CREATE TABLE

O comando CREATE TABLE cria a tabela solicitada e obedece à seguinte forma:

```
CREATE TABLE <tabela>
        (<descrição das colunas>),
        (<descrição das chaves)
```

onde:

<tabela> - é o nome da nova tabela a ser criada

<descrição das colunas> - é uma lista de colunas (campos) e seus respectivos tipos de dados. (smallint, char, money, varchar, integer, decimal, float, real, date, time, logical)

<descrição das chaves> - é a lista de colunas que são tratadas como chave estrangeira.

Alguns campos podem receber o valor **NULL** (nulo) e o campo definido como chave primária, além de não poder receber **NULL**, deve ser um campo **UNIQUE** (sem repetições – chave primária). Para o banco de dados estudado anteriormente temos os seguintes comandos:

```
CREATE TABLE CLIENTE (
        CODIGO_CLIENTE SMALLINT NOT NULL UNIQUE,
        NOME_CLIENTE CHAR(20),
        ENDERECO CHAR(30),
        CIDADE CHAR(15),
        CEP CHAR(8),
        UF CHAR(2),
        CGC CHAR(20),
        IE CHAR(20),
        PRIMARY KEY (CODIGO_CLIENTE)
)
CREATE TABLE PEDIDO (
        NUM_PEDIDO INT NOT NULL UNIQUE,
        PRAZO_ENTREGA SMALLINT NOT NULL,
        CODIGO_CLIENTE SMALLINT NOT NULL,
        CODIGO_VENDEDOR SMALLINT NOT NULL,
        PRIMARY KEY (NUM_PEDIDO),
        FOREIGN KEY (CODIGO_CLIENTE) REFERENCES CLIENTE,
        FOREIGN KEY (CODIGO_VENDEDOR) REFERENCES VENDEDOR
)
CREATE TABLE ITEM_DO_PEDIDO (
        NUM_PEDIDO INT NOT NULL UNIQUE,
        CODIGO_PRODUTO SMALLINT NOT NULL UNIQUE,
        QUANTIDADE DECIMAL,
        FOREIGN KEY (NUM_PEDIDO) REFERENCES PEDIDO,
        FOREIGN KEY (CODIGO_PRODUTO) REFERENCES PRODUTO,
        PRIMARY KEY (NUM_PEDIDO,CODIGO_PRODUTO)
)
```

```
CREATE TABLE VENDEDOR (
        CODIGO_VENDEDOR SMALLINT NOT NULL UNIQUE,
        NOME_VENDEDOR CHAR(20),
        SALARIO_FIXO MONEY,
        FAIXA_COMISSAO CHAR(1),
        PRIMARY KEY (CODIGO_VENDEDOR)
)
CREATE TABLE PRODUTO (
        CODIGO_PRODUTO SMALLINT NOT NULL UNIQUE,
        UNIDADE CHAR(3),
        DESCRICAO_PRODUTO CHAR(30),
        VAL_UNIT MONEY,
        PRIMARY KEY (CODIGO_PRODUTO)
)
```

DROP TABLE

Para eliminar uma tabela criada é utilizado o comando **DROP**:

```
DROP TABLE <tabela>
```

O seguinte comando elimina a tabela de pedidos que foi previamente criada:

```
DROP TABLE PEDIDO
```

ADICIONANDO REGISTRO À TABELA

```
INSERT INTO <tabela>
        (<nome da(s) coluna(s)>)
        VALUES (<valores>)
```

P. Adicionar o produto parafuso à tabela produto.
R. INSERT INTO PRODUTO VALUES (108, 'PARAFUSO', 'KG', 1.25)

ATUALIZANDO UM REGISTRO

```
UPDATE <tabela>
        SET <nome da(s) coluna(s)> = VALOR
        WHERE <CONDICOES>
```

P. Alterar o valor unitário do produto 'parafuso' de R$ 1.25 para R$ 1.62.
R. UPDATE PRODUTO SET VAL_UNIT = 1.62 WHERE DESCRICAO_PRODUTO = 'PARAFUSO'

P. Acrescentar 2,5% ao preço unitário dos produtos que estejam abaixo da média dos preços, para os aqueles comprados a quilo.
R. UPDATE PRODUTO SET VAL_UNIT = VAL_UNIT * 1.025 WHERE VAL_UNIT < (SELECT AVG(VAL_UNIT) FROM PRODUTO WHERE UNIDADE = 'KG')

SELECT

Uma das operações mais comuns, realizadas sobre um banco de dados é a de examinar (selecionar) as informações armazenadas. Neste item iremos mostrar várias situações de utilização do comando **SELECT**.

