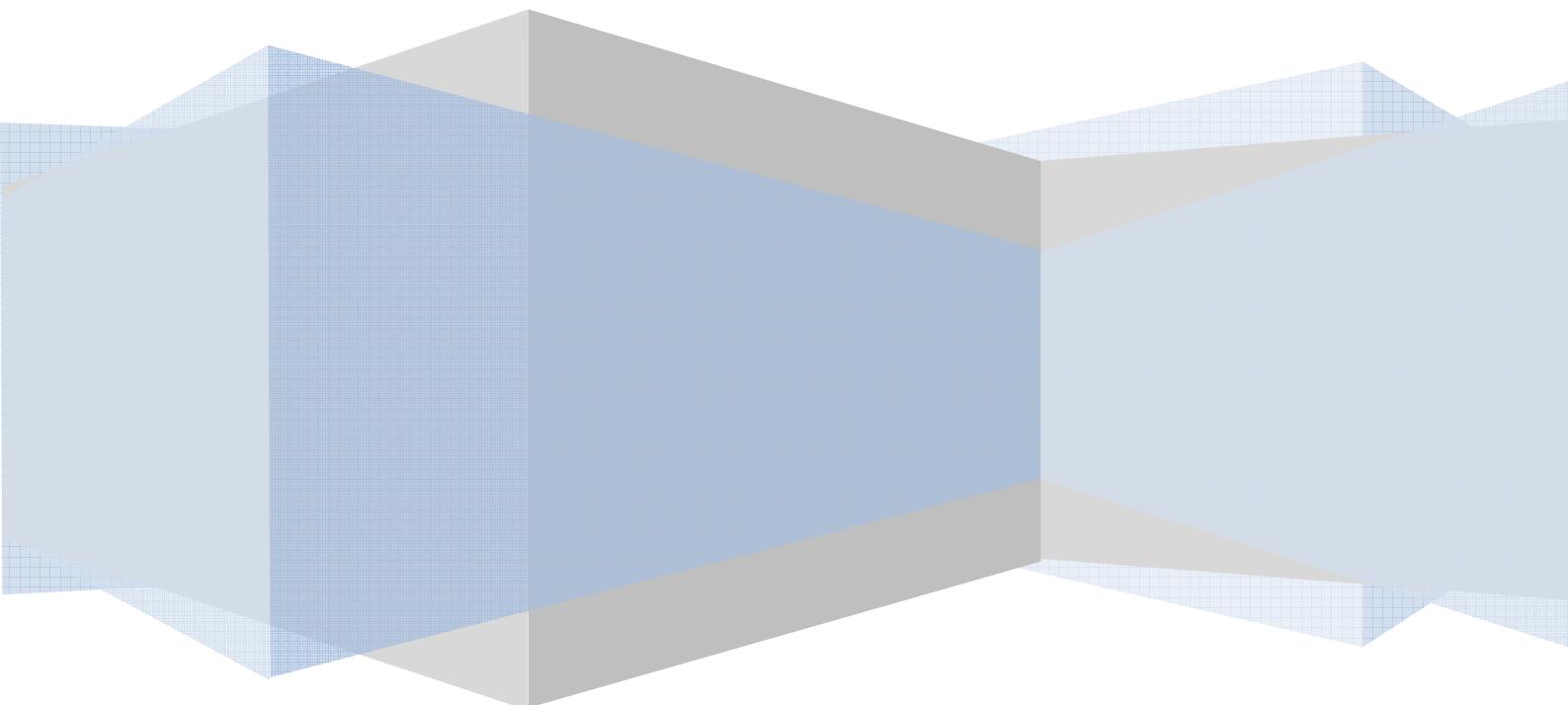


Apostila de SQL Server 2005 Express Edition

Conceitos básicos



Conteúdo

Introdução.....	10
Capítulo 1	11
O Que É SQL?.....	11
O Que É Banco De Dados Relacional?.....	11
SQL Server e o Transact-SQL.....	11
Exercícios.....	12
Capítulo 2	13
Conhecendo o SQL Server 2005 Management Studio Express	13
Capítulo 3	15
Objetos do SQL Server 2005.....	15
Manipulando e Conhecendo os Objetos DATATYPES, DATABASEs e TABLEs.....	16
DATATYPES	16
DATABASE	17
TABLE	20
Exercícios.....	22
Capítulo 4	23
DML Simples.....	23
INSERT.....	23
UPDATE.....	25
DELETE.....	26
SELECT	26
Exibindo Todas as Colunas da Tabela	27
Exibindo Colunas com ALIAS.....	27
Colunas Virtuais.....	28
Exibindo um Cálculo com o Comando SELECT	29
Exibindo Apenas Uma Vez Registros Repetidos (DISTINCT)	29
Copiando Dados De Uma Tabela Para Outra Com INSERT e SELECT.....	31
INSERT Com SELECT Quando as Tabelas São Iguais.....	31
INSERT Com SELECT Quando as Tabelas Não São Iguais.....	32
Exercícios.....	33
Laboratório	34
Capítulo 5	38
A Cláusula WHERE	38

Sintaxe Básica.....	39
Selecionando Registros Específicos.....	39
Atualizando Registros Específicos	39
A Cláusula WHERE Com os Operadores AND e OR.....	40
A cláusula WHERE com o operador IN	40
A Cláusula WHERE Com os Operadores NOT IN	41
A Cláusula WHERE Com o Operador BETWEEN	41
A Cláusula WHERE Com os Operadores NOT BETWEEN.....	42
A Cláusula WHERE Com o Operador LIKE	43
A Cláusula WHERE Com os Operadores NOT LIKE	44
Exercícios.....	45
Laboratório	46
Capítulo 6	47
A Cláusula ORDER BY.....	47
Ordenando Por Colunas	47
Ordenando Por Mais de Uma Coluna	48
ORDER BY ASC e DESC.....	48
ASC e DESC	49
A Cláusula TOP.....	49
A Cláusula TOP Com ORDER BY.....	50
A Cláusula TOP WITH TIES Com ORDER BY	51
Exercícios.....	52
Laboratório	53
Capítulo 7	54
Integridade E Consistência - Parte 1	54
CONSTRAINTS.....	54
Chaves Primárias (PRIMARY KEY)	54
Chaves Secundárias ou Únicas (UNIQUE)	56
Chave Estrangeira (FOREIGN KEY / REFERENCES)	56
Regras de Validação (CHECK).....	57
Valor Padrão (DEFAULT).....	58
Valores NULL.....	59
Regras de constraints.....	60
Criando Tabelas Com Todas as Regras de Integridade e Consistência.....	60

Modelo Entidade - Relacionamento (MER)	62
Relacionamento.....	62
Relacionamento 1 : 1.....	63
Relacionamento 1 : N	63
Relacionamento N : N.....	64
Exercícios.....	67
Laboratório	68
Capítulo 8	79
Associando tabelas.....	79
JOIN	79
INNER JOIN	80
LEFT JOIN	81
RIGHT JOIN	81
Associando Mais de Duas Tabelas.....	82
FULL OUTER JOIN	83
CROSS JOIN	83
Os Comandos UPDATE e DELETE Com Utilização do JOIN	84
Exercícios.....	87
Laboratório	88
Capítulo 9	90
UNION ALL	90
Regras de Utilização	91
Subquery	91
Formatos Apresentados Comumente Pelas Subqueries	92
Exemplo de Subquery	92
Regras de Utilização	93
Exercícios.....	94
Capítulo 10.....	95
Totalizando Dados.....	95
GROUP BY	95
HAVING Com GROUP BY.....	96
WITH ROLLUP Com GROUP BY.....	97
WITH CUBE Com GROUP BY	98
Exercícios.....	100

Laboratório	101
Capítulo 11.....	103
Entendendo os Arquivos de Dados.....	103
Arquivo Primário (PRIMARY DATA FILE)	103
Arquivo Secundário (SECONDARY DATA FILE)	103
Arquivo de Log (LOG DATA FILE)	104
Nome Físico e Lógico.....	104
Grupos de Arquivos (FILEGROUP).....	104
Regras dos FILEs e FILEGROUPs.....	105
Recomendações de Uso	105
Tamanho dos Arquivos	105
Múltiplos Arquivos.....	105
Tabelas.....	105
Separando o Log e Arquivo de Dados	106
Exemplo de Criação de Database.....	106
Criando Tabelas Dentro de FILEGROUPs	107
Transaction Log.....	108
Exercícios.....	109
Laboratório	113
Capítulo 12.....	114
Integridade e Consistência - Parte 2.....	114
UDDT (User-Defined Datatypes)	114
NULL / NOT NULL	115
PRIMARY KEY	115
FOREIGN KEY / REFERENCES.....	115
DEFAULT.....	116
RULEs	116
Ação Em Cadeia	117
Removendo Uma Constraint Com o Comando ALTER TABLE.....	118
Adicionando Uma Constraint Com o Comando ALTER TABLE.....	118
Adicionando Uma Coluna a Tabela	118
Removendo Uma Coluna da Tabela.....	119
Habilitando e/ou Desabilitando TRIGGERS de Uma Tabela.....	119
Habilitando e/ou Desabilitando Constraints	119

