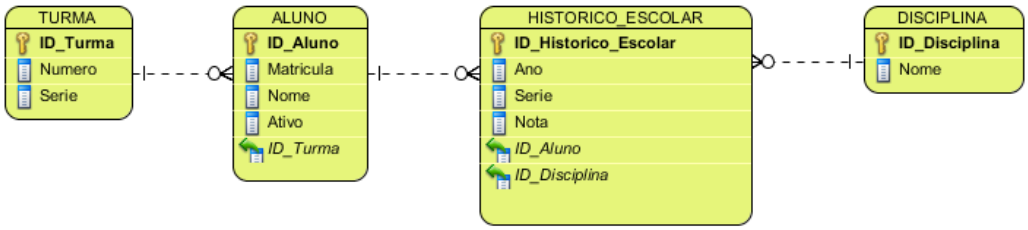
UNIDADE 3 – SQL
MÓDULO 1 – INSTRUÇÃO SELECT (PARTE 3)
01

1 - MODELO E DADOS HIPOTÉTICOS USADOS NOS EXEMPLOS

Olá, seja bem-vindo a mais um módulo de estudos. Neste módulo, continuaremos a tratar sobre os comandos SQL.

Conforme explicamos na unidade anterior, usaremos a estrutura e o conteúdo apresentados na maioria dos nossos exemplos.



Modelo do banco de dados da escola que utilizaremos para exemplificar este módulo.

ID_Turma	Numero	Serie
1	5-1	5a
2	5-2	5a
3	5-3	5a
4	6-1	6a
5	6-2	6a
6	7-1	7a

Registros atuais de turmas e séries

ID_Aluno	Matricula	Nome	Ativo	ID_Turma
1	14562/2	Thiago Ferreira Borges	1	1
2	432/2	Isadora Luccas Fernandes	1	6
3	332/5	Marcelo Correia Luz	0	0
4	4539/1	Mariana Gonçalves Coelho	1	4

Registros dos alunos e ex-alunos

02

Os registros que estão armazenados na tabela de histórico escolar representam os seguintes valores:

HISTÓRICO ESCOLAR				
Matrícula	**Disciplina**	**Ano**	**Série**	**Nota**
14562/2	Matemática	2015	5ª	6,8
14562/2	Português	2015	5ª	8,9
14562/2	Ciências	2015	5ª	10,0
432/2	Matemática	2015	7ª	8,2
432/2	Português	2015	7ª	8,9
432/2	Geografia	2015	7ª	9,4
432/2	História	2015	7ª	9,0
332/5	Matemática	2010	5ª	8,8
332/5	Português	2010	5ª	8,9
332/5	Ciências	2010	5ª	6,4

Nosso modelo de dados incorpora identificadores (chaves estrangeiras) que armazenam os dados desta maneira:

ID_Historico_Escolar	Ano	Serie	Nota	ID_Aluno	ID_Disciplina
1	2015	5a	6.0	1	1
2	2015	5a	8.9	1	2
3	2015	5a	10.0	1	3
4	2015	7a	8.2	2	1
5	2015	7a	8.9	2	2
6	2015	7a	9.4	2	4
7	2015	7a	9.0	2	5
8	2015	5a	8.8	3	1
9	2015	5a	8.9	3	2
10	2015	5a	6.4	3	3

Registros do histórico escolar de todos os alunos.

03

2 - Instrução Select – instrução

Vimos no módulo passado várias configurações e funcionalidades possíveis para criar uma instrução SELECT. Neste módulo concluiremos nosso estudo sobre a instrução SELECT.

2.1 - Concatenando dois campos

Uma operação muito comum é precisar juntar dois ou mais campos como um único registro. Isso é útil, por exemplo, em sistemas que armazenam o nome e o sobrenome da pessoa em campos diferentes. Quando quisermos uma consulta que apresente os dois campos juntos, precisaremos concatenar (juntar), o primeiro nome, um espaço e branco e o sobrenome da pessoa. Para tal, usamos o operador de adição (+) e damos um apelido para o resultado (operador AS). Veja o exemplo:

C1 - Juntar o nome e o sobrenome de uma pessoa (tabela PESSOA, campos Nome e Sobrenome):

```
SELECT Nome + " " + Sobrenome AS NomeCompleto FROM PESSOA.
```

Note que adicionamos um espaço em branco para que Nome e Sobrenome não fiquem unidos (Ex.: JoséSantos de Barro). Essa mesma técnica pode ser utilizada para acrescentar qualquer tipo de informação ao resultado. Por exemplo, poderíamos acrescentar o prefixo "Aluno: "ao nome de todos os alunos escrevendo algo como SELECT "`Aluno: ` + `Nome AS NomeAluno FROM ALUNOS`.

A SQL também oferece a função CONCAT() com mesmo resultado que o operador de adição. Alguns SGBDs só aceitam a função CONCAT(). A instrução C1 com a função CONCAT() seria escrita desta forma:

```
SELECT CONCAT (Nome, " ", Sobrenome) AS NomeCompleto FROM PESSOA.
```

04

2.2 - Operações matemáticas

A SQL também realiza operações matemáticas. Para tal usamos os operadores "+, -, * e /" para representar a adição, subtração, multiplicação e divisão. Por exemplo, suponha uma tabela de nome PRODUTOS que tenha os campos NomeProduto, QuantidadeEstoque e ValorUnitario.

C2 - Para que o sistema mostre o produto da quantidade em estoque pelo valor unitário, teríamos a seguinte consulta:

```
SELECT NomeProduto, QuantidadeEstoque * ValorUnitario AS ValorTotal
FROM PRODUTOS
```

Você também pode usar o processamento do SGBD para realizar contas matemáticas (que obviamente não realiza acesso a nenhuma tabela). Por exemplo, você pode realizar a operação 7 x 8, da seguinte forma: `SELECT 7*8`. O SGBD irá retornar a resposta da operação.

Você poderá usar diversos operadores matemáticos e parênteses para definir uma conta mais complexa.

05

2.3 - Agrupando resultados para operações estatísticas

Já aprendemos que o operador DISTINCT permite agrupar valores de uma relação de forma a não repetir registros. Entretanto a SQL também oferece um operador, denominado **GROUP BY**, que além de criar grupos como o DISTINCT, permite realizar uma série de operações, como contagem de registros, apresentar valores mínimos, máximos, médias, desvio padrão e outros.

O operador GROUP BY tem duas configurações:

- a primeira, que vem descrita na cláusula SELECT, apresenta a operação a ser realizada;
- a segunda, que vem descrita logo após o operador GROUP BY, determina qual(ais) campo(s) você deseja agregar para realizar a função especificada em SELECT.

Dessa forma, o padrão para se escrever uma cláusula com GROUP BY é:

```
SELECT <campo agregado>, OPERADOR_DESEJADO (campo computado)
FROM tabela
GROUP BY <campo_agregado>.
```

A cláusula GROUP BY não é compatível com o filtro de registros realizado pelo operador WHERE. No caso do GROUP BY devemos utilizar o operador HAVING que tem o mesmo efeito e forma de construção da cláusula WHERE. Veremos exemplos da cláusula HAVING logo a seguir.

Os tipos de operadores matemáticos que veremos a seguir apresentarão exemplos que utilizam essa forma básica de construção da cláusula GROUP BY.

06

2.3.1 - Contagem de registros

A contagem de registros pode ser feita por meio do operador COUNT(). Ele tanto pode ser aplicado em uma tabela sem o uso do agregador GROUP BY, como com o uso deste. Vemos ver exemplos disso:

C3 - Apresentar a quantidade de alunos cadastrada na nossa base de dados:

```
SELECT COUNT(*) FROM ALUNOS.
```

O banco de dados irá retornar apenas um número, que representa a quantidade de itens cadastrados na base de dados de alunos. Vamos supor que hipoteticamente o SGBD retornou dizendo que há 100 alunos cadastrados.

Nosso próximo exemplo irá criar subconjuntos para representar os valores agrupados pelos campos "Ativo", para saber quantos dos 100 alunos estão ativos e quantos estão inativos.

C4 - Apresentar a quantidade de alunos cadastrada na nossa base de dados, agrupado pelo campo "Ativo":

```
SELECT Ativo, COUNT(*) FROM ALUNOS GROUP BY (Ativo).
```

Para cada possibilidade cadastrada em "Ativo" (que no nosso caso só pode ser SIM ou NÃO), o banco de dados irá separar os itens cadastrados nesses conjuntos e contar quantos registros faz parte de cada conjunto. Uma resposta hipotética poderia ser informar que há 30 alunos ativos (SIM – 30) e 70 alunos inativos (NÃO – 70).

07

Vamos lançar mais um exemplo hipotético para reforçar essa funcionalidade. Suponha que exista uma tabela de pessoas com a seguinte estrutura: `PESSOAS (ID, Nome, EstadoCivil, Idade)`. E que você queira saber quantas pessoas existem no banco de dados de cada um dos estados civis possíveis. Vamos ver como seria nossa consulta:

C5 - Apresentar os subtotais de pessoas cadastradas para cada estado civil:

```
SELECT EstadoCivil, COUNT(*) FROM PESSOAS GROUP BY (EstadoCivil).
```

Um resultado factível para essa operação seria algo como o que vemos abaixo:

EstadoCivil	Count(*)
Solteiro	192
Casado	382
Viúvo	2
Separado	18
Null	8

Resultado da consulta C25

Perceba que as pessoas que não têm um estado civil cadastrado foram incluídas no estado civil nulo.

08

Vamos agora aplicar um filtro na nossa consulta, vamos eliminar as pessoas com menos de 18 anos da nossa pesquisa.

