Pós-Graduação
UNICAMP – Instituto de Computação

Resumo do capítulo 5
SQL: Consultas, Restrições e Gatilhos

Trabalho apresentado à disciplina de pós-graduação Banco de Dados I, da Universidade Estadual de Campinas.

Gabriel Fróes Franco
Felipe de Almeida Leme

Disciplina: MO410 - Banco de Dados I
Professor: Geovanne Magalhães
Data de Entrega: 09/06/2005

UNICAMP - Campinas
1º semestre de 2005

ÍNDICE

FIGURAS

1. INTRODUÇÃO

A linguagem de consulta estruturada (SQL) é a mais amplamente utilizada linguagem de banco de dados relacionais comerciais. Ela foi originalmente desenvolvida na IBM para os projetos SEQUEL-XRM e System-R (1974-1977). Quase imediatamente, outros produtores introduziram produtos DBMS (*Data Base Management Systems*, ou *Sistemas Gerenciadores de Banco de Dados*) baseados na SQL, o que o concretizou como um padrão de fato. A SQL continuou a evoluir em respostas às mudanças nas necessidades da área de banco de dados, sendo a versão SQL:1999 a mais recente no momento.

As principais características da SQL são:

- A Linguagem de Manipulação de Dados (DML – *Data Manipulation Language*)
- A Linguagem de Definição de Dados (DDL – *Data Definition Language*)
- Gatilhos e Restrições Avançadas de Integridade
- SQL dinâmico e embutido.
- Execução Cliente-Servidor e Acesso Remoto ao Banco de Dados
- Gerenciamento de Transações
- Segurança
- Alguns recursos avançados como: suporte a orientação a objetos, consultas recursivas, consultas de suporte a decisão, data-mining, dados espaciais, textos e XML.

2. FORMA BÁSICA DE UMA CONSULTA SQL

Esta seção apresenta a estrutura básica de uma consulta SQL e explica seu significado através de uma estratégia de avaliação conceitual. A estratégia de avaliação conceitual é uma forma de tornar a avaliação de uma consulta mais fácil, sem se preocupar com o desempenho. Geralmente um DBMS irá executar uma consulta de forma diferente da utilizada pela estratégia de avaliação conceitual (como resultado da atuação automática do *otimizador* do DBMS).

A forma básica de uma consulta SQL é a seguinte:

SELECT [DISTINCT] *lista-seleção*
FROM *lista-de*
[WHERE *qualificação]*

A cláusula **SELECT** define as colunas que farão parte do resultado da consulta, e a cláusula **FROM** especifica o produto cartesiano das tabelas.

A cláusula opcional **WHERE** especifica as condições de seleção das tabelas especificadas na cláusula **FROM**.

A *lista-de* é a lista de nomes das tabelas a serem utilizadas na consulta. Estes nomes podem ser seguidos por uma variável tupla, que define um "apelido" para o nome da tabela. Uma variável tupla é útil quando existem campos com o mesmo nome em várias tabelas da cláusula **FROM**.

A *lista-seleção* é a lista de campos das tabelas que estão na *lista-de* e que farão parte do resultado.

A *qualificação* é uma operação booleana usada para delimitar o resultado da consulta. Note que a cláusula ***WHERE*** *qualificação* é opcional; caso ela não seja usada, a consulta retornara todas as tuplas do produto cartesiano definido pelas colunas da cláusula **FROM**.

A palavra-chave **DISTINCT** é opcional, indicando que a resposta não deverá conter linhas duplicadas. Diferentemente da álgebra relacional, na SQL as linhas duplicadas não são eliminadas por padrão, pois essa é uma operação custosa ao DBMS.

Este tipo de consulta intuitivamente corresponde a uma expressão de álgebra relacional envolvendo seleções, projeções e produtos cartesianos. Esta relação entre a SQL e a álgebra relacional é a base para a otimização da execução pelos DBMS, sendo os planos de execução representados usando uma variação das expressões de álgebra relacional.