IDENTITY	120
Adicionando Uma Coluna IDENTITY Com ALTER TABLE	120
SET IDENTITY_INSERT	120
Obtendo Informações da Coluna IDENTITY	121
DBCC CHECKIDENT	121
IDENTITYCOL	121
IDENT_CURRENT	121
IDENTITY_INCR.....	122
IDENTITY_SEED.....	122
@@IDENTITY	122
SCOPE_IDENTITY	122
Exercícios.....	123
Laboratório	125
Capítulo 13.....	126
O Que é Uma Views?	126
Criando Uma VIEW.....	126
Vantagem das VIEWS.....	127
WITH ENCRYPTION.....	128
WITH CHECK OPTION.....	128
Excluindo Uma VIEW	129
Alterando Uma VIEW	129
VIEWS Indexadas	129
Diretrizes Para Criação de Índices Sobre Uma VIEW	129
Criando e Indexando VIEWS.....	130
Exercícios.....	131
Capítulo 14.....	132
Programando Com Transact-SQL.....	132
Variáveis de Usuário.....	132
Atribuindo Valor a Uma Variável.....	132
Controle de Fluxo.....	133
EXECUTE	136
STORED PROCEDURE	137
Tipos de STORED PROCEDURES.....	137
System Stored Procedure	138

Extended Stored Procedures.....	139
Stored Procedures Temporárias.....	140
Stored Procedures Remotas	140
Stored Procedures Locais	141
Parâmetros.....	141
Passagem de Parâmetro Por Referência.....	142
Retornando Um Valor	142
A Criação de Uma STORED PROCEDURE.....	143
Execução de Uma Procedure	143
Otimização da Procedure.....	143
Compilação da Procedure.....	144
As Próximas Execuções da Procedure	144
Vantagem das Procedures.....	144
Considerações na Criação de STORED PROCEDURES	144
Excluindo Uma Procedure.....	145
Alterando Uma STORED PROCEDURE.....	145
Recompilando Uma STORED PROCEDURE.....	145
Observações Quando ao Uso de STORED PROCEDUREs.....	145
Exercícios.....	147
Laboratório	150
Capítulo 15.....	159
Transações.....	159
Exemplo de Transação.....	160
Transações Explícitas.....	161
Transações Implícitas	161
Considerações Sobre Transações	162
Manipulando Erros	162
SET XACT_ABORT	162
RAISERROR	163
A Variável @@ERROR.....	164
Criando Uma Mensagem de Erro	165
Eliminando Uma Mensagem de Erro.....	166
Batch	167
Exercícios.....	168

Laboratório	171
Capítulo 16.....	172
Funções.....	172
Regras Para Utilização de Funções	172
Funções Built-In	173
Tipos de USER DEFINED FUNCTIONS	173
Funções Escalares.....	173
Funções Table-Valued.....	174
Funções Que Contêm Vários Comandos e Que Retornam Dados de Uma Tabela.....	175
Exercícios.....	176
Capítulo 17.....	179
TRIGGERS.....	179
TRIGGER x Transação	179
TRIGGERS São Reativas	182
Tabelas Criadas Em Tempo de Execução	182
TRIGGER DDL.....	182
Alterando ou Eliminando Um TRIGGER.....	183
TRIGGERS Aninhados	183
INSTEAD OF	184
Exercícios.....	185
Laboratório	188
Capítulo 18.....	189
Concorrência.....	189
LOCKS	189
Tipos de LOCKs.....	190
SHARED (S) LOCKs	190
UPDATE (U) LOCKs.....	190
EXCLUSIVE (X) LOCKs	191
INTENT (I) LOCKs	191
EXTENT LOCKs.....	191
SCHEMA (SCH) LOCKs.....	192
Granularidade	192
Problemas de Concorrência Impedidos Pelos LOCKs.....	193
LOST UPDATE.....	193

UNCOMMITTED DEPENDENCY	193
INCONSISTENT ANALYSIS (NONREPEATABLE READ).....	193
PHANTOMS.....	193
Customizando o LOCK.....	194
Customizando LOCKs na Seção	194
LOCK Dinâmico.....	195
TIMEOUT.....	196
Exercícios.....	197
Capítulo 19.....	199
Distribuição de Dados	199
Acessando Dados de um SQL Server Remoto	199
Conectando-se a um SQL Server Remoto	199
Conectando-se a Uma Origem OLEDB	200
Segurança.....	200
Acessando Dados do Servidor Conectado Com o Nome Totalmente Qualificado.....	201
Acessando Dados do Servidor Conectado Com a Função OPENQUERY().....	202
Comando e Ações Proibidos no Servidor Conectado	202
Opções Para Configurar o Servidor Conectado	202
Modificando Dados em Um Servidor Remoto.....	203
Como as Transações São Distribuídas	204
Participando de Transações Distribuídas	205
Considerações	205
MS DTC (Microsoft Distributed Transaction Coordinator)	205
Exercícios.....	207
Apêndice	210
Banco de Dados de Sistema	210
RESOURCE.....	210
MASTER.....	210
TEMPDB	210
MODEL.....	210
MSDB	211
BACKUP	211
Stored Procedures Úteis.....	211

Introdução

Criei essa apostila com o intuito de compartilhar o pouco de conhecimento que possuo sobre essa ferramenta, o SQL Server 2005 Express. Não é de meu interesse cobrar sobre esse material qualquer tipo de quantia ou valor de qualquer gênero, tendo em vista que retirei boa parte do material aqui contido, de outras apostilas que possuo, de meus próprios conhecimentos sobre essa ferramenta tão poderosa e outros lugares (revistas, sites, arquivos de ajuda, etc).

Capítulo 1

O Que É SQL?

SQL significa **Structured Query Language** (inglês), ou Linguagem Estruturada de Pesquisa (português). Baseia-se em um modelo de dados relacional criado pelo inglês **Edgard F. Codd** no início dos anos 70 para ser usado no trabalho com banco de dados.

O Que É Banco De Dados Relacional?

Em um **Banco de Dados Relacional**, as informações estão dispostas na forma de tabelas. Uma tabela do **BDR** possui os seguintes elementos principais:

- **Atributos** (campos), que são representados pelas colunas.
- **Registros** (dados), que são representados pelas linhas.

Num **BDR** pode-se fazer uso de várias tabelas. Essas tabelas, ou **entidades**, podem ter suas informações combinadas por meio de relacionamentos formados por chaves primárias, secundárias ou estrangeiras.

SQL Server e o Transact-SQL

No SQL Server usa-se o Transact-SQL. Esse é subdividido em três grupos de comandos.

DCL Data Control Language	DDL Data Definition Language	DML Data Manipulation Language
GRANT	CREATE	SELECT
DENY	ALTER	INSERT
REVOKE	DROP	UPDATE
		DELETE

Os comandos são relativamente simples, mas possuem cláusulas que podem torná-los complexos. Os comandos **DCL** tratando de permissões dos usuários dentro do banco de dados. Os **DDL** são para criação de objetos dentro do sistema. E os **DML** são utilizados no tratamento dos dados de um sistema.