Selecionando colunas específicas da tabela:

```
SELECT <NOME(S) DA(S) COLUNA(S)> FROM <TABELA>
```

P. ***Listar todos os produtos com as respectivas descrições, unidades e valores unitários***
R. SELECT DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT FROM PRODUTO

P. ***Listar o nome do cliente com seu endereço e CGC***
R. SELECT NOME_CLIENTE, ENDERECO, CGC FROM CLIENTE

Selecionando todas as colunas da tabela:

```
SELECT * FROM <TABELA>
```

P. ***Listar todo o conteúdo de vendedor***
R. SELECT * FROM VENDEDOR

Selecionando apenas alguns registros da tabela:

```
SELECT <NOME(S) DA(S) COLUNA(S)> FROM <TABELA>
        WHERE <RESTRIÇÕES>
```

P. ***Listar o NUM_PEDIDO, o COD_PRODUTO e a quantidade dos itens do pedido com a quantidade igual a 35***
R. SELECT NUM_PEDIDO, CODIGO_PRODUTO, QUANTIDADE FROM ITEM_PEDIDO WHERE QUANTIDADE = 35

P. ***Listar os clientes que moram em NITEROI***
R. SELECT NOME_CLIENTE FROM CLIENTE WHERE CIDADE = 'NITEROI'

P. ***Listar os produtos que tenham unidade igual a 'M' e valor unitário igual a R$ 1,05***
R. SELECT DESCRICAO_PRODUTO FROM PRODUTO WHERE UNIDADE = 'M' AND VAL_UNIT = 1.05

P. ***Listar os clientes e seus respectivos endereços, que morem em São Paulo ou estejam na faixa de CEP entre 30077-000 e 30079-000***
R. SELECT NOME_CLIENTE, ENDERECO FROM CLIENTE WHERE (CEP >= '30077000' AND CEP <= '30079000') OR CIDADE = 'SÃO PAULO'

P. ***Listar todos os pedidos que não tenham prazo de entrega igual a 15 dias***
R. SELECT NUM_PEDIDO FROM PEDIDO WHERE NOT (PRAZO_ENTREGA = 15)
P. ***Listar o código e a descrição dos produtos que tenham o valor unitário na faixa de R$ 0,32 até R$ 2,00***
R. SELECT CODIGO_PRODUTO, DESCRICAO_PRODUTO FROM PRODUTO WHERE VAL_UNIT BETWEEN 0.32 AND 2.00
Os operadores **LIKE** e **NOT LIKE** só trabalham sobre colunas que sejam do tipo **CHAR**. Eles têm praticamente o mesmo funcionamento que os operadores **=** e **<>**, porém o poder desses operadores está na utilização dos símbolos (**%**) e (**_**) que podem fazer o papel de curinga.

% - substitui um conjunto de letras
_ - substitui um caractere

P. ***Listar todos os produtos que tenham a sua unidade começando por 'K'***
R. SELECT CODIGO_PRODUTO, DESCRICAO_PRODUTO FROM PRODUTO WHERE UNIDADE LIKE 'K_'
P. ***Listar os vendedores que não começam por 'JO'***
R. SELECT CODIGO_VENDEDOR, NOME_VENDEDOR FROM VENDEDOR WHERE NOME_VENDEDOR NOT LIKE 'JO%'
Os operadores **IN** e **NOT IN** pesquisam registros que estão ou não estão contidos no conjunto fornecido.

P. ***Listar os vendedores que são da faixa de comissão A e B***
R. SELECT NOME_VENDEDOR FROM VENDEDOR WHERE FAIXA_COMISSAO IN ('A','B')
A utilização do valor nulo é muito problemática, pois implementações distintas podem adotar representações distintas para o valor nulo.