Considere o seguinte exemplo:

Exemplo 1: Encontrar o nome e idade de todos os marinheiros:

SELECT DISTINCT S.sname, S.age **FROM** sailors S

Sailors			
sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Reserves		
sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Boats		
bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figura 1 *- Tabelas usadas no Exemplo 1*

O resultado é um conjunto de linhas onde cada linha é composta de um par de valores (*sname*, *age*) representando uma tupla da tabela *sailors* (também identificada pela variável tupla *S*). Se dois ou mais marinheiros tem o mesmo nome e idade, a resposta conterá somente um par com este nome e idade. Esta consulta equivale a aplicar o operador de projeção da álgebra relacional. Se eliminássemos a palavra-chave **DISTINCT**, as linhas com nome e idade iguais seriam repetidas, como demonstra a figura a seguir:

Resultado com DISTINCT	
sname	age
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Art	25.5
Bob	63.5

Resultado sem DISTINCT	
sname	age
Dustin	45.0
Brutus	33.0
Lubber	55.0
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Figura 2 *- Resultado da execução do Exemplo 1*

Exemplo 2: Encontrar todos os marinheiros com avaliação maior que 7

SELECT S.sid, S.sname, S.rating, S.age **FROM** sailors **AS** S **WHERE** S.rating>7

Esta consulta usa a palavra-chave opcional **AS** para identificar as variáveis tupla (note que no exemplo anterior a palavra-chave não foi usada). Um detalhe adicional neste exemplo é a possibilidade de substituir os nomes dos campos da cláusula SELECT por *, uma vez que estão sendo selecionados todos os campos da tabela na cláusula FROM. Esta notação é útil para consultas interativas, porém é inadequado para consultas que serão reutilizadas em aplicações, pois dificulta a manutenção em caso de mudanças no esquema da tabela. Este segundo exemplo demonstra a utilização do operador seleção da álgebra relacional.

Através destes dois exemplos, podemos notar que a cláusula **SELECT** é análoga ao operador **projeção** da álgebra relacional, enquanto a cláusula **WHERE** é relacionada ao operador **seleção** (o que pode causar uma confusão).

Agora que descrevemos informalmente a sintaxe de uma consulta básica em SQL, precisamos entender o que ela significa. O resultado de uma consulta em SQL é uma tabela, cujo conteúdo pode ser entendido considerando a seguinte estratégia de avaliação conceitual:

1. Compute o produto cartesiano de todas as tabelas da *lista-de*.
2. Elimine as linhas resultantes que não atendem às condições de qualificação.
3. Elimine todas as colunas que não estão na *lista-seleção*.
4. Caso a cláusula **DISTINCT** tenha sido utilizada, elimine as linhas duplicadas.

Para entendermos como esta estratégia funciona, iremos ilustrá-la no exemplo a seguir:

Exemplo 3: Encontrar todos os nomes dos marinheiros que reservaram o barco número 103

SELECT S.name **FROM** sailors S, reserves R **WHERE** S.sid = R.sid **AND** R.bid = 103

Para a solução deste exemplo consideraremos as seguintes tabelas:

Sailors			
sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Reserves		
sid	bid	day
22	101	10/10/98
58	103	11/12/96

Figura 3 *– Tabelas Utilizadas no Exemplo 3*

Computando o produto cartesiano das tabelas presentes na *lista-de*, obtemos o seguinte resultado:

sid	***sname***	***rating***	***age***	***sid***	***bid***	***day***
22	Dustin	7	45.0	22	101	10/10/98
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/98
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/98
58	Rusty	10	35.0	58	103	11/12/96

Figura 4 *– Produto cartesiano resultante do Exemplo 3*

O próximo passo é eliminar as linhas que não atendem às expressões da *lista-seleção*, que no exemplo é a expressão *S.sid = R.sid* ***AND*** *R.bid = 103*. Obtemos então o seguinte resultado:

sid	***sname***	***rating***	***age***	***sid***	***bid***	***day***
58	Rusty	10	35.0	58	103	11/12/96