Exercícios

1. O que é SQL?

Resposta:

2. O que é um banco de dados?

Resposta:

3. A linguagem SQL é subdividida em três grupos. Quais são e quais os principais comandos de cada grupo?

Resposta:

4. O que é uma entidade, atributo e registro?

Resposta:

Capítulo 2

Conhecendo o SQL Server 2005 Management Studio Express

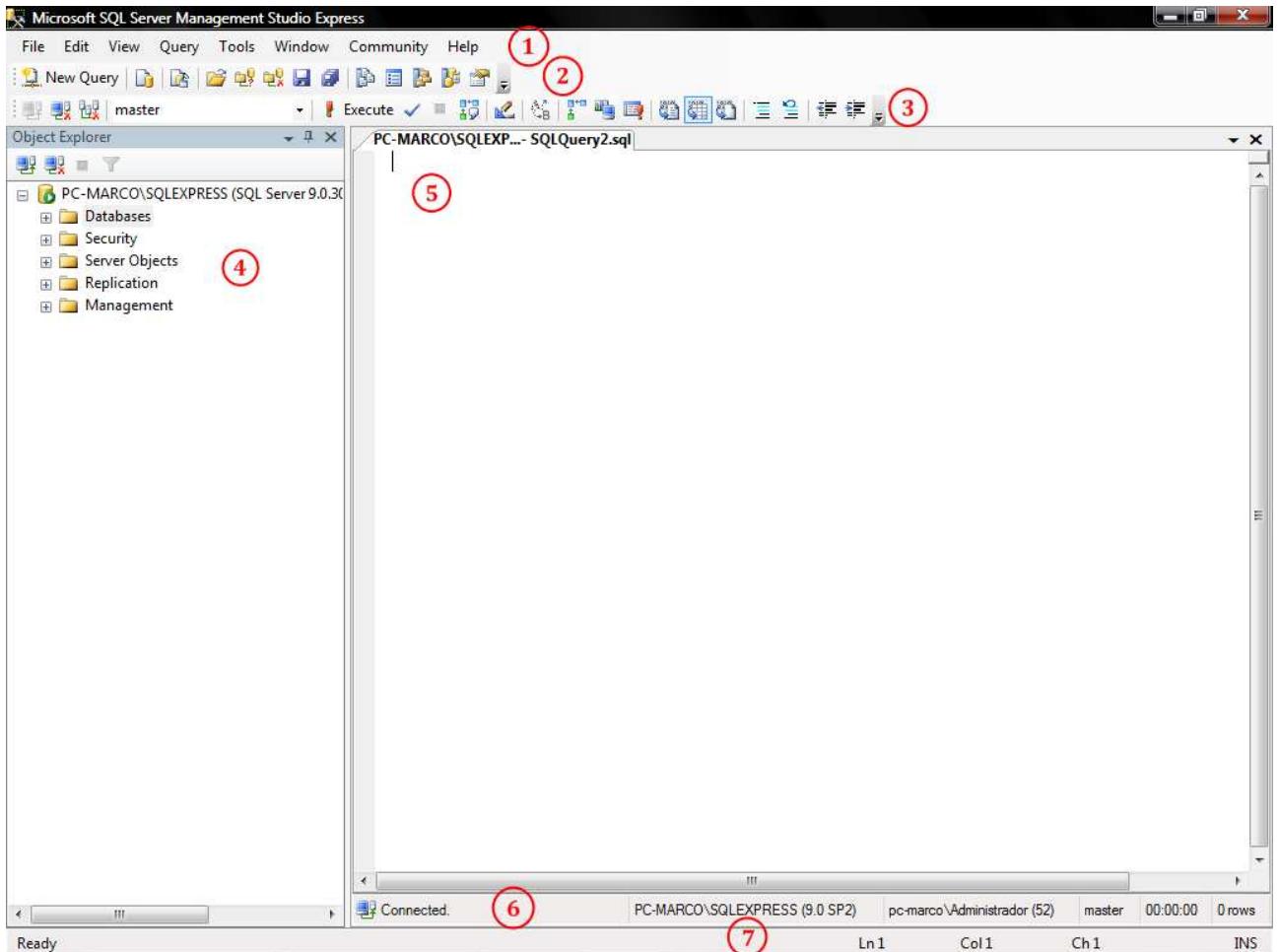
Antes de iniciarmos o nosso curso, vejamos a interface do programa que irá nos ajudar na montagem de um banco de dados no SQL Server 2005, esse programa é o SQL Server 2005 Management Studio Express ou simplesmente SSMSE. Será nossa IDE com o SQL Server.

Depois de instalar o SMSSE, para iniciá-lo, clique em INICIAR / PROGRAMAS / MICROSOFT SQL SERVER 2005 / SQL SERVER MANAGEMENT STUDIO EXPRESS.

Ao iniciar o programa, irá aparecer uma janela como esta abaixo pedindo para que o SSMSE seja conectado a alguma instância do SQL Server presente na máquina local ou em uma máquina remota. No nosso caso, a máquina será local e não será preciso fazer nenhum tipo de identificação agora, pois usaremos o modo **Windows Authentication** que nada mais é que usar o próprio usuário da máquina local como usuário do SQL Server Express.



Depois de conectar-se a instância do SQL Server, teremos essa tela abaixo.



- 1. Barra de Menu:** Aqui estão os principais menus do SSMSE.
- 2. Barra de Ferramentas:** Os comandos mais freqüentemente utilizados estão dispostos nessa região para um acesso rápido. Aqui também se encontra uma drop list chamada **Available Databases**, por meio da qual podemos escolher o banco de dados que desejamos acessar.
- 3. SQL Editor:** Nessa barra iremos encontrar os principais comandos do SQL Server para analisar nossos scripts SQL.
- 4. Object Explorer:** Encontra-se à esquerda. Aqui se localizam as databases, opções de segurança, objetos e serviços extras. Caso você feche o **Object Explorer**, basta clicar F8 para que ele reapareça.
- 5. Área do Editor:** Toda vez que criamos uma nova query, clicando no botão **New Query**, irá se abrir uma área para que possamos digitar nosso script.
- 6. Barra de Status do SQL Server:** Nessa barra de status temos as principais informações, como: Modo de trabalho, Instância, Usuário, Database, Tempo de execução e Linhas afetadas.
- 7. Barra de Status do Editor:** Nessa barra temos informações como: Posição do cursor (Horizontal e Vertical), Posição em relação ao caractere, etc.

Capítulo 3

Objetos do SQL Server 2005

Dentro do SQL Server há vários tipos de objetos. Abaixo estão listados alguns desses objetos:

- **DATABASE** - Os objetos de um sistema são criados dentro de uma estrutura lógica que corresponde ao objeto database.
- **TABLE** - Os dados de um sistema são inclusos nesse objeto de duas dimensões, que é formado por linhas e colunas.
- **CONSTRAINTs, DEFAULTs e RULEs** - Consistem em regras utilizadas para implementar a consistência e a integridade dos dados.
- **Datatypes e User-Defined Datatypes** - Os dados são armazenados no disco sob uma formato representado pelos datatypes. Um datatype deverá ser atribuído a cada coluna de uma tabela.
- **VIEW** - Este objeto oferece uma visualização lógica dos dados de uma tabela, de modo que diversas aplicações possam compartilhá-las.
- **INDEX** - Objetos responsáveis pela otimização de acesso aos dados de uma tabela.
- **PROCEDURE** - Nesse objeto, encontramos um bloco de comandos Transact-SQL, responsável por uma determinada tarefa. Sua lógica pode ser compartilhada por várias aplicações.
- **TRIGGER** - Esse objeto também possui um bloco de comandos Transact-SQL. O objeto **TRIGGER** é criado sobre uma tabela e ativado no momento da execução dos comandos **UPDATE, INSERT e/ou DELETE**.
- **FUNCTION** - Nesse objeto, encontramos um bloco de comando Transact-SQL responsável por uma determinada tarefa. Sua lógica pode ser compartilhada por várias aplicações. Vale ressaltar que uma função sempre retornará um valor.

Nota: Os objetos **PROCEDURE**, **TRIGGER** e **FUNCTION** são processados rapidamente, uma vez que seu código tende a ficar "compilado" na memória. Isso acontece porque tais objetos são executados no servidor de dados.

Manipulando e Conhecendo os Objetos DATATYPES, DATABASES e TABLES

DATATYPES

Cada coluna, expressão, parâmetro ou variável local do SQL Server é referenciado a um atributo que determina um tipo de dado. Por meio do MS .Net Framework ou do Transact-SQL, também é possível que os usuários definam seus próprios tipos de dados. Aqui veremos os tipos de dados, datatypes, padrões do SQL Server.

- **Built-in datatypes baseados em caracteres**

- **CHAR(n)**: Trata-se de um datatype que aceita como valor qualquer dígito, sendo que o espaço ocupado no disco é de um dígito por caractere. É possível utilizar até oito mil dígitos.
- **VARCHAR(n)**: Permite o uso de qualquer dígito, sendo que o espaço ocupado é de um dígito por caractere. Com valor máximo de oito mil dígitos.
- **TEXT**: Qualquer dígito por ser usado neste datatype, sendo ocupado um byte a cada caractere, o que corresponde a $2^{32} - 1$ byte ou 2.147.483.647 bytes.

- **Built-in datatypes baseados em caracteres UNICODE (internacionalização)**

- **NCHAR(n)**: Pode utilizar qualquer dígito, sendo ocupados dois bytes a cada caractere. É possível até oito mil bytes.
- **NVARCHAR(n)**: Aceita qualquer tipo de dígito. São ocupados dois bytes por caractere, sendo um máximo de oito mil bytes.
- **NTEXT**: Aceita qualquer dígito ocupando dois bytes por caractere. Máximo de $2^{30} - 1$ bytes ou 1.073.741.823 bytes.