P. ***Mostrar os clientes que não tenham inscrição estadual***
R. SELECT * FROM CLIENTE WHERE IE IS NULL
Quando se realiza uma seleção, os dados recuperados não estão ordenados. O SQL prevê a cláusula **ORDER BY** para realizar uma ordenação dos dados selecionados.

```
SELECT <NOME(S) DA(S) COLUNA(S)> FROM <TABELA>
        WHERE <RESTRIÇÕES>
        ORDER BY <NOME DA(S) COLUNA(S)> ASC |
        ORDER BY <NÚMERO DA(S) COLUNA(S)> DESC
```

A informação **<número da coluna>** se refere à posição relativa das colunas quando for apresentado o resultado da consulta, e não à posição na tabela original, contada da esquerda para a direita. As palavras **ASC** e **DESC** significam respectivamente, ascendente e descendente. A forma ascendente de ordenação é assumida como padrão.

P. ***Mostrar em ordem alfabética a lista de vendedores e seus respectivos salários fixos***
R. SELECT NOME_VENDEDOR, SALARIO_FIXO FROM VENDEDOR ORDER BY NOME_VENDEDOR
P. ***Listar os nomes, cidades e estados de todos os clientes, ordenados por estado e cidade de forma descendente***

R. SELECT NOME_CLIENTE, CIDADE, UF FROM CLIENTE ORDER BY UF DESC, CIDADE DESC

P. ***Listar a descrição e o valor unitário de todos os produtos que tenham a unidade 'KG', em ordem crescente de valor unitário***

R. SELECT DESCRICAO, VAL_UNIT FROM PRODUTO WHERE UNIDADE = 'KG' ORDER BY 2 ASC

Realizando cálculos com informação selecionada:

Com o SQL pode-se criar um campo que não pertença à tabela original e que seja fruto do cálculo sobre alguns campos da tabela.

P. Mostrar o novo salário fixo dos vendedores, de faixa de comissão 'C', calculando com base no reajuste de 75% acrescido de R$ 120,00 de bonificação. Ordenar pelo nome do vendedor.

R. SELECT NOME_VENDEDOR, NOVO_SALARIO = (SALARIO_FIXO * 1.75) + 120 FROM VENDEDOR WHERE FAIXA_COMISSAO = 'C' ORDER BY NOME_VENDEDOR

Utilizando funções sobre conjuntos:

P. Mostrar o menor e o maior salário de vendedor.

R. SELECT MIN(SALARIO_FIXO), MAX(SALARIO_FIXO) FROM VENDEDOR

P. Mostrar a quantidade total pedida para o produto 'VINHO' de código '78'.

R. SELECT SUM(QUANTIDADE) FROM ITEM_PEDIDO WHERE CODIGO_PRODUTO = '78'

P. Qual a média dos salários fixos dos vendedores?

R. SELECT AVG(SALARIO_FIXO) FROM VENDEDOR

P. Quantos vendedores ganham acima de R$ 2.500,00 de salário fixo?

R. SELECT COUNT(*) FROM VENDEDOR WHERE SALARIO_FIXO > 2500

Utilizando a cláusula DISTINCT:

Normalmente, vários registros dentro de uma tabela podem conter os mesmos valores, com exceção da chave primária. Com isso, muitas consultas podem trazer informações erradas. A cláusula **DISTINCT**, aplicada em uma consulta, foi criada para não permitir que certas redundâncias, obviamente necessárias, causem problemas.

P. Quais são as unidades de produtos, diferentes, na tabela produto?
R. SELECT DISTINCT UNIDADE FROM PRODUTO

Agrupando informações selecionadas (GROUP BY):

Utilizando a cláusula **GROUP BY**, é possível organizar a seleção de dados em grupos determinados.

```
SELECT <NOME(S) DA(S) COLUNA(S)> FROM <TABELA>
        WHERE <RESTRIÇÕES>
        GROUP BY <NOME DA(S) COLUNA(S)>
        ORDER BY <NÚMERO DA(S) COLUNA(S)>
```

P. listar o número de produtos que cada pedido contém.
R. SELECT NUM_PEDIDO, COUNT(*) TOTAL_PRODUTOS FROM ITEM_PEDIDO GROUP BY NUM_PEDIDO

Inicialmente, os registros são ordenados de forma ascendente por número do pedido. Num segundo passo, é aplicada a operação **COUNT(*)** para cada grupo de registros que tenha o mesmo número de pedido. Após a operação de contagem de cada grupo, o resultado da consulta utilizando a cláusula **GROUP BY** é apresentado.

Geralmente, a cláusula **GROUP BY** é utilizada em conjunto com as operações **COUNT** e **AVG**.