Figura 5 *– Resultado da eliminação das linhas que não atendem a cláusula* ***WHERE*** *do Exemplo 3*

A seguir são eliminadas todas as colunas que não estão presentes na *lista-seleção*:

sname
Rusty

Figura 6 *- Resultado da eliminação das colunas não presentes na lista-seleção do Exemplo 3*

2.1. Expressoões e Strings no comando SELECT

A *lista-seleção* no comando **SELECT** pode suportar mais informações que somente o nome das colunas. Cada item na *lista-seleção* pode ser formado por ***expressão AS nome_coluna*** onde *expressão* pode ser qualquer expressão aritmética ou de comparação de Strings sobre os nomes das colunas e/ou constantes e *nome_coluna* é o nome da nova coluna que armazenará o resultado da consulta. Outra possibilidade são as funções agregadas - tais como **SUM**(nome da coluna) ou **COUNT**(nome da coluna) - que serão discutidos posteriormente.

Exemplo 4: Computar incrementos na avaliação dos marinheiros que navegaram dois barcos diferentes no mesmo dia.

SELECT S.sname, S.srating+1 **AS** rating **FROM** sailors S, reserves R1, reserves R2 **WHERE** S.sid = R1.sid **AND** S.sid = R2.sid **AND** R1.day = R2.day **AND** R1.bid <> R2.bid

No próximo exemplo demonstraremos a atulização de uma expressão na forma ***expressão1 = expressão2***:

Exemplo 5: Utilização de expressões na cláusula qualificação.
SELECT S1.sname **AS** name1, S2.sname **AS** name2 **FROM** sailors S1, sailors S2 **WHERE** 2*S1.rating = S2.rating-1

Para comparações de Strings, podem ser utilizados os operadores de comparação (=, <, >, etc...) além do operador **LIKE** para reconhecimento de padrões. O operador LIKE é tipicamente utilizado em conjunto com caracteres coringas como **%** (que representa um ou mais caracteres) e _ (que representa um único caractere). Desta forma, o padrão **_AB%** denota qualquer String que possui no mínimo 3 caracteres, sendo que o segundo e terceiro devem ser as literais **A** e **B** respectivamente.

Exemplo 6: Encontrar a idade de todos os marinheiros cujo nome inicie e termine com a letra B.
SELECT S.age **FROM** sailors S **WHERE** S.sname **LIKE** 'B%B'

3. UNIÃO, INTERSEÇÃO E EXCEÇÃO.

A linguagem SQL provê 3 mecanismos de manipulação de conjuntos para estender as funcionalidades das consultas. Como o resultado de uma consulta é um conjunto de informações, é natural considerar o uso de operações como união, intersecção e exceção. Estes mecanismos são representados pelas palavras chaves **UNION**, **INTERSECT** e **EXCEPT** (também chamado de **MINUS** em alguns DBMS), respectivamente.

Para demonstrar o uso destes mecanismos, considere os seguintes exemplos:

Exemplo 7: Encontrar o nome de todos os marinheiros que reservaram um barco vermelho ou verde.
SELECT S.sname **FROM** sailors S, reserves R, boats B **WHERE** S.sid = R.sid **AND** R.bid = B.bid **AND** B.color = ‘red’
UNION
SELECT S2.sname **FROM** sailors S2, reserves R2, boats B2 **WHERE** S2.sid = R2.sid **AND** R2.bid = B2.bid **AND** B2.color = ‘green’

Exemplo 8: Encontrar o nome de todos os marinheiros que reservaram um barco vermelho e um verde.
SELECT S.sname **FROM** sailors S, reserves R, boats B **WHERE** S.sid = R.sid **AND** R.bid = B.bid **AND** B.color = ‘red’
INTERSECT
SELECT S2.sname **FROM** sailors S2, reserves R2, boats B2 **WHERE** S2.sid = R2.sid **AND** R2.bid = B2.bid **AND** B2.color = ‘green’