- **Built-in datatypes baseados em numéricos inteiros**

- **BIGINT**: Aceita valores entre -2^{63} a $2^{63} - 1$, ocupando oito bytes.
- **INT**: Aceita valores entre -2^{31} a $2^{31} - 1$, ocupando quatro bytes.
- **SMALLINT**: Aceita valores entre -2^{15} a $2^{15} - 1$, ocupando dois bytes.
- **TINYINT**: Aceita valores entre 0 e 255, ocupando um byte.
- **BIT**: Trata-se de um tipo de dado integer (inteiro), cujo valor pode corresponder a NULL, 0 (zero) ou 1. Ocupa um bit.

- **Built-in datatypes baseados em numéricos de precisão**

- **DECIMAL(p, s)**: Aceita valores entre -10^{38} a $10^{38} - 1$, sendo que o espaço ocupado varia de acordo com a precisão. Se a precisão está entre 1 e 9, o espaço ocupado é de cinco bytes. Se a precisão está entre 10 e 19, o espaço ocupado é de nove bytes. Se a precisão está entre 20 e 28, o espaço ocupado é de treze bytes. Caso a precisão seja de 29 a 38, o espaço ocupado é de dezessete bytes.
- **NUMERIC(p, s)**: Idem ao **DECIMAL(p, s)**.

- **Built-in datatypes baseados em numéricos aproximados**

- **FLOAT(n)**: Esse datatype aceita valores entre -1.79×10^{308} e 1.79×10^{308} . O espaço ocupado varia de acordo com o valor de **n**. Se esse valor está entre 1 e 24, a precisão será de 7 dígitos, sendo que o espaço ocupado será de 4 bytes.

Caso o valor de n esteja entre 25 e 53, sua precisão será de 15 dígitos, fazendo com que sejam ocupados 8 bytes.

- **Built-in datatypes baseados em numéricos monetários**

- **MONEY**: Esse datatype aceita valores entre -2^{63} e $2^{63} - 1$ sendo que 8 bytes são ocupados para alocá-lo.
- **SMALLMONEY**: É possível utilizar valores entre -2^{31} e $2^{31} - 1$. Ocupa 4 bytes.
- Built-in datatypes baseados em data e hora
- **DATETIME**: Permite o uso de valores entre 1/1/1753 e 31/12/9999. Esse datatype ocupa 8 bytes, sendo que sua precisão atinge 333 milissegundos.
- **SMALLDATETIME**: Permite o uso de valores entre 1/1/1900 e 6/6/2079, sendo que sua precisão é de 1 minuto. O espaço ocupado é de 4 bytes.

- **Built-in datatypes baseados em binários**

- **BINARY(n)**: Este datatype representa os dados que são utilizados no formato binário. O espaço ocupado é de n+4 bytes, sendo que n pode variar entre 1 e 8000 bytes.
- **VARBINARY(n)**: Descrição idêntica ao **BINARY(n)**.
- **IMAGE**: O espaço ocupado por esse datatype é de 2.147.483.647 bytes ou 2 Gbytes.

- **Built-in datatypes especiais**

- **UNIQUEIDENTIFIER**: O formato hexadecimal é utilizado para armazenamento de dados binários. O espaço ocupado é de 16 bytes.
- **TIMESTAMP**: Um valor binário é gerado pelo SQL Server, sendo que esse datatype ocupa 8 bytes.
- **BIT**: Ocupa um bit. Representa 0 (zero), 1 ou **NULL**.
- **SQL_VARIANT**: O valor pode ser qualquer datatype. Pode armazenar valores de até 8016 bytes.

DATABASE

Os objetos que fazem parte de um sistema são criados dentro de um objeto denominado database, ou seja, uma estrutura lógica formada, basicamente, por dois tipos de arquivo, um responsável pelo armazenamento de dados e outro que armazena as transações feitas.

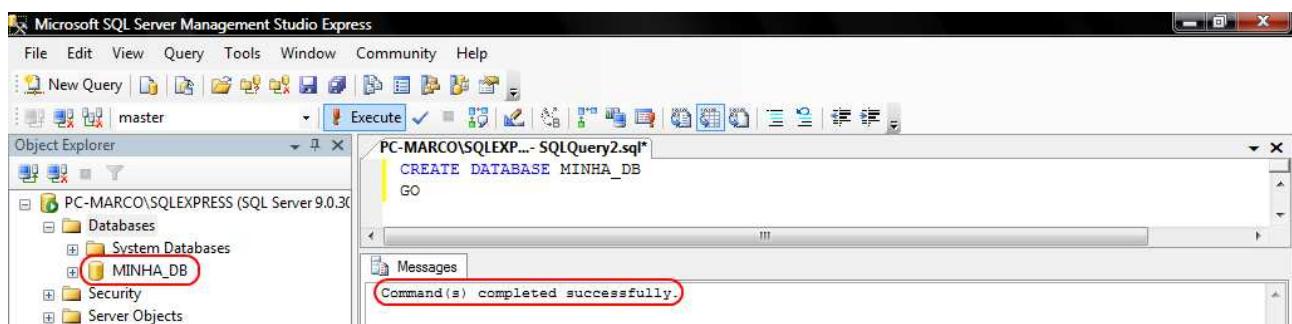
Para que o SQL Server crie um database, é necessário utilizar a instrução **CREATE DATABASE [nome_database]**. Por exemplo, suponhamos que será criado um banco de dados com o nome **MINHA_DB**. Assim, clique em **New Query**, isso irá abrir o editor. No editor digite esse comando abaixo:

```
CREATE DATABASE MINHA_DB  
GO
```



Após digitar, selecione o texto digitado e clique em **Execute** ou pressione a tecla **F5**. Isso irá executar somente o comando selecionado, caso existam outros comando abaixo esses não serão executados.

Agora se verificarmos o **Object Explorer**, no nó **Databases**, veremos que nossa database **MINHA_DB** foi criada com sucesso. (Caso a database **MINHA_DB** não esteja aparecendo, clique com o botão direito sobre o nó **Databases** e depois em **Refresh**)

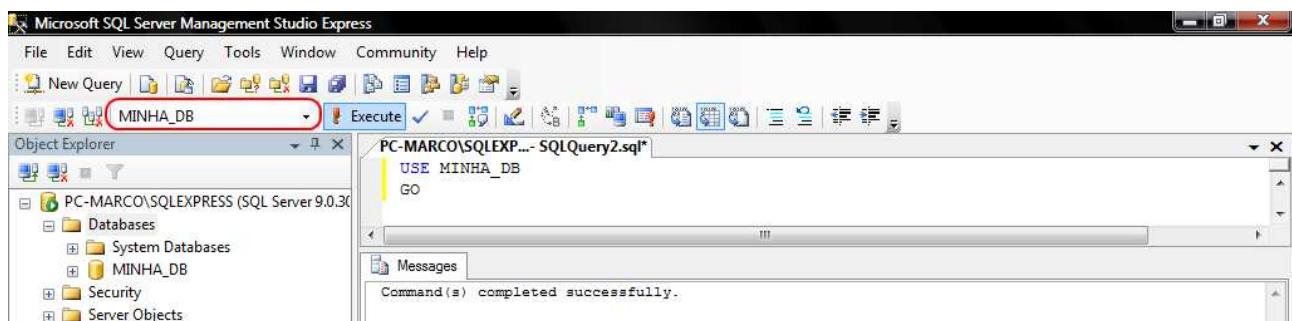


Há também uma aba de mensagens (**Messages**) abaixo do editor que mostrará as mensagens decorrentes das execuções de scripts. Nesse caso, ela mostra a mensagem "**Command(s) completed sucessfully.**" nos informando que o comando passado ao SQL Server foi completado com sucesso e criamos nossa primeira database. Veremos mais a frente que essa aba de mensagens, **Messages**, será muito útil quando ocorrer algum tipo de erro na nossa instrução SQL.