Agrupando de forma condicional (HAVING):

P. Listar os pedidos que têm mais do que 3 produtos.
R. SELECT NUM_PEDIDO, TOTAL_PRODUTOS = COUNT(*) FROM ITEM_PEDIDO GROUP BY NUM_PEDIDO HAVING COUNT(*) > 3

Utilizando consultas encadeadas (Subqueries):

O que é uma subquery? Em linhas gerais, é quando o resultado de uma consulta é utilizado por outra consulta, de forma encadeada e contido no mesmo comando SQL.

P. Que produtos participam em qualquer pedido cuja quantidade seja 10?
R. SELECT DESCRICAO_PRODUTO FROM PRODUTO WHERE CODIGO_PRODUTO IN (SELECT CODIGO_PRODUTO FROM ITEM_PEDIDO WHERE QUANTIDADE = 10)

P. Quais os vendedores que ganham um salário fixo abaixo da média?
R. SELECT NOME_VENDEDOR FROM VENDEDOR WHERE SALARIO_FIXO < (SELECT AVG(SALARIO_FIXO) FROM VENDEDOR)

RECUPERANDO DADOS DE VÁRIAS TABELAS (JOINS)

Até agora viemos trabalhando com a recuperação de dados sobre uma única tabela, mas o conceito de banco de dados reúne, evidentemente, várias tabelas diferentes. Para que possamos recuperar informações de um banco de dados temos, muitas vezes, a necessidade de acessar simultaneamente várias tabelas. Algumas dessas consultas necessitam realizar uma junção (JOIN) entre tabelas, para desta poder extrair as informações necessárias para a consulta formulada.

Temos o qualificador de nome qie consiste no nome da tabela seguido de um ponto e o nome da coluna na tabela, por exemplo: o qualificador de nome para a coluna **DESCRICAO_PRODUTO** da tabela **PRODUTO** será **PRODUTO.DESCRICAO_PRODUTO** Os qualificadores de nome são utilizados em uma consulta para efetivar a junção (JOIN) entre tabelas.

P. Juntar a tabela cliente com pedido.
R. SELECT NOME_CLIENTE, PEDIDO.CODIGO_CLIENTE, NUM_PEDIDO FROM CLIENTE, PEDIDO

Podemos observar que desta junção, poucas informações podem ser extraídas. Devemos então qualificar o tipo de junção, para podermos obter algum resultado concreto.

P. Que clientes fizeram os pedidos? Listar pelo nome dos clientes.
R. SELECT NOME_CLIENTE, PEDIDO.CODIGO_CLIENTE, NUM_PEDIDO FROM CLIENTE, PEDIDO WHERE CLIENTE.CODIGO_CLIENTE = PEDIDO.CODIGO_CLIENTE

A equação apresentada na cláusula **WHERE** é chamada de Equação de Junção. Podemos utilizar as cláusulas **LIKE**, **NOT LIKE**, **IN**, **NOT IN**, **NULL**, **NOT NULL** e misturá-las com os operadores **AND**, **OR** e **NOT**, dentro de uma cláusula **WHERE** na junção entre tabelas.

Para que não seja necessário escrever todo o nome da tabela nas qualificações de nome, podemos utilizar ALIASES (sinônimos) definidos na própria consulta. A definição dos ALIASES é feita na cláusula **FROM** e utilizada normalmente nas outras cláusulas (**WHERE**, **ORDER BY**, **GROUP BY**, **HAVING**, **SELECT**).

P. Quais os clientes que têm prazo de entrega superior a 15 dias e que pertencem aos estados de São Paulo (SP) ou Rio de Janeiro (RJ)?
R. SELECT NOME_CLIENTE, UF, PRAZO_ENTREGA FROM CLIENTE C, PEDIDO P WHERE UF IN ('SP','RJ') AND PRAZO_ENTREGA > 15 AND C.CODIGO_CLIENTE = P.CODIGO_CLIENTE

P. Mostrar os clientes e seus respectivos prazos de entrega, ordenados do maior para o menor.

P. Apresentar os vendedores (ordenados) que emitiram pedidos com prazos de entrega superiores a 15 dias e que tenham salários fixos igual ou superiores a R$ 1.000,00.

P. Mostre os clientes (ordenados) que têm prazo de entrega maior que 15 dias para o produto 'QUEIJO' e que sejam do Rio de Janeiro.