Exemplo 9: Encontrar o nome de todos os marinheiros que reservaram um barco vermelho mas não um verde.
SELECT S.sname **FROM** sailors S, reserves R, boats B **WHERE** S.sid = R.sid **AND** R.bid = B.bid **AND** B.color = ‘red’
EXCEPT
SELECT S2.sname **FROM** sailors S2, reserves R2, boats B2 **WHERE** S2.sid = R2.sid **AND** R2.bid = B2.bid **AND** B2.color = ‘green’

4. CONSULTAS ANINHADAS

Consultas aninhadas é um dos recursos mais poderosos e úteis da linguagem SQL. Uma consulta aninhada é uma consulta secundária embutida dentro de uma consulta principal. Esta consulta embutida também é chamada de sub-consulta. Uma sub-consulta geralmente aparece na cláusula **WHERE** da consulta principal, embora ela também possa fazer parte das cláusulas **FROM** ou **HAVING** (esta última será apresentada mais adiante).

Exemplo 10: Encontra o nome dos marinheiros que reservaram o barco 103.
SELECT S.sname **FROM** sailors S, reserves R, boats B **WHERE**
S.sid **IN (SELECT** R.sid **FROM** reserves R **WHERE** R.bid = 103)

No exemplo apresentado acima, a sub-consulta computa o conjunto de marinheiros que reservaram o barco 103, enquanto a consulta principal retorna o nome dos marinheiros que fazem parte deste conjunto. O operador **IN** permite testar se um valor está ou não presente em um conjunto de valores. Para recuperar todos os marinheiros que não reservaram o barco 103, basta trocar o operador **IN** pelo **NOT IN**.

Para facilitar o entendimento de como as sub-consultas funcionam, abaixo é mostrada a estratégia de avaliação conceitual para uma execução de sub-consultas:

1. Computar o produto cartesiano das tabelas na cláusula **FROM**
2. Para cada linha do produto cartesiano, enquanto testa as qualificações da cláusula **WHERE** re-compute as sub-consultas
3. Elimine as linhas que não atendam as qualificações.
4. Elimine as colunas que não estão presentes na *lista-seleção*
5. Caso a cláusula **DISTINCT** tenha sido utilizada, elimine as linhas duplicadas.

Um outro ponto importante a ser mencionado é a possibilidade de uma sub-consulta aninhada poder conter outras consultas aninhadas, como demonstra o exemplo a seguir:

Exemplo 11: Encontrar o nome de todos os marinheiros que reservaram um barco vermelho. **SELECT** S.sname **FROM** sailors S **WHERE** S.sid **IN**
(**SELECT** R.sid **FROM** reserves R **WHERE** R.sid **IN**
(**SELECT** B.bid **FROM** boats B **WHERE** B.color = ‘red’))

4.1. Consultas aninhadas correlacionadas

Nas consultas aninhadas vistas até agora, as sub-consultas eram totalmente independentes das consultas principais. De uma forma geral, as sub-consultas podem ser dependentes da consulta principal, através do valor que está sendo examinado (em termos de estratégia de avaliação conceitual).

Exemplo 12: Encontra o nome dos marinheiros que reservaram o barco 103.
SELECT S.sname **FROM** sailors S **WHERE EXISTS**
(**SELECT** * **FROM** reserves R **WHERE** R.bid=103 **AND** R.sid = S.sid)

No exemplo anterior o operador **EXISTS** apresenta um outro operador de conjuntos que nos permite testar se o conjunto resultante da consulta não é vazio. Dessa forma, para cada linha de *sailors* S, é testado se o conjunto de linhas *reserves* R contém **R.bid = 103** e **R.sid = S.sid**. Se o marinheiro S reservou o barco 103, seu nome é retornado no conjunto da consulta principal. Um detalhe importante é que a sub-consulta é totalmente dependente do valor na linha atual sendo avaliada da consulta principal, dessa forma para cada linha da consulta principal a sub-consulta deve ser reavaliada.