Apesar de termos criado nossa database, ela ainda não se encontra em uso para que possamos criar outros objetos dentro dela. Logo, para colocá-la em uso utilizaremos o comando **USE [nome_database]**, como vemos abaixo:

```
USE MINHA_DB
GO
```

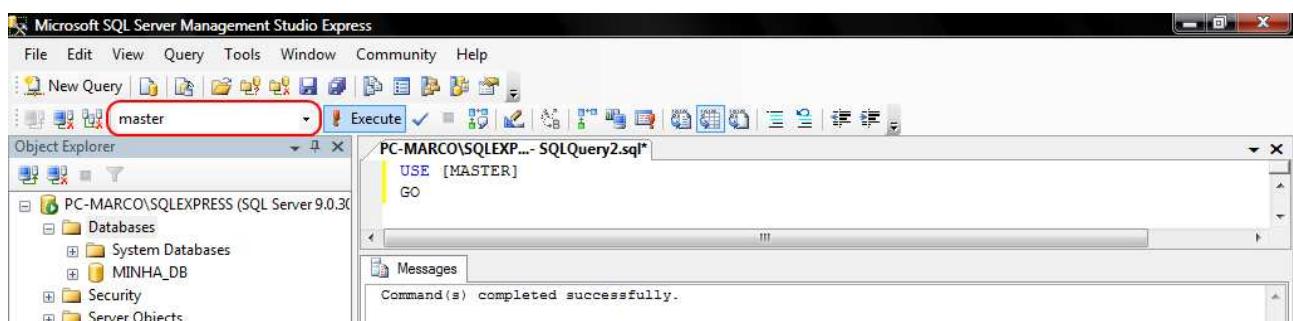
Agora nossa database está em uso, como podemos ver na lista de databases **Available Databases** (Databases Disponíveis):



Com esse exemplo, queríamos demonstrar os comandos básicos para a criação e utilização de uma database. Feito isso, nosso exemplo tem apenas mais uma serventia, demonstrar a utilização do comando **DROP DATABASE [nome_database]**. Esse comando exclui qualquer vestígio da nossa database de exemplo do SQL Server. Mas antes que ele possa ser executado, devemos colocar outra database em uso para que o SQL Server perca o vínculo com a database que desejamos excluir. Podemos colocar em uso a database **MASTER**, que é uma database padrão do SQL Server. Para isso, façamos:

```
USE [MASTER]
GO
```

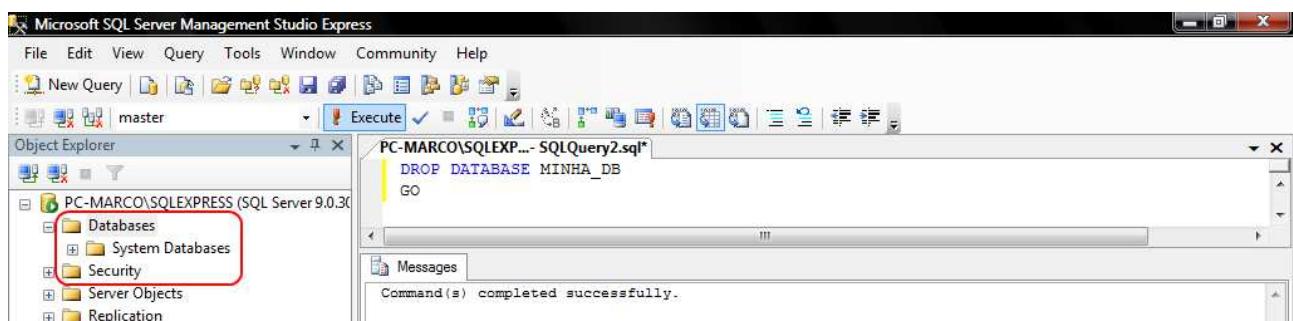
Como resposta ao comando acima, temos que nossa database em uso agora é a database **MASTER**.



Agora podemos usar o comando **DROP DATABASE [nome_database]** para "dropar" o nosso exemplo:

```
DROP DATABASE MINHA_DB
GO
```

Observe na imagem abaixo o resultado desse comando:



CUIDADO: AO UTILIZAR O COMANDO **DROP DATABASE [nome_database]!!! POIS ELE É IRREVERSÍVEL!**

TABLE

Os dados de um sistema são armazenados em objetos denominados tabelas. Cada uma das colunas de uma tabela refere-se a um atributo associado a uma determinada entidade. Os datatypes, ou seja, os formatos utilizados para a gravação dos dados no disco deverão ser especificados para cada coluna da tabela.

A instrução **CREATE TABLE**, que possui a seguinte sintaxe básica:

CREATE TABLE [nome_tabela]

(

[campo1] [datatype],

...,

[campoN] [datatype]

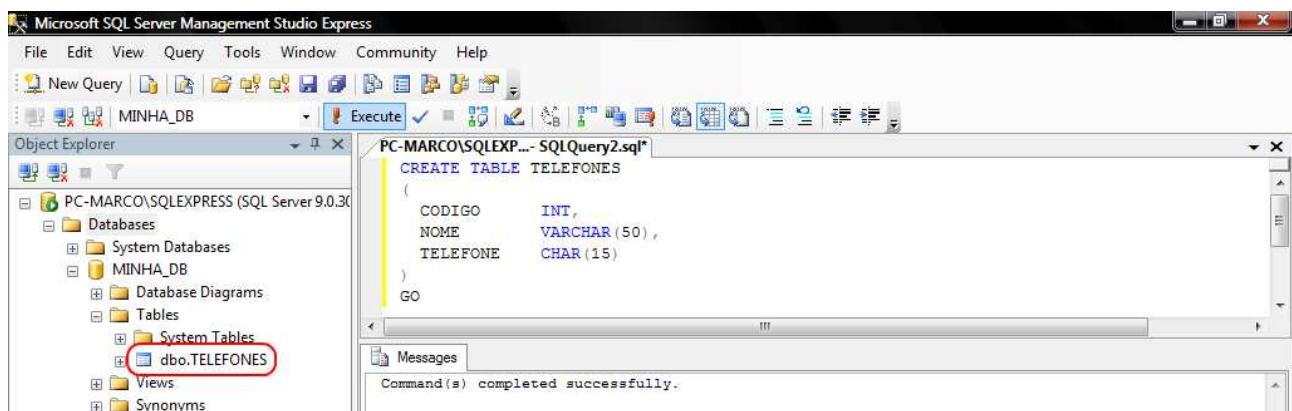
)

Como excluímos nossa database de exemplo, crie outra database e ponha-a em uso para que nela possamos criar nossa tabela, dê a database o nome de **MINHA_DB**, como no exemplo.

Para criar nossa tabela, façamos:

```
CREATE TABLE TELEFONES
(
    CODIGO          INT,
    NOME            VARCHAR(50),
    TELEFONE        CHAR(15)
)
GO
```

Quando executarmos esse comando, nossa tabela **TELEFONES** será criada.



Para localizar a tabela, no **Object Explorer**, expanda o nó **Databases**, localize a nossa database **MINHA_DB**, expanda-a. Dentro da database **MINHA_DB** existem pastas onde ficam

alocados todos os objetos referentes a essa database. Procure a pasta **Tables** e expanda-a. Lá, você localizará a tabela **TELEFONES** que acabamos de criar.

O comando **DROP TABLE [nome_tabela]** executa um comando parecido com o comando que executamos para excluir a nossa database de exemplo, mas ao invés de excluir a database inteira, ela exclui apenas a tabela a qual fazemos referência. É necessário salientar que esse comando **exclui a tabela permanentemente**.

A partir de agora já podemos inserir registros em nossa tabela recém criada. Nesse caso, não é necessário usar o comando **USE** para que possamos ter acesso à tabela, pois como ela está dentro da database **MINHA_DB**, o SQL Server já entende que qualquer comando direcionado a essa tabela admite que a database em que ela se encontra já está em uso.

Exercícios

1. Qual o comando usado para se criar uma database?

Resposta:

2. Como podemos apagar uma database do SQL Server?

Resposta:

3. Apesar de se excluir uma database, é possível recuperá-la?

Resposta:

4. Como podemos criar uma tabela?

Resposta:

5. O que é um datatype?

Resposta:

Capítulo 4

DML Simples

Os comandos **DML** são utilizados para manipular os dados dentro de uma tabela. Essa manipulação pode ser do tipo: Inclusão (**INSERT**), atualização (**UPDATE**), exclusão (**DELETE**) ou seleção (**SELECT**). Dentre esses comandos, o mais complexo, devido a inúmeras quantidades de cláusulas, é o comando de seleção (**SELECT**). Pois ele além de ter suas próprias cláusulas, pode entrar como cláusula dos comandos de inclusão, atualização e exclusão. (Será visto mais adiante).