C6 - Apresentar os subtotais de pessoas cadastradas para cada estado civil, com idade igual ou superior a 18 anos:

```
SELECT EstadoCivil, COUNT(*) FROM PESSOAS GROUP BY (EstadoCivil)
HAVING Idade >= 18.
```

Observe que o uso do HAVING realiza a filtragem dos registros de maneira similar à cláusula WHERE. Um resultado factível para essa operação seria algo como o que vemos abaixo:

EstadoCivil	Count(*)
Solteiro	131
Casado	382
Viúvo	2
Separado	18
Null	2

Resultado da consulta C20

09

2.4 - Somando campos de registros

De utilização similar ao operador COUNT(), a SQL oferece uma forma de somar registros ou operações matemáticas. Esse operador, denominado **SUM**, é muito útil para criar totais e subtotais. Ele permite que ao invés de contar, os valores dos registros (ou operações) sejam somados.

Vamos supor a seguinte estrutura de tabela para um sistema de um mini mercado: `PRODUTOS (NomeProduto, Marca, QuantidadeEstoque, ValorUnitario)`. Suponha que o dono da empresa queira saber quantos itens existem em todo o seu estoque. Isso seria representado pela soma dos campos `QuantidadeEstoque` de todos os registros cadastrados na base de dados. Vamos ver como construir essa consulta usando o operador SUM:

C7 - Somar os itens da uma tabela de produtos, representado pelo campo de quantidade em estoque:

```
SELECT SUM(QuantidadeEstoque) FROM PRODUTOS
```

O SGBD iria processar essa consulta da seguinte forma:

1. Quais são os valores dos campos QuantidadeEstoque de todos os registros cadastrados em PRODUTOS?

2. Agora, qual é a soma de todos esses valores?

10

Vamos ampliar um pouco nosso exemplo, usando uma fórmula ao invés de um campo. Suponha que agora queiramos saber qual é o valor monetário de todos os produtos em estoque. Esse valor monetário seria representado pelo somatório do valor individual de todos os produtos multiplicado pela quantidade em estoque de cada um deles. Vamos ver como fica nossa consulta?

C8 - Para a mesma tabela PRODUTOS do exemplo anterior, qual é a soma do valor total de todos os produtos?

```
SELECT SUM(QuantidadeEstoque * ValorUnitario) AS ValorTotal FROM
PRODUTOS
```

Vamos agora combinar os operadores SUM e GROUP BY.

C9 - Para a mesma consulta do exemplo anterior, qual é o valor individual de cada produto em estoque?

```
SELECT Nome, SUM(QuantidadeEstoque * ValorUnitario) AS ValorTotal
FROM PRODUTOS
GROUP BY Nome
```

Note que inserimos o nome dos produtos da consulta, pois queremos subtotais por produto.

11

Se utilizarmos outros campos, o SGBD tentará grupar os produtos por esses campos. Para exemplificar isso, vamos agrupar os produtos por marca agora, criando subtotais não mais pelo nome do produto, mas sim pela marca.

C10 - Para a mesma consulta do exemplo C28, qual é o valor dos produtos em estoque, agrupados pelas respectivas marcas?

```
SELECT Marca, SUM(QuantidadeEstoque * ValorUnitario) AS ValorTotal
FROM PRODUTOS
GROUP BY Marca
```

Note que sempre que usamos um campo em GROUP BY, ele também aparece na cláusula SELECT.

12

2.5 - Valores máximos, mínimos, médios e outros

Da mesma forma que utilizamos o operador COUNT() e SUM(), a SQL oferece inúmeros outros operadores. Vamos ver os mais comuns:

MAX(<<campo>>)	Apresenta o valor máximo de uma determinada coluna ou de uma relação.
MIN(<<campo>>)	Apresenta o valor mínimo de uma determinada coluna ou de uma relação.
AVG(<<campo>>)	Apresenta o valor médios de uma determinada coluna ou de uma relação.
STD(<<campo>>)	Apresenta o desvio padrão de um conjunto de dados.
VARIANCE(<<campo>>)	Apresenta a variância de um conjunto de dados.

Veja a seguir alguns exemplos da utilização desses operadores.

13

Exemplos:

C11 - Quais são a maior, a menor e a média das notas presentes em histórico escolar?

```
SELECT MAX(Nota) AS Maior, MIN(Nota) AS Menor, AVG(Nota) AS Media FROM
HISTORICO_ESCOLAR
```

Maior	Menor	Media
10.0	6.0	8.45000

Resultado da consulta C30.

C12 - Quais são a maior, a menor e a média das notas presentes em histórico escolar, agrupadas por séries e ordenadas por série também?

```
SELECT Serie, MAX(Nota) AS Maior, MIN(Nota) AS Menor, AVG(Nota) AS
Media
FROM HISTORICO_ESCOLAR
GROUP BY Serie
ORDER BY Serie
```

Serie	Maior	Menor	Media
5a	10.0	6.0	8.16667
7a	9.4	8.2	8.87500

Resultado da consulta C31

2.6 - Usando os operadores de > e < para filtrar texto

Já aprendemos a lógica matemática que os operadores > e < servem para extrair subconjuntos de um total. Quando usamos ambos os operadores em uma consulta, eles oferecem a possiblidade de definir limites inferiores e superiores do subconjunto. Exemplo, a cláusula WHERE idade > 17 AND idade < 22 irá selecionas os registros de idade entre 18 e 21 anos. O mesmo poderia ser feito com os operadores >= (maior ou igual) e <= (menor ou igual) usando a cláusula WHERE idade >= 18 AND idade <=22.

Isso também é válido para datas, para poder selecionar todos os registros do ano de 2015 poderíamos usar a cláusula WHERE data >= '01-01-2015' AND data <'01-01-2016'. Note que para o limite superior valeria até o último segundo de 31/12/2015 23:59:59.

Pois bem, a novidade aqui é utilizar os operadores >, <, >= e <= com texto. A SQL interpreta que como os textos podem ser ordenados alfabeticamente, a letra F, por exemplo, é MENOR que a letra M. Segundo a mesma lógica, Marcelo é menor que Pedro, Azul é menor que Branco. Portanto, é possível usar os operadores citados para separar trechos de registros de um conjunto. Vamos a um exemplo para ficar mais claro.

C13 - Quais são os alunos cujos nomes começam com a letra 'M'?

```
SELECT Nome FROM ALUNO WHERE nome > "M" AND nome < "N"
```

Nome
Marcelo Correia Luz
Mariana Gonçalves Coelho

Resultado da consulta C32

C14 - Quais são os alunos cujo nomes estão entre 'M' e 'Z', ordenados por nome?

```
SELECT * FROM ALUNO WHERE nome >= "M" ORDER BY nome
```

Nome
Marcelo Correia Luz
Mariana Gonçalves Coelho
Thiago Ferreira Borges

Resultado da consulta C33

Note que como Z é a última letra, não é necessário usar um limite superior.

15

2.7 - Usando o operador IN

O operador IN permite você especificar um conjunto de resultados possíveis para que uma consulta resulte em um valor para a cláusula WHERE da pesquisa. Os valores são colocados entre parêntesis, após o nome do operador IN, separando os possíveis valores por vírgulas.

Esse operador é extremamente poderoso, vejamos exemplos.

C15 - Quais são nomes dos alunos que estão na 5ª, na 6ª, e na 7ª série?

Usando o operador igual (=), podemos construir essa consulta assim:

```
SELECT Nome FROM Aluno WHERE Serie = "5ª" OR Serie = "6ª" OR Serie = "7ª"
```

Essa mesma consulta utilizando o operador IN seria escrita da seguinte forma:

```
SELECT Nome FROM Aluno WHERE Serie IN ("5ª", "6ª", "7ª")
```

Veja como é mais simples e mais facilmente analisada.

16

Vamos agora entender o poder desse operador. Nós podemos substituir o conteúdo que está enumerado entre parêntesis por uma cláusula SELECT que traga apenas um campo de valores válidos para a pesquisa. Isso permite que duas cláusulas SELECTs sejam feitas ao mesmo tempo, uma que identifica valores pesquisáveis e a outra que realiza a operação solicitada. Vamos ver um exemplo para ficar mais fácil entender essa questão:

C16 - Quais são os nomes dos alunos que tiveram nota acima de 7 no ano de 2015.

Usando o operador JOIN, podemos escrever essa consulta assim:

```
SELECT A.Nome
FROM ALUNO AS A INNER JOIN HISTORICO_ESCOLAR AS H
ON A.ID_Aluno = H.ID_Aluno
WHERE H.Nota > 7 AND H.Ano = 2015.
```

Usando o operador IN, poderemos escrever a consulta que reflita “quais os nomes dos alunos, cujo ID_Aluno esteja em (Quais os ID_Alunos cujas notas são maiores que 7 e o ano igual a 2015)”. Isso seria escrito em SQL desse modo:

```
SELECT Nome
FROM ALUNO
WHERE ID_Aluno IN (
               SELECT ID_Aluno
               FROM HISTORICO_ESCOLAR
               WHERE Nota > 7 AND Ano = 2015)
```

Observe como a cláusula SELECT que está entre parêntesis retornará os ID_Aluno daqueles que satisfazem a condição imposta. Esses ID_Aluno servirão de chave para a outra consulta SELECT pesquisar o nome dos alunos.

17

2.8 - Operador NOT

O operador NOT nega uma cláusula invertendo o resultado pesquisado. Essa inversão refere-se ao conjunto dos itens excluído da relação afirmativa. Ele pode ser usado em cláusulas WHERE, IN e LIKE. Vamos ver alguns exemplos.

C17 - Quais são as notas dos alunos que não são nem da 5ª nem da 6ª série?

```
SELECT Notas FROM HISTORICO_ESCOLAR WHERE NOTA NOT IN (“5ª”, “6ª”)
```

C18 - Quais são os nomes dos alunos que não contenham “Marcelo” no nome?

```
SELECT Nome FROM ALUNO WHERE Nome NOT LIKE “%Marcelo%”
```

Lembre-se: para o operador WHERE com símbolo de igual não utilizamos “NOT =”, mas sim “<>” (que significa diferente).