Um outro operador apresentado nesta consulta é o ***** discutido anteriormente. Este operador faz com que todas as colunas sejam selecionadas e permite ao SGBD otimizar a consulta em casos onde não seja necessária a projeção de resultados (como nesse, onde a cláusula **EXISTS** só se importa com a existência de uma tupla na resposta).

4.2. Operadores de comparações de conjuntos

Além dos operadores já apresentados (**EXISTS**, **IN** e **UNIQUE**) a SQL também suporta outros operadores para operações com conjuntos. Estes operadores são o **ANY** (ou **SOME** em alguns DBMS) e **ALL** que são utilizados junto de um operador aritmético de comparação (<, <=, =. >. <>, >=, >). O exemplo a seguir demonstra a utilização destes operadores:

Exemplo 13: Encontrar os marinheiros com avaliação maior do que qualquer marinheiro chamado Horatio.
SELECT S.sid **FROM** sailors S **WHERE** S.rating > ANY (**SELECT** S2.rating **FROM** sailors S2 **WHERE** S2.sname = ‘Horatio’)

Vale lembrar que os operadores **IN** e **NOT IN** equivalem ao funcionamento de **= ANY** e **<> ALL**, respectivamente.

5. OPERADORES AGREGADOS

A SQL fornece algumas outras funcionalidades além de recuperações de informações. Muitas vezes precisamos de informações de sumarização e/ou cálculo de valores agregados. Para atender à essas necessidades, a SQL suporta 5 operações agregadas:

- **COUNT ([DISTINCT] A)** – quantia de valores (únicos) na coluna A.
- **SUM([DISTINCT] A)** – soma de todos os valores (únicos) na coluna A.
- **AVG([DISTINCT] A)** – média de todos os valores (únicos) na coluna A.
- **MAX(A)** – valor máximo encontrado na coluna A.
- **MIN(A)** – valor mínimo encontrado na coluna A.

Exemplo 14: Encontrar o nome dos marinheiros mais velhos que o mais velho dos marinheiros com avaliação igual a 10.
SELECT S.sname **FROM** sailors S **WHERE** S.age > (**SELECT MAX(**S2.age) **FROM** sailors S2 **WHERE** S2.rating = 10)

Exemplo 15: Encontrar a média de idades entre os marinheiros com avaliação igual a 10.
SELECT AVG(S.age) **FROM** sailors S **WHERE** S.rating = 10

5.1. Cláusulas GROUP BY e HAVING

Até agora vimos o emprego das operações agregadas em todos os valores de uma relação. Pode haver casos, porém, onde seja necessário aplicar os operadores em grupos distintos de valores nos quais o número de valores distintos é desconhecido até o momento da execução. Para ilustrar esta situação, considere a seguinte consulta:

Encontrar a idade do marinheiro mais novo de cada nível de avaliação.

Se soubéssemos que a avaliação varia de 1 a 10 poderíamos escrever 10 consultas da seguinte forma:

SELECT MIN (S.age) **FROM** sailors S **WHERE** S.rating = i;

Onde i = 1,2,3,…,10. Escrever 10 consultas dessa forma é tedioso e impraticável, uma vez que não sabemos os valores de todas avaliação existentes.

Para tornar estas consultas mais práticas precisamos utilizar uma outra extensão à SQL básica, que são as cláusulas **GROUP BY** e **HAVING**. A cláusula **GROUP BY** é utilizada para agrupar os valores obedecendo alguma regra, enquanto a cláusula opcional **HAVING** pode ser utilizada para delimitar os valores a serem agrupados. Um exemplo utilizando a consulta apresentada anteriormente seria:

Exemplo 15: Encontrar a idade do marinheiro mais novo de cada nível de avaliação.
SELECT S.rating, MIN(S.age) **FROM** sailors S **GROUP BY** S.rating

Levando em consideração a seguinte tabela *sailors* para este exemplo, o resultado da consulta seria a relação apresentada na Figura 8:

Sailors			
sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figura 7 - Tabela *sailors* usada no Exemplo 15

rating	age
1	33.0
3	25.5
7	35.0
8	25.5
9	35.0
10	16.0

Figura 8 *- Resultado do Exemplo 15.*

6. VALORES NULOS (NULL)

Até agora assumimos que todos os valores para os conjuntos de resultados retornados por uma consulta são conhecidos. Na prática, porém, existem situações onde o valor de uma coluna pode ser desconhecido ou inaplicável. Um exemplo seria a adição de um novo marinheiro à tabela sailors. Quando um marinheiro é adicionado à tabela sailors, sua avaliação ainda não possui um valor, ou seja, no momento da adição de um novo marinheiro, o valor da avaliação é desconhecido.

Para possibilitar tais situações, a SQL provém um valor chamado **NULL**, o que possibilita forma a adição de um marinheiro com a seguinte tupla: (98, Dan, NULL ,39). Porém a presença do valor **NULL** causa várias complicações que devem ser levadas em consideração, conforme explicado nas próximas seções.

6.1. Comparações com valores NULL

Considerando o marinheiro adicionado (98, Dan, NULL, 39) qual seria o resultado de uma consulta com a comparação rating > 8? Como o valor da avaliação de Dan é **NULL**, qualquer comparação com este valor irá retornar **NULL**.

Existe também uma operação de comparação para valores nulos. O operador **IS NULL** testa se o valor de uma determinada coluna é **NULL**. Este operador pode ser também utilizado junto com o operador **NOT** (ou seja, usando-se a expressão **IS NOT NULL**) para verificar se uma coluna não é nula.

6.2. Conectivos lógicos AND, OR e NOT

O mesmo problema aparece quando utilizamos conectivos lógicos como na expressão *rating > 8* ***AND*** *age < 40*. Para solucionar este problema a SQL utiliza uma lógica de 3 valores onde o resultado pode ser **TRUE**, **FALSE** ou **UNKNOWN**. Essa lógica obedece as seguintes regras:

1. A expressão **NOT** com um valor desconhecido sempre retornará desconhecido.
2. A expressão **OR** retornará **TRUE** se qualquer um dos argumentos for **TRUE**, retornará **UNKNOWN** se um dos argumentos for **FALSE** e o outro **UNKNOWN**.
3. A expressão **AND** retornará **FALSE** se qualquer um dos argumentos forem **FALSE**, retornará **UNKNOWN** se um dos argumentos for **TRUE** e o outro **UNKNOWN**.

6.3. Impactos em expressões SQL.

As expressões de comparações são utilizadas em várias partes da SQL, dessa forma, a maneira como a SQL lida com valores **NULL** gera vários impactos que devem ser reconhecidos.

Um dos impactos causados é na comparação de linhas duplicadas num conjunto. Pela definição da SQL duas linhas são iguais quando os valores de cada coluna nas duas tabelas são iguais ou ambos nulos. Porém, quando 2 valores são comparados atráves do operador de igualdade (**=**), o resultado é **UNKNOWN**. Ou seja, no contexto de duplicação essa comparação retornaria **TRUE**, mas no contexto de comparação de valores ela retornaria **UNKNOW**, o que é uma anomalia.

Outros impactos acontecem com os operadores agregados. As expressões aritiméticas onde um valor é nulo, sempre retornam **NULL**, porém nas operações **SUM()**, **AVG()**, **COUNT()**, **MIN()** e **MAX ()** estes valores são tratados de uma forma diferente: a operação **COUNT()** considera o **NULL** como um valor qualquer e o considera-o no resultado final; já as demais operações agregadas simplesmente ignoram o valor **NULL**, a não ser que todos os valores do conjunto sejam **NULL**, quando a resposta então também é **NULL**.