INSERT

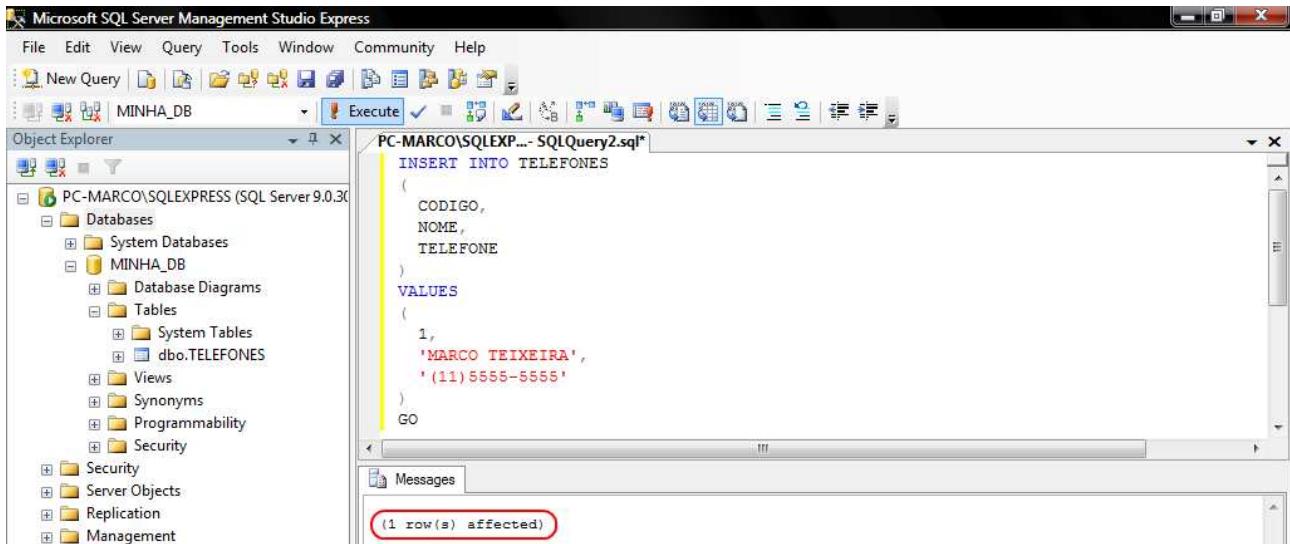
Primeiramente veremos o comando **INSERT**. Que nos ajudará a inserir registros em nossa tabela recém criada. O comando **INSERT** segue a seguinte sintaxe:

INSERT INTO [nome_tabela] ([nome_campo1], ..., [nome_campoN]) VALUES ([valor1], ..., [valorN])

Como exemplo prático, tomemos o seguinte comando dentro do SSMSE:

```
INSERT INTO TELEFONES
(
    CODIGO,
    NOME,
    TELEFONE
)
VALUES
(
    1,
    'MARCO TEIXEIRA',
    '(11) 5555-5555'
)
GO
```

Isso irá inserir o primeiro registro na nossa tabela **TELEFONES**.



The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'MINHA_DB' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery2.sql' contains the following SQL code:

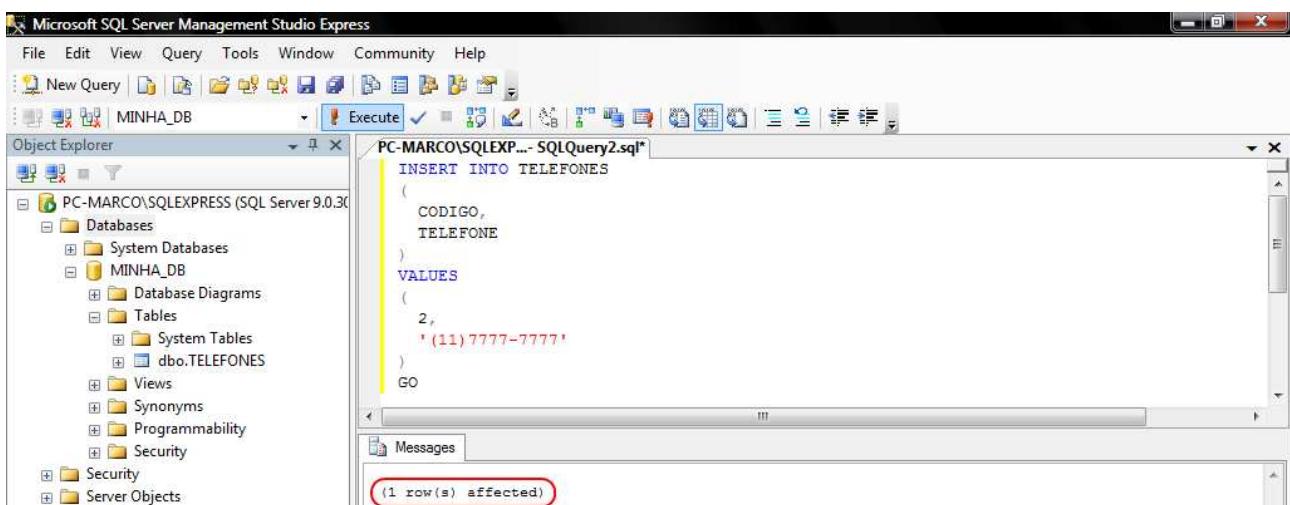
```
INSERT INTO TELEFONES
(
    CODIGO,
    NOME,
    TELEFONE
)
VALUES
(
    1,
    'MARCO TEIXEIRA',
    '(11) 5555-5555'
)
GO
```

In the 'Messages' pane at the bottom, the output is shown in a red box: '(1 row(s) affected)'.

Observando na imagem acima, podemos perceber que nossa aba **Messages** diz o seguinte: **(1 row(s)) affected**, ou seja, "uma linha afetada". Isso nos mostra que o registro foi inserido com sucesso. Quando executarmos os comandos **UPDATE** e **DELETE**, caso esses sejam bem sucedidos, uma mensagem similar aparecerá na aba **Messages**. Basicamente o comando **INSERT** é isso, mas é possível modificarmos um pouco esse comando. Existe a possibilidade de se fazer um **INSERT posicional**. Isso significa que podemos escolher quais campos serão preenchidos pelo **INSERT**. Vejamos:

```
INSERT INTO TELEFONES
(
    CODIGO,
    TELEFONE
)
VALUES
(
    2,
    '(11) 7777-7777'
)
GO
```

Utilizando o comando:



The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'MINHA_DB' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery2.sql' contains the following SQL code:

```
INSERT INTO TELEFONES
(
    CODIGO,
    TELEFONE
)
VALUES
(
    2,
    '(11) 7777-7777'
)
GO
```

In the 'Messages' pane at the bottom, the output is shown in a red box: '(1 row(s) affected)'.

Nesse caso inserimos apenas o código e o telefone da pessoa. Ficou sem o nome. Isso só é possível se o campo permitir, ou seja, se nele não contiver nenhuma restrição ao **NOME** ser um valor nulo. Veremos mais a frente como criar essa restrição.

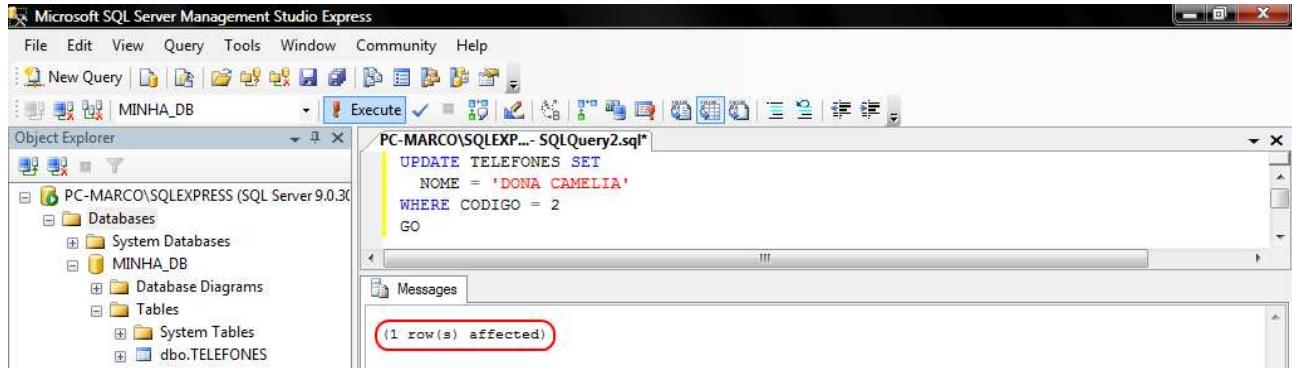
UPDATE

Os dados pertencentes a múltiplas linhas de uma tabela podem ser alterados por meio do comando **UPDATE**. O comando **UPDATE** tem a seguinte sintaxe:

```
UPDATE [nome_tabela] SET [nome_campo1] = [valor1], ..., [nome_campoN] = [valorN]
WHERE [nome_campo] = [valor]
```

Supondo que desejamos atualizar o registro '2' da nossa tabela de telefones e incluir o nome da pessoa que está faltando. Vejamos como:

```
UPDATE TELEFONES SET
    NOME = 'DONA CAMELIA'
WHERE CODIGO = 2
GO
```



Ao executarmos esse comando o registro '2' da tabela telefones será atualizado com o **NOME = 'DONA CAMELIA'**. Para que o SQL Server entenda que estamos querendo atualizar o registro '2', precisamos informá-lo. Essa informação vem depois da cláusula **WHERE**.

CUIDADO: CASO A CLÁUSULA WHERE SEJA OMITIDA NO COMANDO UPDATE, ISSO FARÁ COM QUE TODOS OS REGISTROS DA TABELA SEJAM ATUALIZADOS COM O MESMO VALOR.