18

RESUMO

Neste módulo, aprendemos que:

a) Podemos usar o operador de adição para concatenar dois ou mais campos. Também é possível usar a função CONCAT(campo1, campo2, ..., campo n) para realizar o mesmo procedimento.

b) Os símbolos +, -, / e * promovem as quatro operações matemáticas básicas.

c) O operador GROUP BY permite agrupar resultados, realizando operações sobre conjuntos como contagem, soma, valores máximos, mínimos e médias.

d) O operador IN permite criar uma lista de possíveis valores que atendam a uma cláusula WHERE. Esses valores podem ser substituídos por outra cláusula SELECT que retorna itens a serem pesquisáveis.

e) O operador NOT serve para negar expressões IN e LIKE, de forma a obter o resultado inverso do conjunto pesquisado.

UNIDADE 3 – SQL
MÓDULO 2 – INSTRUÇÕES INSERT, UPDATE E DELETE

01

1 - MODELO E DADOS HIPOTÉTICOS USADOS NA MAIORIA DOS EXEMPLOS

Olá, seja bem-vindo a mais uma etapa do nosso estudo. A instrução SELECT é tão complexa que precisamos de vários módulos para falar dela. Essa instrução não afeta o conteúdo do banco de dados, visto que ela só extrai informações. Neste módulo, trataremos das instruções que podem ser usadas para modificar o conteúdo do banco de dados: INSERT, UPDATE e DELETE. Também usaremos cláusulas SELECT com esses operadores para fazer operações baseadas em consultas.

Devemos tomar muito cuidado ao executar esses comandos, especialmente em bancos de dados de produção, visto que podemos danificar todo um conjunto de informações, e, eventualmente, precisaremos de backups para restaurar o que fizemos de errado!

Enquanto o operador SELECT pode trabalhar com várias tabelas ao mesmo tempo (usando, por exemplo, o operador JOIN), as operações de INSERT, UPDATE e DELETE só atuam com uma tabela por vez. Entretanto, quando há regras de operações em cascata, comandos de UPDATE e DELETE podem afetar mais de uma tabela.

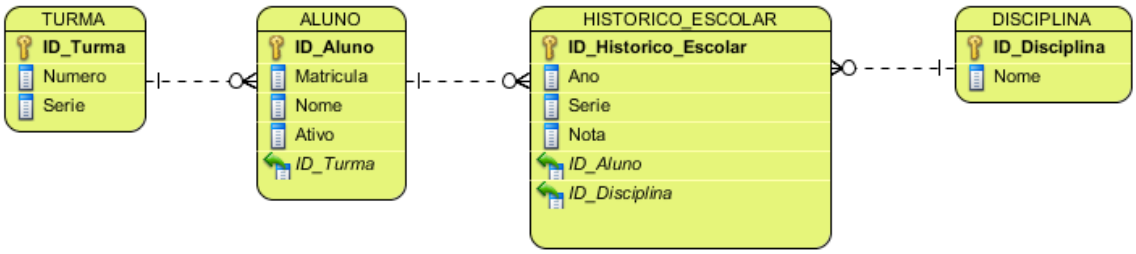
As consultas SELECT não precisam ser controladas por transações, pois elas não alteram os bancos de dados. Já as operações INSERT, UPDATE e DELETE podem ser controladas por transações para que seus resultados sejam confirmados antes da confirmação da operação (COMMIT).

Lembrando que já estudamos antes como controlar transações:

- Começamos pelo operador BEGIN TRANS (ou BEGIN TRANSACTION);
- Realizamos todas as operações de manipulação que desejamos (INSERT, UPDATE e/ou DELETE);
- Testamos os resultados (geralmente realizando comandos SELECT);
- Finalizamos a operação confirmando-a (com o COMMIT) ou cancelando-a (com o ROLL BACK).

02

Conforme já explicamos, usaremos a estrutura e o conteúdo abaixo na maioria dos nossos exemplos.



Modelo do banco de dados da escola que utilizaremos para exemplificar este módulo

ID_Turma	Numero	Serie
1	5-1	5a
2	5-2	5a
3	5-3	5a
4	6-1	6a
5	6-2	6a
6	7-1	7a

Registros atuais de turmas e séries

ID_Aluno	Matricula	Nome	Ativo	ID_Turma
1	14562/2	Thiago Ferreira Borges	1	1
2	432/2	Isadora Luccas Fernandes	1	6
3	332/5	Marcelo Correia Luz	0	0
4	4539/1	Mariana Gonçalves Coelho	1	4

Registros dos alunos e ex-alunos

03

Os registros que estão armazenados na tabela de histórico escolar representam os seguintes valores:

HISTÓRICO ESCOLAR				
Matrícula	**Disciplina**	**Ano**	**Série**	**Nota**
14562/2	Matemática	2015	5ª	6,8
14562/2	Português	2015	5ª	8,9
14562/2	Ciências	2015	5ª	10,0
432/2	Matemática	2015	7ª	8,2
432/2	Português	2015	7ª	8,9
432/2	Geografia	2015	7ª	9,4
432/2	História	2015	7ª	9,0
332/5	Matemática	2010	5ª	8,8
332/5	Português	2010	5ª	8,9
332/5	Ciências	2010	5ª	6,4

Nosso modelo de dados incorpora identificadores (chaves estrangeiras) que armazenam os dados desta maneira:

ID_Historico_Escolar	Ano	Serie	Nota	ID_Aluno	ID_Disciplina
1	2015	5a	6.0	1	1
2	2015	5a	8.9	1	2
3	2015	5a	10.0	1	3
4	2015	7a	8.2	2	1
5	2015	7a	8.9	2	2
6	2015	7a	9.4	2	4
7	2015	7a	9.0	2	5
8	2015	5a	8.8	3	1
9	2015	5a	8.9	3	2
10	2015	5a	6.4	3	3

Registros do histórico escolar de todos os alunos

04

2 - O COMANDO INSERT

O comando INSERT serve para cadastrar novas informações em uma tabela.

Ou seja, ele acrescenta um ou mais registros a uma relação. Temos de especificar o nome da relação e uma lista de valores para a(s) tupla(s). A forma padrão do comando INSERT é:

```
INSERT INTO tabela (atributo1, atributo2, ..., atributoN) VALUES
(valor1, valor2, ..., valorN)
```

Há duas formas de se escrever um comando INSERT. A primeira delas é especificando os atributos e os valores, da mesma forma que observamos no modelo acima. Por exemplo, para acrescentar uma nova tupla à relação TURMA:

- **Forma a**:

```
INSERT INTO TURMA (ID_Turma, Numero, Serie) VALUES (7, '7-2', '7a')
```

Usando esta forma completa de instrução, não é obrigatório que a lista de atributos e a lista de valores estejam exatamente na ordem dos campos na instrução CREATE TABLE da tabela em questão. Poderíamos, por exemplo, mudar a ordem dos campos e dos valores, porém mantendo a mesma relação entre os nomes dos atributos e os valores. Por exemplo, a instrução da "Forma a" poderia ser reescrita desta maneira, obtendo o mesmo resultado: `INSERT INTO TURMA (Serie, ID_Turma, Numero) VALUES ('7a', 7, '7-2')`.

A segunda forma é não escrevendo os nomes dos atributos da tabela (campos) e especificando apenas os valores na mesma ordem em que os atributos correspondentes foram especificados no comando CREATE TABLE. Por exemplo, desta maneira:

- **Forma b**:

```
INSERT INTO TURMA VALUES (7, '7-2', '7a')
```

05

2.1 - Valores padrão, nulos e autoincremento

Caso uma tabela possua atributos com valores padrões ou que admitam valores nulos, estes podem ser omitidos da cláusula INSERT. Nesse caso, o SGBD inserirá os valores padrão ou nulo automaticamente. Exemplo: Suponha que a tabela de ALUNO possua o valor padrão 'T' (true = verdadeiro) para o campo ATIVO. Se não informarmos um valor para o campo ATIVO ao inserir um novo aluno nessa tabela, o SGBD automaticamente cadastrará 'T' para esse campo.

Muitos modelos de banco de dados criam tabelas que utilizam chaves primárias com valor autoincrementável. Nesse caso, não é possível informar o valor desejável na cláusula INSERT. Deve-se, obrigatoriamente, omitir esse valor e o SGBD automaticamente irá inserir o valor apropriado.

Exemplo: suponha que o valor ID_Turma em TURMA seja um inteiro autoincrementável. A instrução INSERT que representa o exemplo “Forma a” citada anteriormente, poderia ser escrita satisfatoriamente dessa maneira: `INSERT INTO TURMA (Numero, Serie) VALUES ('7-2', '7a').`

Tenha atenção especial em como descrever os valores dos atributos: textos, datas e valores especiais (como bytes) devem vir entre aspas. Apenas números e booleano (True/False) podem vir sem aspas. Se você colocar números ou booleano entre aspas é bem provável que o SGBD não gere erro, aceitando seu comando.

06

2.2 - Restrições de integridade

Ao realizar uma operação de INSERT, o SGBD sempre irá realizar todos os controles de restrição de integridade para cada campo inserido.

Essas restrições incluem:

- Validação de tipos de dados.

Ou seja, se o valor a ser inserido corresponde ao tipo de campo da tabela. Exemplo: Se você tentar inserir o valor “Marcelo” em campo do tipo numérico, o SGBD gerará um erro.

- Validação de dados válidos.

Ou seja, se o valor informado corresponde a um valor válido para o campo. Exemplo: Se você tentar inserir o valor “2015/02/30” em campo do tipo data, o SGBD gerará um erro, pois não existe dia “30 de fevereiro”.

- Validação de limites.

Ou seja, se o valor a ser inserido corresponde aos limites inferior ou superior do campo. Exemplos: Se você tentar inserir o valor “Marcelo” em campo CHAR(6), que só aceita seis caracteres, o SGBD gerará um erro. Da mesma forma, se você tentar cadastrar o valor 1.912 em um campo numérico do tipo TYNEINT, o SGBD gerará um erro, visto que o valor máximo para TYNEINT é 254.