6.4. Desabilitando os valores NULL

Os valores nulos podem ser desabilitados em uma coluna de uma tabela através do uso da palavra chave **NOT NULL** no comando DDL de criação (ou alteração) da tabela. Uma chave primária automaticamente não aceita valores nulos.

7. RESTRIÇÕES DE INTEGRIDADE COMPLEXAS EM SQL

7.1. Restrições de Integridades Complexas para uma Tabela

Uma restrição de integridade pode ser associada a uma tabela através da opção **CHECK** *expressão-condicional*. Por exemplo, para garantir que a avaliação deve ter um valor entre 1 e 10, a restrição seria feita da seguinte forma:

Exemplo 16: Garantir que o valor da avaliação sempre será um número entre 1 e 10.

```
CREATE TABLE sailors (
sid      INTEGER,
sname    CHAR(10),
rating   INTEGER,
age      REAL,
PRIMARY KEY(sid),
CHECK (rating >= 1 AND rating <= 10)
)
```

7.2. Restrições de Domínio e Tipos Distintos

Uma outra opção para gerar restrições de valores é a criação de domínios de valores. Uma vez que um domínio é definido, ele pode ser utilizado para restringir valores numa tabela.

Exemplo 17: Criação de um domínio para avaliação

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1 CHECK (VALUE >= 1 AND VALUE <= 10)
```

Exemplo 18: Utilizando o domínio em uma tabela.

```
CREATE TABLE sailors (
sid      INTEGER,
sname    CHAR(10),
rating   ratingval,
age      REAL,
PRIMARY KEY(sid)
)
```

Muitas vezes porém queremos que, além de definir um domínio, dois valores de colunas diferentes não possam ser comparados (por exemplo sid e bid). Se os dois forem de domínios que tenham

como base o **INTEGER**, as comparações serão possíveis. Para solucionar este problema, a SQL permite a definição de tipos distintos:

Exemplo 19: Criação de tipos distintos.
CREATE TYPE ratingtype **AS INTEGER**

Dessa forma, a comparação entre um *ratingtype* e qualquer outro valor será proibiad pela SQL.

7.3. Asserções: Restrições de Integridade em várias tabelas

Existem casos onde uma restrição pode englobar várias tabelas de forma que fica difícil (senão impossível) colocar um **CHECK** em cada tabela para verificar a restrição. Para solucionar este problema a SQL permite a criação de *asserções* que são restrições associadas a várias tabelas.

Exemplo 20: Criação de uma asserção
CREATE ASSERTION smallClub **CHECK ((SELECT COUNT**(S.sid**) FROM** sailors S**) + (SELECT COUNT** (B.bid**) FROM** boats B**) >** 100**)**

8. GATILHOS E BANCOS DE DADOS ATIVOS

Um gatilho (ou *trigger*) é um procedimento que é executado pelo DBMS em resposta a algum evento ou mudança na base de dados. Uma base de dados que possua 1 ou mais gatilhos é também chamada de base de dados ativa.

Um gatilho é formado por 3 partes:

- ***evento*** que dispara o gatilho
- ***condição*** para execução do gatilho
- ***ação*** executada pelo gatilho.

Exemplo 21: Criação de um gatilh

```
CREATE TRIGGER init_count BEFORE INSERT ON sailors
        DECLARE count INTEGER
        BEGIN count := 0;
        END
CREATE TRIGGER incr_count AFTER INSERT ON sailors
        WHEN  (new.age > 21)
        FOR EACH ROW
        BEGIN
        Count := count +1;
        END
```

No exemplo apresentado acima, as cláusulas **BEFORE INSERT ON** e **AFTER INSERT ON** representam os eventos que devem ocorrer para que o gatilho seja ativado, enquanto que os comandos entre as cláusulas **BEGIN** e **END** indicam as ações a serem executadas e a cláusula **WHEN** define a condição que deve ser verdadeira para que a ação seja executada.

BIBLIOGRAFIA

RAMAKRISHNAN, Raghu, JOHANNES, Gehrke. **Database Management Systems.** 3rd Ed. McGraw Hill 2003