação SQL será executada, especificando a ordem conceitual de execução de cada uma das seis cláusulas.

5.3. Discuta como os NULLs são tratados nos operadores de comparação em SQL. Como os NULLs são tratados quando funções de agregação são aplicadas em uma consulta SQL? Como os NULLs são tratados quando existem nos atributos de agrupamento?

5.4. Discuta como cada uma das seguintes construções é usada em SQL e quais são as diversas opções para cada construção. Especifique a utilidade de cada construção.

a. Consultas aninhadas.
b. Tabelas de junção e junções externas.
c. Funções de agregação e agrupamento.
d. Triggers.
e. Asserções e como elas diferem dos triggers.
f. Views e suas formas de atualização.
g. Comandos de alteração de esquema.

Exercícios

5.5. Especifique as seguintes consultas no banco de dados da Figura 3.5 em SQL. Mostre os resultados da consulta se cada uma for aplicada ao banco de dados da Figura 3.6.
   a. Para cada departamento cujo salário médio do funcionário seja maior do que R$30.000,00, recupere o nome do departamento e o número de funcionários que trabalham nele.
   b. Suponha que queiramos o número de funcionários do sexo *masculino* em cada departamento que ganhe mais de R$30.000,00, em vez de todos os funcionários (como no Exercício 5.5a). Podemos especificar essa consulta em SQL? Por quê?

5.6. Especifique as seguintes consultas em SQL sobre o esquema de banco de dados da Figura 1.2.
   a. Recupere os nomes e departamentos de todos os alunos com notas A (alunos que têm uma nota A em todos as disciplinas).
   b. Recupere os nomes e departamentos de todos os alunos que não têm uma nota A em qualquer uma das disciplinas.

5.7. Em SQL, especifique as seguintes consultas sobre o banco de dados da Figura 3.5 usando o conceito de consultas aninhadas e conceitos descritos neste capítulo.
   a. Recupere os nomes de todos os funcionários que trabalham no departamento que tem o funcionário com o maior salário entre todos os funcionários.
   b. Recupere os nomes de todos os funcionários cujo supervisor do supervisor tenha como Cpf o número '88866555576'.
   c. Recupere os nomes dos funcionários que ganham pelo menos R$10.000,00 a mais que o funcionário que recebe menos na empresa.

5.8. Especifique as seguintes views em SQL no esquema de banco de dados EMPRESA mostrado na Figura 3.5.
   a. Uma view que tem o nome do departamento, nome do gerente e salário do gerente para todo departamento.
   b. Uma view que tenha o nome do funcionário, nome do supervisor e salário de cada funcionário que trabalha no departamento 'Pesquisa'.
   c. Uma view que tenha o nome do projeto, nome do departamento que o controla, número de funcionários e total de horas trabalhadas por semana em cada projeto.
   d. Uma view que tenha o nome do projeto, nome do departamento que o controla, número de funcionários e total de horas trabalhadas por semana no projeto para cada projeto *com mais de um funcionário trabalhando nele*.

5.9. Considere a seguinte view, RESUMO_DEPARTAMENTO, definida sobre o banco de dados EMPRESA da Figura 3.6:

```
CREATE VIEW   RESUMO_DEPARTAMENTO (D,
              C, Total_sal, Media_sal)
AS SELECT     Dnr, COUNT (*), SUM (Salario),
              AVG (Salario)
FROM          FUNCIONARIO
GROUP BY      Dnr;
```

Indique quais das seguintes consultas e atualizações seriam permitidas sobre a view. Se uma consulta ou atualização for permitida, mostre como ficaria a consulta ou atualização correspondente nas relações da base e seu resultado quando aplicado ao banco de dados da Figura 3.6.

a.
```
SELECT   *
FROM     RESUMO_DEPARTAMENTO;
```
b.
```
SELECT   D, C
FROM     RESUMO_DEPARTAMENTO
WHERE    TOTAL_SAL > 100.000;
```
c.
```
SELECT   D, MEDIA_SAL
FROM     RESUMO_DEPARTAMENTO
WHERE    C > ( SELECT C FROM
         RESUMO_DEPARTAMENTO
         WHERE D=4);
```
d.
```
UPDATE   RESUMO_DEPARTAMENTO
SET      D=3
WHERE    D=4;
```
e.
```
DELETE   FROM RESUMO_
         DEPARTAMENTO
WHERE    C > 4;
```

Bibliografia selecionada

Reisner (1977) descreve uma avaliação dos fatores humanos da SEQUEL, precursora da SQL, em que descobriu que os usuários possuem alguma dificuldade de

para especificar corretamente as condições de junção e agrupamento. Date (1984) contém uma crítica da linguagem SQL que indica seus portos fortes e fracos. Date e Darwen (1993) descrevem a SQL2. ANSI (1986) esboça o padrão SQL original. Diversos manuais de fabricante descrevem as características da SQL implementadas em DB2, SQL/DS, Oracle, INGRES, Informix e outros produtos de SGBD comerciais. Melton e Simon (1993) oferecem um tratamento abrangente do padrão ANSI 1992, chamado SQL2. Horowitz (1992) discute alguns dos problemas relacionados à integridade referencial e propagação das atualizações em SQL2.

A questão de atualizações de view é abordada por Dayal e Bernstein (1978), Keller (1982) e Langerak (1990), entre outros. A implementação de view é discutida em Blakeley et al. (1989). Negri et al. (1991) descrevem a semântica formal das consultas SQL.

Existem muitos livros que descrevem vários aspectos da SQL. Por exemplo, duas referências que descrevem a SQL-99 são Melton e Simon (2002) e Melton (2003). Outros padrões SQL — SQL 2006 e SQL 2008 — são descritos em diversos relatórios técnicos; mas não existem referências padrão.