P. Mostre todos os vendedores que venderam chocolate em quantidade superior a 10 Kg.

P. Quantos clientes fizeram pedido com o vendedor João?

P. Quantos clientes da cidade do Rio de Janeiro, e Niterói tiveram os seus pedidos tirados com o vendedor João?

P. Quais os produtos que não estão presentes em nenhum pedido?

P Quais os vendedores que só venderam produtos por grama ('G')?

P. Quais clientes estão presentes em mais de três pedidos?

P. Apagar todos os vendedores com faixa de comissão nula.
R. DELETE FROM VENDEDOR WHERE FAIXA_COMISSAO IS NULL

P. Apagar todos os registros de pedidos realizados por vendedores fantasmas.

UTILIZANDO VIEWS

As tabelas criadas em um banco de dados relacional têm existência física dentro do sistema de computação. Muitas vezes é necessário criar tabelas que não ocupem espaço físico, mas que possam ser utilizadas como tabelas normais. Essas são chamadas de VIEWS (Tabelas Virtuais).

```
CREATE VIEW <NOME DA VIEW>
        (<nome da(s) coluna(s)>) AS
        SELECT <nome da(s) coluna(s)> FROM <tabela>
        WHERE <RESTRIÇÕES>
```

As VIEWS são utilizadas para se ter uma particular visão de uma tabela, para que não seja necessária a utilização do conjunto como um todo.

P. Criar uma VIEW que contenha só os produtos pedidos a metro.
R. CREATE VIEW PR_METRO (COD_PR_METRO, DESCRICAO, UNIDADE) AS SELECT CODIGO_PRODUTO, DESCRICAO_PRODUTO, UNIDADE FROM PRODUTO WHERE UNIDADE = 'M'

P. Criar uma VIEW contendo o código do vendedor, o seu nome e o seu salário fixo com R$ 100.00 de bônus.

P. Criar uma VIEW contendo os vendedores, seus pedidos efetuados e os respectivos produtos.

As VIEWS criadas passam a existir no banco de dados como se fossem tabelas reais. As VIEWS são voláteis, desaparecendo no final da sessão de trabalho. Depois de criadas elas podem ser utilizadas em todas as funções de programação da linguagem SQL (listar, inserir, modificar, apagar, etc.).

P. Mostrar os vendedores que possuem salário fixo superior a R$ 1.000,00 após o bônus de R$ 100,00.

P. Inserir o registro: 110, Linha_10, M, na VIEW PR_METRO. Verifique o conteúdo da tabela PRODUTO.

P. Alterar o descrição de Linha_10 para Linha_20 no código 110 da VIEW PR_METRO. Verifique o conteúdo da tabela PRODUTO.

P. Apagar da VIEW o registro de código 110 da VIEW PR_METRO. Verifique o conteúdo da tabela PRODUTO.

TRABALHANDO COM ÍNDICES

Índice é uma estrutura que permite rápido acesso às linhas de uma tabela, baseados nos valores de uma ou mais colunas. O índice é simplesmente uma outra tabela no banco de dados, na qual estão armazenados valores e ponteiros, arrumados de forma ascendente ou descendente. Através dos ponteiros se localiza em qual linha o valor desejado está armazenado. As tabelas de índice são utilizadas internamente, ficando transparente ao usuário a sua utilização.

Cada índice é aplicado a uma tabela, especificando uma ou mais colunas desta tabela.

```
CREATE [UNIQUE] INDEX <nome do índice>
        ON <tabela> (<coluna(s)>
```

A cláusula UNIQUE é opcional e define que para aquela coluna não existirão valores duplicados, ou seja, todos os dados armazenados na coluna serão únicos. A junção do índice UNIQUE e da especificação NOT NULL para uma coluna a chave primária da tabela.

A criação dos índices depende muito do projeto do banco de dados e das necessidades de pesquisa formuladas pelos usuários do banco de dados. Os índices estão muito ligados às necessidades de velocidade na recuperação da informação, e na execução rápida de uma operação de JOIN.