- Validação de chaves.

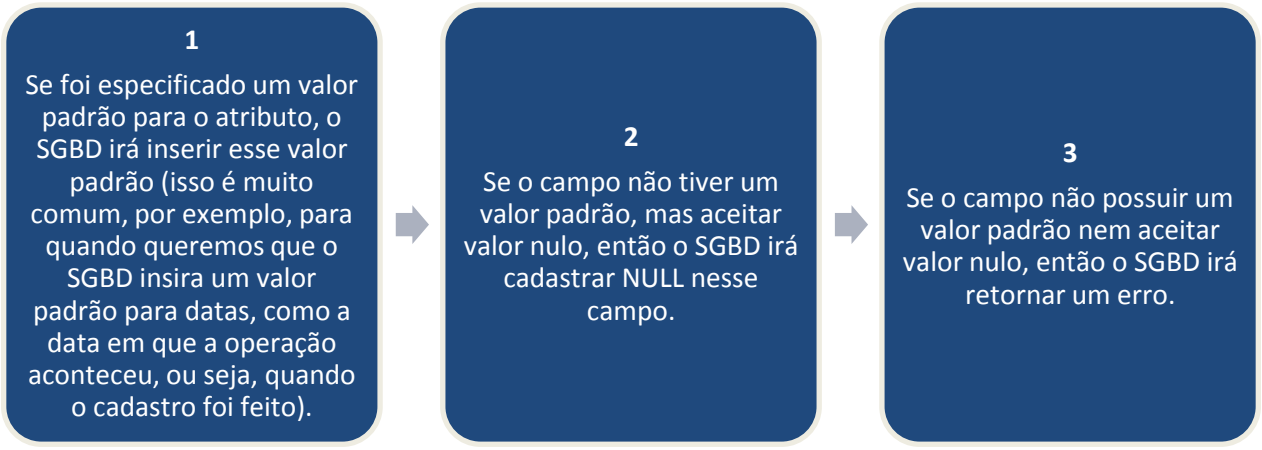
Ou seja, se o valor a ser inserido é uma chave estrangeira, o SGBD irá validar se essa chave estrangeira existe na tabela de origem. Do contrário, o SGBD gerará um erro.

- Validação de valores não informados. Veja a seguir como o SGBD trata valores não informados.

07

2.3 - Valores não informados

Sempre que você não especificar um valor para um campo em uma instrução INSERT, o SGBD irá se comportar da seguinte maneira:



08

2.3 - Inserindo vários valores ao mesmo tempo em uma única instrução INSERT

É possível inserir em uma tabela múltiplos registros separados por vírgulas em um único comando INSERT. Os valores de atributo que formam cada registro ficam entre parênteses e separados por vírgula.

O exemplo a seguir insere três registros na tabela de TURMA:

```
INSERT INTO TURMA VALUES (7, '7-2', '7a'), (8, '7-3', '7a'), (9, '8-1', '8a')
```

Normalmente um SGBD aceitaria, sem problemas, milhares de registros sendo inseridos ao mesmo tempo utilizando apenas uma instrução INSERT.

09

2.4 - Inserindo itens a partir de uma consulta SELECT

Uma variação do comando INSERT inclui a opção de utilizar uma cláusula SELECT para identificar os registros a serem incluídos. Isso é feito pela substituição dos valores dos atributos da instrução por um comando SELECT que retorne exatamente os campos de devem ser inseridos. Dessa forma, também é possível inserir um ou vários registros em uma tabela. O padrão para uma instrução INSERT baseado em um comando SELECT é o seguinte:

```
INSERT INTO tabela (atributo1, atributo2, ..., atributoN) VALUES
(SELECT valor1, valor2, ..., valor FROM tabela WHERE ...)
```

A cláusula SELECT da instrução INSERT pode conter qualquer um dos atributos e relacionamentos que estudamos anteriormente.

Vamos a um exemplo. Suponha que exista uma tabela de nome Historico_Escolar_2015 que deva conter apenas os registros de histórico escolar do ano de 2015, suponha ainda que essa tabela possui a estrutura (ID_Historico_Escolar, Serie, Nome, ID_Aluno, ID_Disciplina), ou seja, a mesma estrutura da tabela Historico_Escolar, exceto o campo "Ano".

Uma possível instrução que selecionaria todos os registros de 2015 da tabela de Historico_Escolar e inserisse na tabela de Historico_Escolar_2015 seria algo como:

```
INSERT INTO Historico_Escolar_2015 (ID_Historico_Escolar, Serie, Nome,
ID_Aluno, ID_Disciplina) VALUES (SELECT ID_Historico_Escolar, Serie,
Nome, ID_Aluno, ID_Disciplina FROM Historico_Escolar WHERE Ano =2015)
```

Perceba que, como as estruturas das tabelas não são idênticas, foi necessário enumerar cada um dos campos da operação.

10

3 - O COMANDO DELETE

O comando DELETE remove registros de uma tabela. Ele inclui uma cláusula WHERE, semelhante a que é usada em uma consulta SQL, para selecionar os registros a serem excluídos.

Os registros são explicitamente excluídos de apenas uma tabela por vez. No entanto, a exclusão pode se propagar para registros de outras tabelas, caso as ações de propagação em cascata (criadas por meio de Trigger/Gatilhos) forem especificadas nas restrições de integridade referencial da tabela.

Dependendo do número de registros selecionados pela condição na cláusula WHERE, zero, um, vários ou todos os registros de uma tabela podem ser excluídos por um único comando DELETE. Uma cláusula WHERE inexistente especifica que todos os registros na relação deverão ser excluídas; porém, a tabela permanece no banco de dados como uma tabela vazia. O formato de uma instrução DELETE é:

```
DELETE FROM Tabela WHERE Condições
```

Vamos a alguns exemplos: excluindo todos os registros da tabela de alunos:

```
DELETE FROM ALUNOS
```

A inexistência de uma cláusula WHERE fará com que todos os registros da tabela sejam afetados, e consequentemente apagados. Entretanto, o comando DELETE irá falhar nas relações de restrições de integridade referencial, onde não seja possível excluir um item, caso a chave primária dele esteja cadastrada em uma outra tabela referenciada. Por exemplo, se esse modelo de restrição houvesse sido aplicado ao modelo da escola (nossos exemplos), não seria possível excluir nenhum aluno que tivesse informações de histórico escolar cadastradas.

Outro exemplo: excluindo da tabela alunos que contenham “Marcelo” como parte do nome:

```
DELETE FROM ALUNOS WHERE Nome Like “%Marcelo%”
```

Neste exemplo usamos uma cláusula WHERE com a instrução LIKE, exatamente como aprendemos quando estudamos o comando SELECT ... LIKE.

11

4 - O COMANDO TRUNCATE

O comando TRUNCATE é uma forma simplificada do comando DELETE. Ele faz com que todos os registros de uma tabela sejam excluídos.

Tem o mesmo resultado que “DELETE FROM Tabela”. O comando TRUNCATE tem o seguinte layout:
`TRUNCATE Tabela.`

Exemplo: para excluir todos os registros da tabela ALUNOS

```
TRUNCATE ALUNO
```

O comando TRUNCATE só será executado se ele não ferir nenhuma regra de integração para deleção em cascata.

12

5 - O COMANDO DROP TABLE

Ao excluir todos os registros de uma tabela, seja utilizando o comando DELETE, seja utilizando o comando TRUNCATE, a estrutura da tabela permanece no banco.

Para se eliminar uma estrutura de uma tabela (e consequentemente todos os registros ali cadastrados), usamos o comando DROP TABBLE.

Esse comando tem o seguinte layout: `DROP TABLE Tabela`.

Exemplo: para excluir a tabela Aluno (e os registros):

```
DROP TABLE ALUNO
```

O DROP TABLE também só será executado se não ferir nenhuma regra de deleção em cascata.

13

6 - O COMANDO UPDATE

O comando UPDATE é usado para modificar valores dos atributos de um ou mais registros selecionados.

Assim como no comando DELETE, uma cláusula WHERE no comando UPDATE seleciona os registros a serem modificados em uma única relação. No entanto, a atualização de uma chave primária pode ser propagada para os valores de chave estrangeira dos registros em outras relações se tal ação de disparo referencial for especificada nas restrições de integridade referencial da tabela.

Uma cláusula SET adicional no comando UPDATE especifica os atributos a serem modificados e seus novos valores. O layout do comando UPDATE é:

```
UPDADE Tabela SET Campo1 = NovoValor1, Campo2 = NovoValor2, ...,
CampoN = NovoValorN WHERE Condições.
```

Por exemplo, para alterar o nome do aluno de ID_Aluno 1, para Tiago Ferreira Borges, poderíamos utilizar a seguinte instrução:

```
UPDATE ALUNO SET Nome = "Tiago Ferreira Borges" WHERE ID_Aluno = 1
```

Outra forma de realizar essa mesma operação é utilizar o nome atual para localização do registro, construindo a seguinte instrução:

```
UPDATE ALUNO SET Nome = "Tiago Ferreira Borges" WHERE Nome = "Thiago
Ferreira Borges"
```

Essas duas instruções só afetam um único registro. Vamos ver um exemplo de uma instrução UPDATE que afete vários registros. Por exemplo, alterar a tabela de Historico_Escolar, alterando todos os registros do ano 2015, mudando o ano para 2016.

```
UPDATE Historico_Escolar SET Ano = 2016 WHERE Ano = 2015
```

14

Outro exemplo um pouco mais complexo: suponha que desejássemos alterar para “8,9” a nota no histórico escolar do aluno de matrícula “332/5”, da matéria de Ciências, do ano de 2010, da 5ª série. Note que a tabela de histórico escolar não possui chave primária para identificação do registro. Dessa forma precisamos construir uma cláusula WHERE com todos os campos que identifiquem e selecionem apenas esse registro.