P. Criar a tabela de índices chamada NOME_PRO baseada no campo DESCRICAO_PRODUTO da tabela PRODUTO.
R. CREATE INDEX NOME_PRO ON PRODUTO(DESCRICAO)

P. Criar a tabela de índices PED_PRO baseada na concatenação dos campos NUM_PEDIDO e COD_PRODUTO da tabela ITEM_PEDIDO.

P. Criar o índice único para a tabela cliente baseada no código do cliente, não podendo haver duplicidade de informação armazenada.

MAIS EXERCÍCIOS COM JOINS

Temos três tipos de JOINS:

- INNER JOINS: inclui apenas as linhas que satisfazem as condições de JOIN;
- CROSS JOINS: inclui todas as combinações de todas as linhas entre as duas tabelas;
- OUTER JOINS: inclui as linhas que satisfazem a condição de JOIN de uma tabela e as linhas remanescentes de uma das duas tabelas que participam do JOIN.

O INNER JOIN conecta as duas tabelas em uma terceira tabela que inclui apenas as linhas que satisfazem a condição de JOIN. Os dois tipos comuns de INNER JOIN são *equijoin* e *join natural*.

Equijoin é um join no qual os valores das colunas que estão sendo comparados são iguais. Um equijoin produz uma coluna redundante porque a coluna de conexão é exibida duas vezes.

O join natural é um pelo qual os valores que estão sendo comparados são iguais para uma ou mais tabelas, mas a coluna de conexão é exibida apenas uma vez. O join natural elimina a coluna redundante no conjunto de resultados.

O CROSS JOIN produz um resultado que inclui todas as combinações de todas as linhas entre as tabelas que participam do join. Por exemplo, se existirem 8 linhas em uma tabela e 9 linhas em outra tabela, os resultados incluem 72 colunas. Um resultado deste tipo geralmente é inútil.

O OUTER JOIN permite que você restrinja uma tabela enquanto, não restrinjimos as linhas da outra tabela. Isto é últil quando se deseja verificar onde chaves primárias e chaves secundárias estão fora de sincronismo, ou possuem membros órfãos. Um OUTER JOIN só pode ser realizado entre duas tabelas. O ANSI SQL inclui estes três tipos de OUTER JOIN:

- LEFT OUTER JOIN que inclui todas as linhas da tabela que está à esquerda na expressão;
- RIGHT OUTER JOIN que inclui todas as linhas da tabela que está à direita na expressão;
- FULL OUTER JOIN que inclui todas as linhas que não se casam tanto da tabela à direita quanto da tabela à esquerda.

Existem duas possíveis sintaxes para representar o JOIN entre tabelas: sintaxe **ANSI JOIN** e sintaxe **SQL Server join**.

P. Realize o OUTER JOIN entre as tabelas de item de pedido e de produto.
R. SELECT I.NUM_PEDIDO, P.DESCRICAO_PRODUTO FROM ITEM_DO_PEDIDO I, PRODUTO P WHERE I.CODIGO_PRODUTO =* P.CODIGO.PRODUTO

No caso acima estamos representando a sintaxe do **SQL Server join**.

O mesmo comando feito segundo a sintaxe do **ANSI JOIN** é representada abaixo:

R. SELECT I.NUM_PEDIDO, P.DESCRICAO_PRODUTO FROM PRODUTO P LEFT OUTER JOIN ITEM_DO_PEDIDO I ON I.CODIGO_PRODUTO = P.CODIGO.PRODUTO

Observe que os exercícios abaixo devem ser feitos com a sintaxe do **ANSI JOIN**, pois com a sintaxe do **SQL Server join** ele falham.

P. Quais os produtos que não estão presentes em nenhum pedido?

P. Quais os vendedores que não intermediaram nenhum pedido?

P. Quais os clientes que nãofizeram nenhum pedido?

Uma outra classe de JOINS são os SELF JOINS que correlacionam as linhas de uma tabela com as linhas da mesma tabela. Um SELF JOIN é utilizado quando estamos interessados em comparar a mesma informação. Por exemplo:

P. Listar os clientes que estão na mesma cidade aos pares.
R. SELECT C1.NOME_CLIENTE, C2.NOME_CLIENTE FROM CLIENTE C1, CLIENTE C2 WHERE C1.CIDADE = C2.CIDADE

P. Listar os produtos que possuem as mesmas medidas aos pares.

SUBQUERY CORRELACIONADA

A maioria das subqueries anteriores eram executadas apenas uma vez, e substituindo-se o valor resultante na cláusula WHERE da query externa. A subquery correlacionada é uma na qual a cláusula WHERE referencia a tabela na cláusula FROM da cláusula da query externa. Isto significa que a subquery é executada repetidamente, uma para cada linha que possa ser selecionada pela query externa.

P. Listar os produtos e suas quantidades máximas presentes nos pedidos.
R. SELECT DISTINCT I.CODIGO_PRODUTO, I.QUANTIDADE FROM ITEM_DO_PEDIDO I WHERE QUANTIDADE = (SELECT MAX(QUANTIDADE) FROM ITEM_DO_PEDIDO WHERE CODIGO_PRODUTO = I.CODIGO_PRODUTO)

P. Listar o número de pedidos que cada cliente fez..
R.

P. Listar o número de pedidos que cada vendedor cadastrou..
R.

P. Listar o total de entrada previsto para cada vendedor.
R.

P. Listar o valor total associado a cada pedido.
R.

P. Listar o total de entrada de dinheiro associado a cada cliente.
R.

TRABALHANDO COM TRIGGERS

O TRIGGER é um tipo especial de Procedimento que é acionado quando alguma modificação é realizada na tabela utilizando-se dos comandos INSERT, UPDATE ou DELETE.

Para o exemplo de TRIGGERS vamos definir uma nova tabela, cópia de produtos. Que conterá o histórico das modificações feitas na tabela de produtos.

```
CREATE TABLE PRODUTO_COPY (
      PRODUTO_ID SMALLINT IDENTITY,
      CODIGO_OP CHAR(1),
      CODIGO_PRODUTO SMALLINT,
      DESCRICAO_PRODUTO CHAR(30),
      UNIDADE CHAR(3),
      VAL_UNIT MONEY NULL,
      PRIMARY KEY (PRODUTO_ID)
)
```

Criando TRIGGERS para a tabela de produtos.

```
CREATE TRIGGER PRODUTO_INSERT ON
 PRODUTO FOR
 INSERT AS
   INSERT INTO PRODUTO_COPY (CODIGO_OP, CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT) SELECT 'I', CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT FROM INSERTED

CREATE TRIGGER PRODUTO_DELETE ON
 PRODUTO FOR
 DELETE AS
   INSERT INTO PRODUTO_COPY (CODIGO_OP, CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT) SELECT 'D', CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT FROM DELETED

CREATE TRIGGER PRODUTO_UPDATE
 ON PRODUTO FOR UPDATE AS
 BEGIN
   INSERT INTO PRODUTO_COPY (CODIGO_OP, CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT) SELECT 'D', CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT FROM DELETED
 INSERT INTO PRODUTO_COPY (CODIGO_OP, CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT) SELECT 'U', CODIGO_PRODUTO,
   DESCRICAO_PRODUTO, UNIDADE, VAL_UNIT FROM INSERTED
 END
```

TRABALHANDO COM STORED PROCEDURES

As **STORED PROCEDURES** são criadas utilizando a cláusula **CREATE PROCEDURE**. A cláusula **CREATE PROCEDURE** não poderá ser executada em combinação com outras cláusulas SQL em um único comando.

Criando uma **STORED PROCEDURE**:

```
CREATE PROCEDURE COUNT_PEDIDOS
AS
SELECT COUNT(*) FROM PEDIDO
```

Executando a **STORED PROCEDURE**:

```
EXEC COUNT_PEDIDOS
```

Se você executar um procedimento que aciona outro procedimento, o procedimento acionado por acessar objetos criados pelo procedimento acionante. Isto é feito definindo-se parâmetros com a cláusula **CREATE PROCEDURE** utilizando estas opções:

@parameter_name datatype [= default] [OUTPUT]

- **@parameter_name**: especifica o parâmetro no procedimento. Um ou mais parâmetros podem ser declarados em uma cláusula **CREATE PROCEDURE**. O usuário precisa fornecer os valores de cada parâmetro declarado quando o procedimento é executado (a menos que um valor default seja definido para este parâmetro);
- **datatype**: especifica o tipo do parâmetro;
- **default**: especifica o valor default para o parâmetro.

```
CREATE PROCEDURE COUNT_PEDIDOS_VEN
@VEN SMALLINT
AS
SELECT COUNT(*) FROM PEDIDO WHERE CODIGO_VENDEDOR = @VEN
```

```
CREATE PROC COUNT_PEDIDOS_VEN1
  @VEN SMALLINT,
  @SAIDA SMALLINT OUTPUT
AS
BEGIN
SELECT @SAIDA = (SELECT COUNT(*) FROM PEDIDO WHERE
                    CODIGO_VENDEDOR = @VEN)
END
```