Veja como seria a instrução:

```
UPDATE HISTORICO_ESCOLAR
SET Nota = 8.9
WHERE Ano = 2010
AND Serie = “5a”
AND ID_Aluno = (SELECT ID_Aluno FROM ALUNO WHERE Matricula = “332/5”)
AND Disciplina = (SELECT ID_Disciplina FROM DISCIPLINA Where Nome =
“Ciências”)
```

Observe que como não sabemos de antemão os valores de ID_Aluno e de ID_Disciplina, precisamos utilizar instruções SELECT para identificar esses registros.

Vamos pensar em outro cenário agora. Suponha que exista uma tabela denominada PRODUTOS, que possua um campo de nome ValorDeVenda. Vamos construir uma instrução UPDATE que acrescente 10% ao valor de venda de todos os produtos:

```
UPDATE PRODUTOS, SET ValorDeVenda = ValorDeVenda * 1.1
```

Note agora que o novo valor do campo Valor de venda será o valor de venda atual multiplicado por 1,1, ou seja, 10% de acréscimo.

Também podemos utilizar os atributos NULL ou DEFAULT como valores para um campo. Vamos ver um exemplo com os dois atributos:

Para uma tabela de produtos, alterar a marca de todos os produtos para o valor padrão do modelo de dados e o atributo de preço de compra para o valor nulo:

```
UPDATE Produtos SET Marca = DEFAULT, PrecoCompra = NULL
```

RESUMO

Neste módulo, aprendemos que:

a) Os operadores INSERT, UPDATE e DELETE alteram o conteúdo do banco de dados, e portanto seu uso deve ser muito cauteloso. Operações erroneamente feitas em bancos de dados, principalmente os de produção, precisarão de backups para restaurar o conteúdo.

b) Consultas SELECT podem participar em conjunto com operações de manipulação de dados, na medida em que as consultas SELECT podem prover informações que serão utilizadas como filtros nas operações de alteração e deleção, ou como dados para cadastro e modificação, nas operações de inserção e alteração de dados.

c) O controle transacional pode auxiliar os usuários na medida em que este controle permite validar os resultados ou desfazer as ações erroneamente executadas.

d) O comando INSERT permite cadastrar (inserir) novas informações no banco de dados.

e) Caso algum campo não seja informado na instrução INSERT, o SGBD irá verificar se o campo possui um valor padrão ou se aceita nulo como dado válido. Se nenhuma dessas opções foi viável, o SGBD irá gerar um erro.

f) O SGBD sempre executa controles de restrição de integridade nas operações de manipulação de dados.

g) O comando INSERT permite cadastrar vários conjuntos de registros (tuplas) ao mesmo tempo, bastando para isso separar os conjuntos de registros por vírgulas.

h) Podemos substituir o conjunto de registros da cláusula VALUES por uma instrução SELECT, dessa forma, os registros a serem inseridos serão extraídos por meio de uma consulta SELECT.

i) O comando DELETE exclui um ou mais registros. Antes da exclusão, todos os testes de integridade deverão ser realizados com sucesso pelo SGBD.

j) O comando TRUNCATE é semelhante ao comando DELETE, ele elimina todos os registros de uma tabela e não admite a cláusula WHERE.

k) O comando DROP TABLE exclui uma tabela do banco de dados, eliminando não só os dados, mas a própria estrutura da tabela.

l) O comando UPDATE atualiza registros em uma tabela. Podemos usar cláusulas SELECT como parâmetros para inserção ou atualização de registros.

UNIDADE 3 – SQL

MÓDULO 3 – GATILHOS, VISÕES E MODIFICAÇÃO DE ESTRUTURA DE TABELAS

01

1 - GATILHOS

Olá, iniciaremos agora mais uma etapa do nosso estudo. Este módulo descreve recursos mais avançados da linguagem SQL. Aprenderemos o que são **gatilhos** (triggers) e **visões** (views). Também estudaremos como modificar um esquema.

Um gatilho, chamado de Trigger, em inglês, refere-se aos miniprogramas criados em SQL que podem ser executados antes ou depois de operações de seleção ou manipulação de dados. As triggers têm esse nome porque, assim como uma arma, elas "disparam" uma ação.

Normalmente, as triggers são criadas para que as configurações de integridade referencial possam existir.

Por exemplo: Suponha que exista uma tabela de nome PESSOA e outra tabela de nome DEPENDENTE. Na tabela de DEPENTENDE aparece a chave primária da tabela PESSOA para que seja possível identificar qual é a pessoa a quem aquele dependente depende. Algo como uma tabela de pais (PESSOA) e filhos (DEPENDENTE). Uma regra que poderia ser criada é "ao se excluir uma pessoa, todos os dependentes dessa pessoa também devem ser excluídos". Para criar essa regra no banco de dados é necessário criar uma trigger de deleção, onde, no mome nto que um comando de deleção de dados for executado, como por exemplo DELETE FROM PESSOA WHERE ID_Pessoa = 123, a trigger seria disparada e executaria um comando como DELETE FROM DEPENDENTE WHERE ID_Pessoa = 123.

As triggers são muito poderosas e podem implementar uma centena de regras de negócio. Embora isso não seja indicado como melhor opção em nível de projeto de sistema, é possível colocar as regras dentro de triggers. Entretanto, o ideal é deixar as regras de negócio da camada da aplicação, e no banco de dados, apenas a camada de persistência (armazenamento).

02

Vamos ver alguns exemplos factíveis de triggers que poderiam realizar **controles de regras de negócio**:

- Uma trigger poderia conter uma regra em que a data de cadastro dos subordinados só poderiam ser datas mais recentes do que a data de cadastro do chefe.
- Uma trigger poderia conter uma regra em que o salário de um subordinado não poderia ser maior que o salário do próprio chefe.
- Uma trigger poderia conter uma regra em que um determinado campo de data só aceitasse datas em dias úteis.
- Uma trigger poderia conter uma regra em que limitasse um número máximo de itens relacionados para uma determinada tabela.

- Uma trigger poderia conter uma regra de associação de endereços na qual só permitisse selecionar endereços de uma determinada cidade.
- Uma trigger poderia conter uma regra na qual informaria um gerente quando um funcionário realizasse um determinado volume de vendas.
- Uma trigger poderia conter uma regra na qual informaria um gerente quando um funcionário excedesse um determinado número de faltas no trabalho.
- Uma trigger poderia conter uma regra na qual sempre que uma consulta a uma tabela de dados sigilosos fosse realizada, também seria registrado o nome do usuário que realizou a consulta (para fins de auditoria de segurança).

As regras criadas são **ações** a serem executadas quando as **condições** forem verdadeiras para o SGBD. A condição, portanto, é usada para monitorar o banco de dados. Outras ações podem ser especificadas, como executar um procedimento armazenado (em inglês "stored procedure", que é outra espécie de um miniprograma feito em SQL) ou disparar outras atualizações. Discutiremos sobre trigger em detalhes em módulos futuros, no momento vamos apenas analisar uma estrutura simples.

03

O padrão para que criação de uma trigger é:

```
CREATE TRIGGER <<nome da trigger>> <<BEFORE ou AFTER>> <<SELECT OR
INSERT OR UPDATE OR DELETE>> OF <<lista dos campos>> ON <<nome da
tabela>> <<sequência de comandos>>
```

Vamos explicar a estrutura da criação de uma Trigger:

- <<Nome da Trigger>>

Uma trigger precisa de um nome. Normalmente esses nomes remetem à regra de negócio que a trigger impõe. Por exemplo: ViolacaoSalarial, ExcluiDependentes, PermiteDataDiaUtil.

- BEFORE ou AFTER

Define quando a trigger será executada, se antes (BEFORE) da ação dispara a trigger ou logo depois (AFTER) da ação ser executada. Você deve escolher apenas uma das opções.

- INSERT ou UPDATE ou DELETE

Define que tipo de operação SQL dispara a trigger. Caso você queira que a trigger seja disparada em mais de uma ação, use o operador OR para separar as ações (exemplo: INSERT OR UPDATE).

- <<Lista de campos>>

Define os campos que serão monitorados para o caso de a trigger ser do tipo UPDATE. Somente os campos listados aqui poderão disparar a trigger. Separe os nomes dos campos por vírgulas.

- <<Nome da tabela>>

Tabela a qual a trigger está relacionada. Só pode ser uma única tabela.

- <<sequência de comando>>

Refere-se ao miniprograma que realizará as ações propostas pela trigger.

04

Caso a sequência de comando contenha mais de um comando, o bloco de comandos deve vir separado pelos delimitadores BEGIN e END, seguindo o padrão a seguir:

```
BEGIN
        Comando 1
        Comando 2
        Comando N
END
```

Vamos ver e analisar um exemplo de uma trigger que tem por objetivo identificar casos em que o salário de um funcionário é maior que o salário do respectivo chefe e executando um procedimento armazenado que poderia, por exemplo, enviar um e-mail alertando alguém sobre essa questão.

Atenção: numeramos as linhas abaixo apenas para posteriormente descrever o que cada uma delas significa. Na prática, essa numeração não existe.

```
1. CREATE TRIGGER Violacao_Salarial
2. BEFORE INSERT OR UPDATE OF Salario, CPF_Supervisor
3. ON Funcionario
4. FOR EACH ROW
5. BEGIN
6. WHEN (NEW.Salario > (SELECT Salario FROM Funcionario WHERE CPF = NEW.CPF_Supervisor))
7. Informar_Supervisor (New.CPF_Supervisor, NEW.CPF)
8. END
```

CREATE TRIGGER Violacao_Salarial

1. Define que está sendo criada uma trigger de nome Violacao_Salarial. Logo, podemos concluir que essa trigger fará alguma ação quando uma regra de violação salarial acontecer.

BEFORE INSERT OR UPDATE OF Salario, CPF_Supervisor

2. Informa que a trigger será executada antes (before) dos comandos de inserção de registros (insert) ou antes da atualização (update) dos campos Salario e/ou CPF_Supervisor. São esses os campos monitorados, uma atualização em outros campos não executará a trigger.

ON Funcionario

3. Define o nome da tabela a ser monitorada.

FOR EACH ROW

4. Início do miniprograma que será executado, o caso o miniprograma diz que "para cada campo" (Salario ou CPF_Supervisor)...

BEGIN

5. Indica que um ou mais comandos em série participarão da Trigger. Esta linha indica o início dos comandos.

WHEN (NEW.Salario > (SELECT Salario FROM Funcionario WHERE CPF = NEW.CPF_Supervisor)

6. Continuação do miniprograma: "...quando o novo salário que está querendo ser atualizado for maior que o salário do funcionário cujo CPF for igual ao CPF do supervisor deste (ou seja, o salário do chefe dele)..."

Informar_Supervisor (New.CPF_Supervisor, NEW.CPF)

7. Última linha do miniprograma: "...deve-se executar o procedimento armazenado d nome 'Informar_Supervisor', com os parâmetros 'CPF do Supervisor' e 'novo CPF a ser inserido/atualizado'". Desta forma, provavelmente a storage procedure irá enviar um e-mail para o chefe da pessoa, informando que o salário dela é maior que a do próprio chefe.

END

8. Esta linha indica o término dos comandos da trigger, ela conclui o que foi iniciado na linha 5.

05

2 - Visões

Uma visão (View em inglês) é uma técnica utilizada para criar consultas SQL que ficam permanentemente disponibilizadas no banco de dados por meio do que parece ser uma tabela.

Funciona assim: muitos sistemas baseados em bancos de dados exigem diversas consultas e relatórios complexos. Por vezes, a complexidade para montar essas consultas é grande demais para que os programadores consigam produzir os resultados desejados. Nesse momento, os administradores de dados, que normalmente são especialistas em SQL, podem escrever as consultas desejadas e transformá-las em uma visão. Após a visão ser criada, os programadores só precisam consultar a visão da mesma forma que se consulta uma tabela, ou seja, uma cláusula do tipo SELECT * FROM <<Nome da Visão>>.

Uma visão é criada a partir de uma cláusula SELECT qualquer, e tem o seguinte *layout* de criação:

```
CREATE VIEW <<Nome da visão>> AS <<Cláusula SELECT>>
```

Usando o banco de dados hipotético da escola, um exemplo bem simples de uma view que mostrasse os alunos registrados para o ano de 2015 seria:

```
CREATE VIEW Alunos2015 AS SELECT * FROM Alunos WHERE ID_Aluno IN
(SELECT ID_Aluno FROM HistoricoEscolar WHERE Ano = 2015)
```

Criada a visão, para acessar esses registros, bastaria agora apenas uma consulta simples à visão:

```
SELECT * FROM Alunos2015
```

06

Podemos também usar parâmetros SELECT na View. Por exemplo, para a view anterior, caso desejássemos apresentar somente o nome dos alunos que contenham a palavra “Marcelo”, teríamos a seguinte consulta:

```
SELECT Nome FROM Alunos2015 WHERE Nome LIKE '%Marcelo%'
```

Percebeu como as coisas ficam mais simples? Vamos agora ver um exemplo real retirado de uma aplicação que existe de fato em uma empresa. Veja só a complexidade dessa consulta SQL:

```
SELECT t.tarefa_id ID, t.tarefa_nome Tarefa,
   CONCAT( FORMAT( t.tarefa_percentagem, 0 ) ,  '%' ) AS Perc,
   COALESCE( REPLACE( l.tarefa_log_data,  ' 00:00:00',  '' ) , '- Sem
registro -' ) AS  'Último Andamento',
   COALESCE( l.tarefa_log_descricao,  '- Sem registro -' ) AS
Andamento
FROM tarefas AS t
LEFT JOIN tarefa_log AS l
ON t.tarefa_id = l.tarefa_log_tarefa
WHERE t.tarefa_projeto =432
AND t.tarefa_nome LIKE  '%Atividade:%'
```

```
ORDER BY tarefa_id
LIMIT 0, 999999
```

07

Sem querer entrar no mérito do que faz essa consulta, podemos perceber que ela é altamente complexa. Para facilitar as coisas (e poder repetir o que essa consulta produz), poderíamos criar uma visão de nome “RelatorioDeAndamento”, onde uma consulta simples a essa visão, SELECT * FROM RelatorioDeAndamento”, produzirá o mesmo resultado da consulta SELECT anterior.

Uma visão se comporta exatamente como uma tabela, podendo, por exemplo, especificar campos, realizar filtros, agregações (JOIN) com outras tabelas ou visões, e outras operações. Entretanto, dependendo de como a visão é construída (o que acontece na maioria das vezes), não é possível realizar operações de INSERT, UPDATE ou DELETE com visões.

Lembre-se também que uma visão não é uma tabela física de fato, ela não possui uma cópia dos registros provenientes da pesquisa e, portanto, não consome espaço no banco de dados. Ela é considerada uma **tabela virtual**, que aponta para os campos definidos na instrução de criação dela.

Para se excluir uma visão, use o comando DROP VIEW <<nome da visão>>.

08

3 - Alterando os componentes de um banco de dados

Assim como os sistemas de informação evoluem com o tempo, o banco de dados também precisa evoluir. Essa evolução em banco de dados normalmente se dá de duas formas:

a) criação de novas tabelas e associações e

b) adição (ou remoção) de campos da tabela.

Uma forma eficiente de se modificar a estrutura de uma tabela sem que ocorra a perda de dados é utilizando o comando ALTER. O layout padrão do comando ALTER é:

```
ALTER <<Nome da tabela>> <<Operação>> <<Parâmetros da operação>>
```

Há muitos tipos de operações possíveis. Vamos ver algumas delas.

3.1 - Adicionando uma nova coluna (campo) em uma tabela

Para adicionar um novo campo a uma tabela, usamos o comando ALTER TABLE com o seguinte layout:

```
ALTER TABLE <<Nome da Tabela>> ADD COLUMN <<Nome da coluna>> <<Tipo de dado/Tamanho>>
```

Vamos acrescentar o campo CPF na tabela de alunos com o seguinte exemplo:

```
ALTER TABLE ALUNO ADD COLUMN CPF Varchar(11)
```

09

3.2 - Alterando propriedades de uma coluna em uma tabela

Há algumas possibilidades de alteração de estrutura e propriedades em relação aos campos de uma tabela. Por exemplo, podemos excluir ou modificar um valor padrão de um campo, usando para isso, operadores ALTER/SET ou DROP. Vejamos alguns exemplos.

Alterar o valor padrão do campo CPF para “00000000000”:

```
ALTER TABLE ALUNO ALTER COLUMN CPF SET DEFAULT TO '00000000000'
```

Vamos agora excluir esse valor padrão:

```
ALTER TABLE ALUNO ALTER COLUMN CPF DROP DEFAULT
```

Ao criar um novo campo em uma tabela que possui registros, todos os registros receberão NULL como valor para esse novo atributo.

10

3.3 - Excluindo uma coluna (campo) em uma tabela

Para excluir um campo a uma tabela, usamos o comando ALTER TABLE com o seguinte layout:

```
ALTER TABLE <<Nome da Tabela>> DROP COLUMN <<Nome da coluna>> RESTRICT/CASCADE
```

A opção **CASCADE** faz com que todos os elementos referenciados pela coluna também sejam excluídos. Esses elementos podem ser colunas em visões ou campos referenciados em outras tabelas.

Já se a opção **RESTRICT** for utilizada, o comando só será executado com sucesso caso a coluna não seja referenciada em nenhum objeto.

Vamos excluir o campo CPF na tabela de alunos com o seguinte exemplo:

```
ALTER TABLE ALUNO DROP COLUMN CPF CASCADE
```

Ao excluir uma coluna, estamos excluindo também todos os valores armazenados nessa coluna nos registros existentes na tabela.

11

3.4 - Alterando uma tabela para excluir um índice

Uma das alterações possíveis pelo comando ALTER é a exclusão de um índice. O layout para essa operação é:

```
ALTER <<Nome da tabela>> DROP INDEX <<Nome do Índice>>
```

Vamos ver um exemplo hipotético que excluiria um índice de nome de alunos da tabela de aluno:

```
ALTER TABLE ALUNO DROP INDEX IndiceNome
```

A mesma sintaxe vale para a exclusão de restrições, bastando mudar o nome INDEX para CONSTRAINT.

12

RESUMO

Neste módulo, aprendemos que:

a) Gatilhos (triggers) são como miniprogramas que são configurados para serem executados nas operações de manipulação de dados. Esses miniprogramas podem conter regras de negócios que têm por objetivo impedir ou permitir a modificação de dados, bem como executar outros procedimentos que podem ser usados para, por exemplo, enviar e-mails.
b) Os gatilhos podem ser configurados para serem executados antes ou depois das operações de manipulação. Quando eles são configurados para ocorrerem antes, eles têm a possibilidade de impedir que a ação aconteça.

c) As triggers são criadas pelo comando CREATE TRIGGER.

d) Uma view (visão) representa uma ou mais tabelas associadas no formato de uma consulta SQL. Utilizamos a mesma estrutura da instrução SELECT para acessar uma view.

e) O comando ALTER permite alterar a estrutura física de tabelas, índices e outros elementos do banco de dados. É possível, por exemplo, adicionar, alterar ou excluir campos de uma tabela, modificar valores padrões, configurações de NULL/NOT NULL, entre outros.

f) Ao excluir uma coluna, devemos usar a opção CASCADE para propagar referências em tabelas ou visões, ou usar a opção RESTRICT para impedir a exclusão de campos referenciados.

UNIDADE 3 – SQL

MÓDULO 4 – PROCEDIMENTOS ARMAZENADOS (STORED PROCEDURES)

01

1 - CRIAÇÃO DE UM PROCEDIMENTO ARMAZENADO

Olá, seja bem-vindo a mais uma etapa do nosso estudo. Este módulo descreve um recurso denominado **Procedimento Armazenado** (***Stored Procedure*** em inglês).