A segunda **STORED PROCEDURE** utiliza uma cláusula **SELECT** para atribuir um valor à variável **@SAIDA**. Estas duas **STORED PROCEDURES** são equivalentes. Sendo que a execução de cada uma delas é feita de modo distinto, respectivamente:

```
EXEC COUNT_PEDIDOS_VEN 11
```

```
DECLARE @SAIDA2 SMALLINT
EXEC COUNT_PEDIDOS_VEN1 11, @SAIDA2 OUTPUT
SELECT 'TOTAL = ', @SAIDA2
```

Uma outra **STORED PROCEDURE**, neste caso para calcular o produto entre dois números inteiros está definida abaixo:

```
CREATE PROCEDURE MATHTUTOR
  @M1 SMALLINT,
  @M2 SMALLINT,
  @RESULT SMALLINT OUTPUT
AS
  SELECT @RESULT = @M1 * @M2

DECLARE @GUESS SMALLINT
SELECT @GUESS = 50
EXECUTE MATHTUTOR 5, 6, @GUESS OUTPUT
SELECT 'THE RESULT IS: ', @GUESS
```

Neste caso estaremos executando a **STORED PROCEDURE** para calcular o produto entre **5** e **6**. Primeiro armazenamos um valor incorreto na variável **@GUESS** para verificarmos que será substituído pelo resultado correto durante o acionamento da **STORED PROCEDURE MATHTUTOR**. Se a palavra **OUTPUT** não estiver presente no acionamento da **STORE PROCEDURE**, ela ainda será executada mas o valor da variável **@GUESS** não será modificado.

CRIANDO PROGRAMAS – COMANDO WHILE

Existem várias cláusulas para controlar o fluxo de programas definidos em SQL. Alguns deles estão listados abaixo:

- **DECLARE**: declara uma variável local;
- **RETURN**: retorna incondicionalmente;
- **CASE**: permite que uma expressão tenha valores retornados condicionalmente;
- **BEGIN..END**; define um bloco;
- **WHILE**: repete uma cláusula enquanto a condição for verdadeira;
- **IF..ELSE**: define uma execução condicional, e opcionalmente, uma execução alternativa;
- **BREAK..CONTINUE**: **BREAK** abandona o laço mais interno e **CONTINUE** reinicia o laço **WHILE**.

```
SELECT * FROM PRODUTO

DECLARE @NEXT CHAR(30)
SELECT @NEXT = ' '

WHILE @NEXT IS NOT NULL
BEGIN
  SELECT @NEXT = MIN(NAME)
    FROM SYSOBJECTS
    WHERE TYPE = 'U' AND NAME > @NEXT
  EXEC SP_SPACEUSED @NEXT
END
```

Nesta programa estamos utilizando uma tabela do sistema de nome **SYSOBJECTS** (tabela que associa um record para cada tabela, view, stored procedure, triggers, etc... existentes na base de dados), e estamos executando uma **STORED PROCEDURE** do sistema **SP_SPACEUSED** que reporta quanto de espaço a tabela está utilizando.

A expressão **CASE** permite que definamos um valor para cada expressão. Um **CASE** é equivalente a vários **IFs**.

```
SELECT NOME_VENDEDOR, CATEGORIA = CASE FAIXA_COMISSAO
  WHEN 'A' THEN 'FAIXA SUPERIOR'
  WHEN 'B' THEN 'FAIXA INTERMEDIARIA'
  WHEN 'C' THEN 'FAIXA INFERIOR'
  ELSE 'FAIXA NAO CADASTRADA'
END FROM VENDEDOR
```

P. Listar os produtos e suas unidades por extenso.
R.

O bloco **IF..ELSE** impõe condições para a execução da cláusula SQL.

```
IF EXISTS (SELECT * FROM PRODUTO WHERE UNIDADE NOT IN ('M', 'G', 'KG', 'SAC', 'L'))
  BEGIN
    SELECT  * FROM PRODUTO WHERE UNIDADE NOT IN ('M', 'G', 'KG', 'SAC', 'L')
  END
```