Um procedimento armazenado é um programa de computador que é executado dentro do banco de dados. Esse miniprograma é codificado na linguagem SQL e é, portanto, direcionado ao tratamento de operações em bancos de dados.

A maioria dos procedimentos armazenados implementados em ambientes reais não requer e nem possui interface com os usuários, ou seja, não depende de interação com o usuário, e todas as informações necessárias para que eles sejam executados são obtidas do próprio banco de dados. Entretanto, alguns comandos de impressão de texto em terminal, pedidos de confirmação ou informação para o usuário podem ser aplicados (são raríssimos os casos onde isso é realmente necessário; e quando usados, somente usuários administradores de dados ou DBAs utilizam esse tipo de recurso).

Os procedimentos armazenados são úteis nas seguintes **circunstâncias**:

- Se um programa de banco de dados é utilizado por várias aplicações, ele pode ser armazenado no servidor e invocado por qualquer um dos programas de aplicação. Isso reduz a duplicação de esforço e melhora a modularidade do *software*.
- A execução de um programa no servidor pode reduzir a transferência de dados e o custo de comunicação entre o cliente e o servidor em certas situações.
- Esses procedimentos podem melhorar o poder de modelagem fornecido pelas visões ao permitir que tipos mais complexos de dados derivados estejam disponíveis aos usuários do banco de dados. Além disso, eles podem ser usados para verificar restrições complexas que estão além do poder de especificação de triggers e restrições de campos de tabela.

02

Muitos SGBDs implementam a funcionalidade de procedimentos armazenados escritos em linguagem SQL (nem todo SGBD possui essa funcionalidade). Porém, alguns SGBDs vão ainda além, permitindo a criação de procedimentos armazenados em linguagem PL/SQL (Oracle Programming Language), C, C++, XML e/ou Java dentro do próprio SGBD.

Os procedimentos armazenados permitem a realização de quaisquer operações SQL. Veja alguns **exemplos de aplicabilidade.**

Esta parte do nosso estudo tem por propósito apresentar uma introdução ao tema, sendo que procedimentos mais complexos serão abordados na disciplina de Bancos de dados II.

Cada SGBD tem particularidades a respeito da criação de procedimentos armazenados. Vamos aqui apresentar os conceitos e padrões aplicáveis à maioria dos SGBDs. Entretanto, pode ser necessário adaptar as instruções fornecidas para um SGBD em particular. Nesse caso, consulte o manual do SGBD para confirmar os padrões dos comandos apresentados.

É ainda importante lembrar que a programação de bancos de dados é um assunto muito amplo. Há livros inteiros dedicados a cada técnica de programação de banco de dados e como essa técnica é realizada em um ambiente específico.

Novas técnicas e tecnologias são desenvolvidas a cada dia, e as mudanças nas técnicas existentes são incorporadas às versões mais recentes dos SGBDs. Mesmo sendo a SQL formalmente um padrão internacional, ela também é continuamente evoluída e adaptada a cada SGBD.

Exemplos de aplicabilidade

Alguns exemplos de aplicabilidade de procedimentos armazenados seriam:

- Geração automática de relatórios, como consolidados mensais.
- Notificação de usuários por meio do envio de e-mails.
- Importação ou exportação de dados de/para arquivos físicos, como arquivo CSV, XML ou TXT.
- Transferência de dados de um SGBD para outro, ou de um banco de dados para outro.
- Alimentação de bancos de dados de Data Warehouse.
- Validação de dados.
- Monitoramento de segurança de dados, baseada no registro de log de operações de usuários.
- Realização de consultas e/ou cálculos complexos em grande massa de dados que é muito mais eficiente ser executada dentro do banco de dados do que numa aplicação.
- Realização de rotinas de manutenção periódica que podem ser executadas em horários fora do expediente (normalmente de madrugada ou em finais de semana).

03

O **formato geral** da declaração de procedimentos armazenados é o seguinte:

```
CREATE PROCEDURE <<Nome do procedimento>> (<<Lista de parâmetros>>)
RETURNS <<Tipo de Retorno>>
<<Declarações locais>>
<<Instruções SQL>>;
```

Sendo que:

- **Nome do procedimento** é um nome como “CalculaSalarios”, ou “InformaGerente”, ou “EnviaEmail”, ou “MigrarDados”, ou “CriarRelatorioMensal”.
- **Lista de Parâmetros** são variáveis que podem ser passadas para dentro do procedimento, assim como passamos argumentos em objetos ou em funções nas linguagens de programação.
- **Tipo de retorno** refere-se a uma informação que o procedimento pode responder ao seu término, normalmente esses retornos podem ser usados para testar se o procedimento executou com sucesso ou mesmo como informações para outros procedimentos inter-relacionados.
- **Declarações locais** (opcionais, usadas apenas se necessário) são variáveis e cursores que podem ser criados, para fins de utilização de controles dentro do procedimento. Exemplo
- **Instruções SQL** é uma série de comandos que compõem o programa do procedimento armazenado. Podem incluir testes de lógica (IF/THEN/ELSE), loops, ponteiros, cursores, funções internas ou outros elementos. Exemplo

Para excluir um procedimento armazenado use o comando DROP PROCEDURE <<Nome do procedimento>>. Ao excluir um procedimento, nenhum dado ou estrutura do banco de dados é afetado.

Nome do procedimento
Uma característica comumente utilizada é a inclusão de um verbo no infinitivo ou no presente para nomear um procedimento armazenado. Muitas equipes de projeto também utilizam prefixos para definir nomes de objetos do banco de dados, como “spMigrarDados”, por exemplo, para indicar que é uma stored procedure.

Parâmetros
Esses parâmetros serão usados dentro do procedimento para a realização da operação. Por exemplo, um procedimento denominado “EnviarEmail” poderia receber como parâmetro a chave primária de um usuário do sistema, então, o procedimento poderia recuperar o e-mail dessa pessoa consultando a tabela de usuários e pesquisando o campo de e-mail por meio da chave primária fornecida.

Tipo de retorno
Um tipo de retorno muito comum é retornar verdadeiro, se o procedimento executou corretamente, ou falso se houve erro.

Exemplo (declarações locais)
Por exemplo, é possível criar um cursor baseado em uma instrução SELECT para recuperar dados a serem utilizados dentro do procedimento armazenado.

Exemplo (instrução)
Por exemplo, a rotina que identifica o endereço de e-mail de uma pessoa a partir da chave primária dele, e envia um e-mail com determinado texto coletado do banco de dados, retornando verdadeiro se tudo ocorreu a contento, é um exemplo de instrução.

04

Vamos ver agora como seria a **criação de um procedimento armazenado** que envia um e-mail para uma pessoa com um título e uma mensagem predefinidos. Iremos numerar as linhas abaixo apenas para explicar o que significa cada linha, essa numeração não existe na prática.

Clique sobre os números para ver a explicação.

```
1.    CREATE PROCEDURE EnviaEmail (IN ID_Destinatario INTEGER, OUT
Erro VARCHAR(100))
2.    RETURNS BOOLEAN
3.    BEGIN
4.    DECLARE EnderecoEmail VARCHAR(100);
5.    DECLARE TituloEmail VARCHAR(100);
6.    DECLARE TextoEmail VARCHAR(4000);
7.    DECLARE ConseguiuEnviar BOOLEAN;

8.    SELECT "Olá, este é o título do e-mail" INTO TituloEmail;
9.    SELECT "Este é o texto do e-mail. Bem-vindo ao mundo do SQL!"
INTO TextoEmail;
10.   SELECT Email INTO EnderecoEmail FROM Pessoa WHERE ID_Pessoa =
ID_Destinatario;

11.   IF EnderecoEmail IS NULL
12.   THEN
12.1. SELECT "Esta pessoa não existe na base de dados" INTO Erro;
12.2. RETURN FALSE;
13.   ELSEIF EnderecoEmail = "" THEN
13.1. SELECT "Esta pessoa não possui um e-mail cadastrado na base de
dados" INTO Erro;
13.2. RETURN FALSE;
14.   ELSE
14.1. Call SENDMAIL (EnderecoEmail, TituloEmail, TextoEmail,
ConseguiuEnviar)
15.   END IF;
16.   IF NOT ConseguiuEnviar
16.1. THEN SELECT "Não foi possível enviar o e-mail" INTO Erro;
17.   RETURN TRUE;
18.   END;
```

Note que cada comando completo é encerrado por um sinal de ponto de vírgula. Alguns comandos são executados em uma única sequência, como comandos SELECT. Outros comandos, como cláusulas IF/THEN/ELSE possuem várias sequências, e apenas na instrução ENDIF é que deve aparecer o ponto e vírgula para encerrar a operação.

1
Indica a criação de um procedimento armazenado, de nome EnviaEmail, que possui um parâmetro de entrada do tipo inteiro de nome ID_Destinatario, e um parâmetro de saída do tipo varchar de nome Erro. Os parâmetros serão explicados no próximo subitem.

2
Indica que o procedimento irá retornar um valor booleano, V ou F, indicado se a operação foi executada a contento ou se houve algum erro. Se houver erro, um texto explicativo do erro ocorrido será armazenado na variável Erro da linha anterior.

3
Indica que ali começam as instruções do procedimento. Deve ser finalizado posteriormente pela instrução END.

4
Declara a variável EnderecoEmail, que será utilizada para armazenar o endereço de e-mail do destinatário.

5
Declara a variável TituloEmail, que será utilizada para armazenar o título do e-mail a ser enviado.

6
Declara a variável TextoEmail, que será utilizada para armazenar o conteúdo do e-mail a ser enviado.

7
Declara a variável ConseguiuEnviar, que será utilizada para definir se o procedimento conseguiu ou não enviar o e-mail.

8
Define um título para o e-mail, armazenando o valor descrito dentro da variável TituloEmail.

9
Define um texto para o e-mail, armazenando o valor descrito dentro da variável TextoEmail.

10
Pesquisa o banco de dados Pessoa, procurando pelo e-mail da pessoa que tem a chave primária ID_Pessoa informada pelo parâmetro ID_Destinatario (argumento do procedimento, veja linha 1). O endereço identificado será armazenado na variável EnderecoEmail.

11
Testa a variável EnderecoEmail, se ela for nula, significa que não existe pessoa no banco de dados para a chave primária informada como parâmetro.

12
Indica que ali começa uma sub-rotina para quando o teste anterior for verdadeiro (e-mail nulo).

12.1
Atribui mensagem de erro no parâmetro de saída do procedimento.

12.2
Atribui o valor falso como resultado do procedimento.

13
Em caso de o valor de endereço não ser nulo, realiza outro teste para verificar se o endereço de e-mail consultado no passo 10 está em branco. Se estiver em branco significa que a pessoa existe na base, mas não possui um e-endereço de e-mail cadastrado.

13.1
Atribui mensagem de erro no parâmetro de saída do procedimento.

13.2
Atribui o valor falso como resultado do procedimento.

14
Se o endereço de e-mail não for nulo nem em branco, então se inicia a rotina a seguir que irá enviar o e-mail.

14.1
Chama a função interna de envio de e-mail, passando os parâmetros já definidos, e recebendo um V ou F no último parâmetro para identificar se conseguiu enviar ou não o e-mail.

14.2
Encerra o IF da linha 13.1.

15
Encerra o IF da linha 11.

16
Realiza um último teste para avaliar se conseguiu enviar o e-mail.

16.1
Atribui a mensagem de erro para o parâmetro de saída do procedimento.

17
Retorna verdadeiro, indicando que o procedimento executou com sucesso.

18
Encerra o procedimento.

05

1.1 - Parâmetros

Cada parâmetro de um procedimento armazenado deve ser declarado como um parâmetro SQL válido, como por exemplo, INTEGER, VARCHAR(200), BOOLEAN. Os parâmetros podem ser de três tipos:

- de entrada,
- de saída,
- de entrada e saída ao mesmo tempo.

Quando um **parâmetro é de entrada** significa que o procedimento armazenado deve receber esse parâmetro para dentro do procedimento, utilizá-lo, e então descartá-lo. Parâmetros de entrada são declarados como IN.

Exemplo: Um procedimento armazenado que enviar um e-mail pode ter a chave primária do usuário destinatário como parâmetro de entrada. Esse parâmetro seria declarado como `(IN ID_Destinatario INTEGER)`.

Um **parâmetro de saída** significa que quem dispara o procedimento armazenado receberá esse parâmetro como uma variável de saída, podendo utilizá-lo ao término do procedimento. Parâmetros de

saída são declarados como OUT.

Esse tipo de parâmetro tem o seu valor definido dentro do procedimento armazenado, geralmente, ou um valor consultado do banco de dados, ou um valor calculado pelo procedimento, ou um conjunto de ambos.

Exemplo: `(OUT Erro VarChar (100))`.

06

Por último, temos os parâmetros que são **de entrada e saída ao mesmo tempo**.

Esse tipo de parâmetro significa que a partir de um resultado inicial, fornecido por quem dispara o procedimento armazenado, tem seu valor modificado pelo procedimento armazenado e então retornado para a rotina que chamou o procedimento armazenado. Esse tipo de parâmetro é declarado como INOUT.

Exemplo: Suponha que um procedimento receba uma string de texto como parâmetro e altere essa string para letras com as iniciais em maiúsculas e as demais letras em minúsculas. Poderíamos tanto criar um procedimento com dois parâmetros, um para entrada e outro para saída, ou apenas um parâmetro que seria alterado pelo procedimento e posteriormente utilizado por quem chamou o procedimento.

Nesse caso, poderia ser especificado como:

```
(INOUT TextoASerConvertido Varchar (1000)).
```

07

1.2 - Executando um procedimento armazenado

O comando que faz com que um procedimento armazenado seja executado é o comando CALL. O *layout* desse comando é: `CALL <<Nome do procedimento armazenado>> (<<Lista de argumentos>>)` **.**

Um procedimento armazenado pode ser executado de diversas formas, quais sejam:

a) Via linha de comando

Um usuário com acesso de execução de operações no banco de dados pode abrir uma interface de acesso (gráfica ou textual) e executar o procedimento armazenado.

b) Por meio de uma TRIGGER (gatilho)

Uma trigger pode executar um procedimento armazenado, bastando para isso utilizar a função CALL.

c) Por meio de um JOB (tarefa agendada)

Um JOB (aprenderemos sobre ele em módulos futuros) representa uma tarefa agendada para ocorrer em uma determinada data e horário. Um job é um mecanismo muito utilizado para executar rotinas periódicas, como realização de procedimentos de backup.

d) Por meio de outro Procedimento Armazenado

Um procedimento armazenado pode chamar outros, bastando para isso usar a função CALL.

e) Por meio de um programa que acesse o banco de dados

Um sistema de informação ou um *software* que possua acesso ao banco de dados pode ser capaz de executar um procedimento armazenado. Esses programas podem utilizar bibliotecas como a ODBC para realizar chamadas a procedimentos armazenados.

08

2 - PRINCIPAIS MECANISMOS DE CONTROLE

Assim como as linguagens de programação em geral, a SQL também possui mecanismos de controle. Vamos apresentar os mecanismos mais comuns:

2.1 - IF/THEN/ELSE

O controle IF/THEN/ELSE é, sem dúvida, o mais comum e usado dos controles. Ele permite fazer comparações binárias (cujo resultado é verdadeiro ou falso) e realizar ações baseadas nessa comparação.

O *layout* padrão é:

```
IF <<condição >>
THEN <<lista de instruções para quando a condição é verdadeira>>
ELSE <<lista de instruções para quando a condição é falsa>>
ENDIF
```

Vale lembrar que a cláusula e instruções para a condição ELSE é **opcional**.

09

A instrução IF/THEN/ELSE permite aninhar testes, ou seja, usar a cláusula ELSEIF para realizar testes em sequência.

Para esse modo, o *layout* é:

```
IF <<condição 1>>
THEN <<lista de instruções para quando a condição 1 é verdadeira>>
ELSEIF <<condição 2>>
THEN <<lista de instruções para quando a condição 2 é verdadeira>>
ELSEIF <<condição 3>>
THEN <<lista de instruções para quando a condição 3 é verdadeira>>
...
ELSEIF <<condição n>>
THEN <<lista de instruções para quando a condição n é verdadeira>>
ELSE <<lista de instruções para quando a condição n é falsa
ENDIF
```

O exemplo que citamos no início deste módulo utiliza esse tipo de instrução condicional acima.

10

2.2 - WHILE/DO

O controle WHILE/DO representa um dos mecanismos cíclicos (loop). Um mecanismo cíclico repete uma sequência de operações enquanto o teste do operador WHILE for verdadeiro. Assim que o teste tornar-se falso, o loop é encerrado. Se o teste já for falso desde a primeira vez que for testado, as operações dentro do laço WHILE/DO nunca serão executadas.

O *layout* dessa instrução é:

```
WHILE <<condição>> DO
<<lista de instruções>>
END WHILE;
```

Exemplo, uma condição bem simples que exclui um-a-um os registros de uma tabela:

```
WHILE SELECT COUNT (*) FROM Pessoa > 0 DO
DELETE TOP(1) FROM Pessoa ORDER BY Pessoa_ID.
END WHILE;
```

11

2.3 - REPEAT/UNTIL

O controle REPEAT/UNTIL é um controle muito similar ao WHILE/DO, com a exceção de que o teste condicional é feito depois que a sequência de comandos é executada. Com isso, temos a garantia de que os comandos serão executados ao menos uma vez.

O *layout* padrão é:

```
REPEAT
<<lista de instruções>>
UNTIL <<condição>>
END REPEAT;
```

Exemplo, uma condição bem simples que exclui um-a-um os registros de uma tabela:

```
REPEAT
DELETE TOP(1) FROM Pessoa ORDER BY Pessoa_ID.
UNTIL SELECT COUNT (*) FROM Pessoa > 0
END REPEAT;
```

12

RESUMO

Neste módulo, aprendemos que:

a) Procedimentos armazenados (Stored Procedures) são como programas de computador que rodam dentro do ambiente do SGBD.
b) Os procedimentos armazenados podem executar qualquer tipo de operação SQL, desde operações básicas de manipulação de registros (instruções INSERT/UPDATE/DELETE) como instruções de manutenção de banco de dados (como criação de tabelas, rotinas de backup, etc).

c) Os procedimentos armazenados podem ser executados por operadores de bancos de dados, por triggers, por outros procedimentos armazenados, ou por outros programas e sistemas que tenham acesso ao banco de dados.

d) A lista de parâmetros oferece informações que são utilizadas pelo procedimento armazenado. Eventualmente, os parâmetros podem ser utilizados para saída de informações (tipos OUT e INOUT).

e) A cláusula RETURNS é utilizada para o retorno padrão dos procedimentos armazenados. Na prática, essa cláusula é utilizada principalmente para retornar se o procedimento foi executado com sucesso ou não.

f) Um procedimento armazenado é delimitado pelos indicadores BEGIN e END, no início e final do procedimento.

g) Instruções do tipo DECLARE são utilizadas para definir variáveis utilizadas dentro do procedimento.

h) Cada comando completo dentro do procedimento precisa ser encerrado por um sinal de ponto e vírgula.

i) Um procedimento armazenado por ser excluído pela cláusula DROP e executado pela cláusula CALL.

j) A cláusula IF/THEN/ELSE é um mecanismo de controle que permite comparações binárias (teste lógico do tipo V ou F) para executar um ou outro conjunto de ações (cláusulas THEN e ELSE).

k) Operadores WHILE/DO e REPEAT/UNTIL realizam operações em loop até que a condição de teste seja falsa.