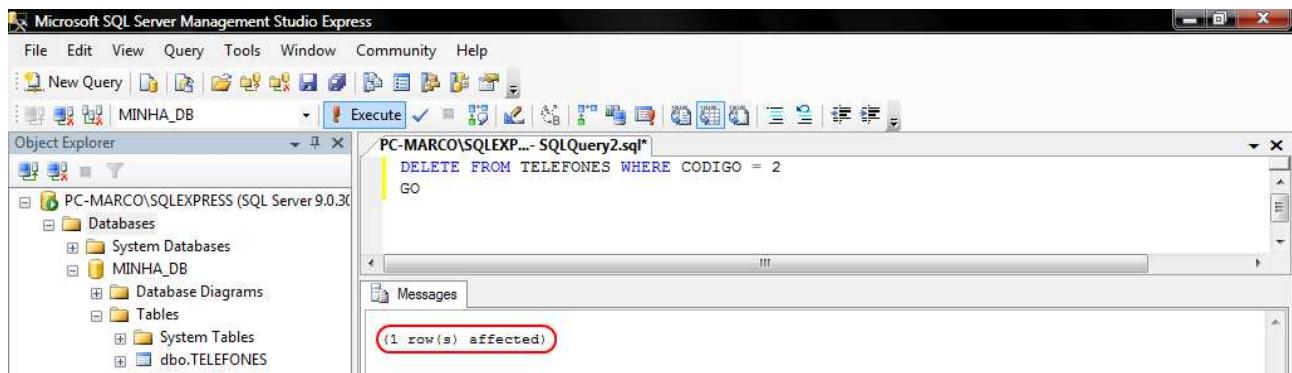
DELETE

O comando **DELETE** deve ser utilizado quando desejamos eliminar os dado de uma tabela. Para isso, temos o seguinte comando:

DELETE FROM [nome_tabela] WHERE [nome_campo] = [valor]

Dessa forma iremos eliminar o registro '2' da nossa tabela **TELEFONES**:

```
DELETE FROM TELEFONES WHERE CODIGO = 2  
GO
```



A aba **Messages** nos informa que o comando foi executando com sucesso e o registro foi eliminado da tabela. Da mesma forma que a cláusula **WHERE** foi utilizada no comando **UPDATE**, ela será utilizada no comando **DELETE**.

CUIDADO: CASO A CLÁUSULA WHERE SEJA OMITIDA, ISSO FARÁ COM QUE TODOS OS REGISTROS DA TABELA MENCIONADA SEJAM ELIMINADOS.

SELECT

Talvez o comando **SELECT** seja o mais importante comando dentro dos **DMLs**. É utilizado para realizar consultas a dados pertencentes a uma tabela. Por meio dele, podemos retornar dados para outros comandos SQL e, também, para outras aplicações. A sintaxe básica do comando é:

SELECT [nome_campo1], ..., [nome_campoN] FROM [nome_tabela]

Vejamos como isso ficaria aplicado à nossa tabela **TELEFONES**:

```
SELECT CODIGO, NOME, TELEFONE FROM TELEFONES  
GO
```

Como podemos ver, ao executar o comando de **SELECT** nos campos especificados, o retorno foi uma linha da nossa tabela **TELEFONES**. Podemos ver o retorno ao observar a nova aba que apareceu, a aba **Results**. Ela nos mostra o resultado de um comando de seleção ou comandos que tenham relação ao comando de seleção. Nesse caso, o retorno foi apenas uma linha, pois quando vimos o comando **DELETE** havíamos excluído o registro '2'.

Exibindo Todas as Colunas da Tabela

É possível realizar um comando de seleção que retorne todas as colunas de uma tabela. Para isso basta que se substituam os nomes dos campos da tabela pelo caractere asterisco (*). Vejamos:

```
SELECT * FROM TELEFONES
GO
```

Esse comando nos retornará o mesmo resultado do anterior, mostrará as três colunas da tabela **TELEFONES** e os registros que ela contém. No caso anterior, nos especificamos as três colunas que queríamos como retorno.

Exibindo Colunas com ALIAS

ALIAS são as legendas das colunas. Na nossa tabela as **alias** das colunas são **CODIGO**, **NOME**, **TELEFONE**. Mas podemos por meio da instrução **SELECT** modificar essas legendas temporariamente, para efeito de relatório. Vejamos como seria a sintaxe do comando:

```
SELECT [nome_campo1] AS [nome_alias1], ..., [nome_campoN] AS [nome_aliasN] FROM
[nome_tabela]
```

Aplicando isso no editor do SSMSE teríamos esse resultado:

```
SELECT
    CODIGO AS [Código da Linha],
    NOME AS [Nome da Pessoa],
    TELEFONE AS [Telefone da Pessoa]
FROM TELEFONES
GO
```

```

SELECT
    CODIGO AS [Código da Linha],
    NOME AS [Nome da Pessoa],
    TELEFONE AS [Telefone da Pessoa]
FROM TELEFONES
GO

```

Código da Linha	Nome da Pessoa	Telefone da Pessoa
1	MARCO TEIXEIRA	(11)5555-5555

Se observarmos a aba **Results**, veremos que as legendas das colunas foram alteradas. Isso é interessante caso os nomes dos campos sejam muito longos e foram abreviados por medida de normalização. Assim, quando for necessário emitir um relatório, por exemplo, podemos modificar novamente a legenda das colunas da forma que elas deveriam ter originalmente.

No caso de utilizarmos espaços em branco entre as palavras na alias, será necessário utilizar colchetes para conter os nomes. Dessa forma: **[nome da alias com espaços em branco]**.

Colunas Virtuais

Por meio da instrução **SELECT**, podemos exibir colunas existentes em uma tabela e também criar colunas virtuais que não existem fisicamente na tabela. Essas colunas virtuais serão preenchidas com dados que serão passados por nós. A seguir vamos mostrar a coluna **AMIGO** com o conteúdo '**SIM**' em nosso **SELECT**. Atente para o fato que a coluna **AMIGO** não existe na tabela **TELEFONES**.

```

SELECT
    CODIGO AS [Código da Linha],
    NOME AS [Nome da Pessoa],
    TELEFONE AS [Telefone da Pessoa],
    'Sim' AS [Amigo]
FROM TELEFONES
GO

```

```

SELECT
    CODIGO AS [Código da Linha],
    NOME AS [Nome da Pessoa],
    TELEFONE AS [Telefone da Pessoa],
    'Sim' AS [Amigo]
FROM TELEFONES
GO

```

Código da Linha	Nome da Pessoa	Telefone da Pessoa	Amigo
1	MARCO TEIXEIRA	(11)5555-5555	Sim

Observe a aba **Results**. Você perceberá que ela mostra a coluna **AMIGO**, mas essa coluna não existe na nossa tabela **TELEFONES**.

Exibindo um Cálculo com o Comando SELECT

Nem sempre é necessário gravar um resultado de um cálculo para que esse seja exibido na tabela ou retornado a alguma aplicação. Imagine que a partir do **CÓDIGO** da linha de um registro, você deseja aplicar uma soma, multiplicação, divisão ou subtração. No nosso caso, usaremos o campo **CÓDIGO** e iremos multiplicá-lo pelo valor de π (pi). Poderemos fazer isso da seguinte forma:

```
SELECT
    CODIGO AS [Código da Linha],
    NOME AS [Nome da Pessoa],
    TELEFONE AS [Telefone da Pessoa],
    CODIGO * 3.141692 AS [Código vezes PI]
FROM TELEFONES
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the center, there is a query editor window titled "PC-MARCO\SQLEXP... - SQLQuery2.sql". The query is:`SELECT
 CODIGO AS [Código da Linha],
 NOME AS [Nome da Pessoa],
 TELEFONE AS [Telefone da Pessoa],
 CODIGO * 3.141692 AS [Código vezes PI]
FROM TELEFONES
GO`

Below the query editor is a "Results" grid. It has four columns: "Código da Linha", "Nome da Pessoa", "Telefone da Pessoa", and "Código vezes PI". There is one row of data: "1", "MARCO TEIXEIRA", "(11)5555-5555", and "3.141692". The last column, "Código vezes PI", is highlighted with a red rectangle.

Observando novamente a aba **Results**, vemos que agora existe uma coluna virtual com o resultado da multiplicação. Se existisse um segundo registro, ele iria mostrar o cálculo para ele também, por exemplo, $2 * 3.141692 = 6.283384$.

Exibindo Apenas Uma Vez Registros Repetidos (DISTINCT)

Para que possamos observar a cláusula **DISTINCT**, iremos adicionar um novo registro a nossa tabela. Mais precisamente um registro idêntico a qualquer um existente nela. Façamos:

```
INSERT INTO TELEFONES
(
    CODIGO,
    NOME,
    TELEFONE
)
```

```

VALUES
(
    2,
    'MARCO TEIXEIRA',
    '(11) 5555-5555'
)
GO

```

Isso irá inserir um novo registro com o mesmo nome na nossa tabela **TELEFONES**.

CODIGO	NOME	TELEFONE
1	MARCO TEIXEIRA	(11)5555-5555
2	MARCO TEIXEIRA	(11)5555-5555

Utilizando a cláusula **DISTINCT**:

```

SELECT DISTINCT NOME, TELEFONE FROM TELEFONES
GO

```

NOME	TELEFONE
MARCO TEIXEIRA	(11)5555-5555

O comando **SELECT** mostra apenas um resultado, quando na nossa tabela existem dois registros idênticos. Nesse caso fizemos uso dos campos **NOME** e **TELEFONE** no comando **SELECT**, pois eram nesses campos que os registros estavam duplicados. Caso utilizemos o campo **CODIGO**, os dois registros serão retornados. Vejamos:

```

SELECT DISTINCT CODIGO, NOME, TELEFONE FROM TELEFONES
GO

```

CODIGO	NOME	TELEFONE
1	MARCO TEIXEIRA	(11)5555-5555
2	MARCO TEIXEIRA	(11)5555-5555

Copiando Dados De Uma Tabela Para Outra Com INSERT e SELECT

É possível copiar dados de uma tabela para outra em alguns casos. Um desses casos é quando a tabela origem apresenta a mesma estrutura da tabela destino. O outro é quando a tabela origem não apresenta a mesma estrutura que a tabela destino, ou vice-versa.

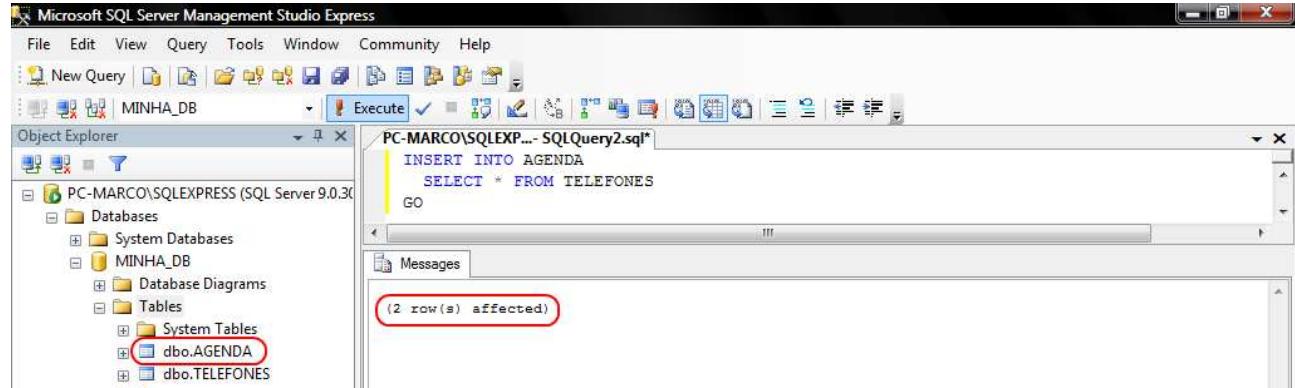
INSERT Com SELECT Quando as Tabelas São Iguais

Iniciemos criando uma nova tabela com a mesma estrutura da tabela **TELEFONES**, só que o nome dessa tabela será **AGENDA**.

```
CREATE TABLE AGENDA
(
    CODIGO      INT,
    NOME        VARCHAR(50),
    TELEFONE    CHAR(15)
)
GO
```

Como os dados que iremos inserir na tabela **AGENDA** já estão disponíveis na tabela **TELEFONES**, iremos usar uma instrução de **INSERT** com uma instrução de **SELECT** para preenchê-la. E como ambas possuem a mesma estrutura, usaremos a marcação asterisco (*) na instrução **SELECT** para indicar que queremos que todos os campos sejam copiados de uma tabela a outra. Vejamos:

```
INSERT INTO AGENDA
    SELECT * FROM TELEFONES
GO
```



Observe na aba **Messages** a mensagem "**(2 row(s) affected)**". Isso indica-nos que as linhas da tabela **TELEFONES** foram inseridas em **AGENDA**. Execute um **SELECT** na tabela **AGENDA** para conferir o resultado.

Nesse caso em especial, só foi possível copiar dessa forma o conteúdo da tabela **TELEFONES** para **AGENDA** pois as duas tabelas tinham a mesma estrutura, mesma quantidade de campos e mesmos datatypes.

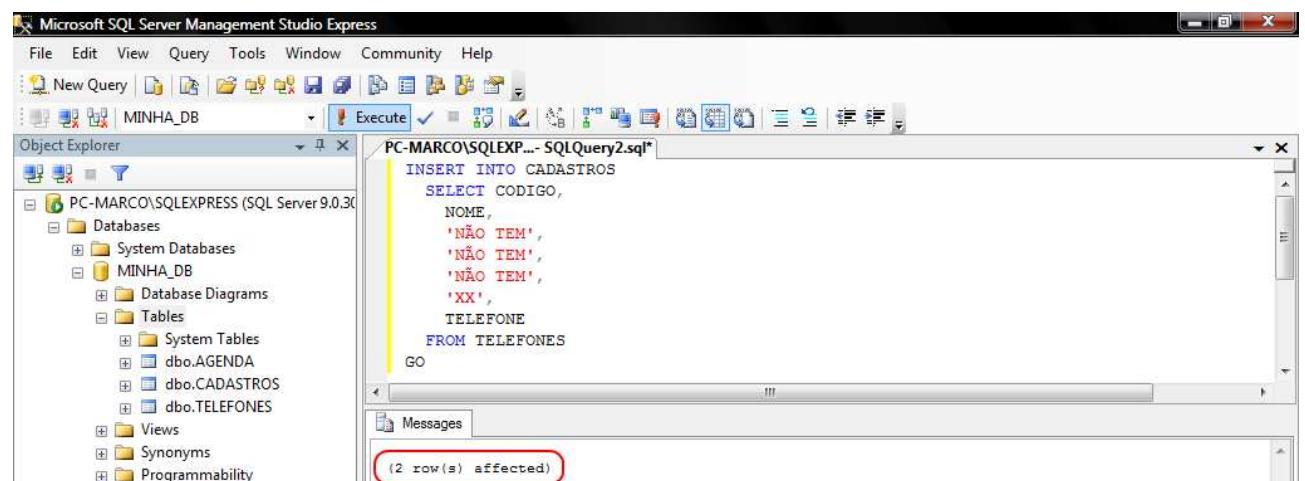
INSERT Com SELECT Quando as Tabelas Não São Iguais

Para que possamos observar esse tipo de comando, iremos criar uma nova tabela chamada **CADASTROS**. Segue código abaixo:

```
CREATE TABLE CADASTROS
(
    CODIGO      INT,
    NOME        VARCHAR(50),
    ENDERECO    VARCHAR(50),
    BAIRRO      VARCHAR(25),
    CIDADE      VARCHAR(25),
    ESTADO      CHAR(2),
    TELEFONE    CHAR(15)
)
GO
```

Embora a tabela **TELEFONES** contenha dados, nem todas as colunas de **CADASTROS** podem ser encontradas nela. Nesse caso, para copiar os dados de **TELEFONES** para **CADASTROS** utilizaremos o **SELECT** de colunas virtuais e o **INSERT** posicional (Após, faça um **SELECT** na tabela **CADASTROS** para ver o resultado). Vejamos:

```
INSERT INTO CADASTROS
    SELECT CODIGO,
           NOME,
           'NÃO TEM',
           'NÃO TEM',
           'NÃO TEM',
           'XX',
           TELEFONE
    FROM TELEFONES
GO
```



Faça um **SELECT** na tabela **CADASTROS** e observe o resultado.

Exercícios

1. Para que usamos os comandos **DML**?

Resposta:

2. Qual a função da cláusula **DISTINCT**?

Resposta:

3. Qual sintaxe devemos utilizar para copiar todos os dados de uma tabela para outra quando essas possuem a mesma estrutura?

Resposta:

4. E qual sintaxe usar quando as tabelas não tem a mesma estrutura?

Resposta:

5. Para que utilizar os comando **UPDATE** e **SELECT**?

Resposta:

Laboratório

1. Crie uma database chamada **ZOOLOGICO**.

Resposta:

2. Na database **ZOOLOGICO**, criei uma tabela chamada **ANIMAIS** com os seguintes campos:

campo	CODIGO	TIPO	NOME	IDADE	VALOR
datatype	INT	CHAR(15)	CHAR(30)	TINYINT	DECIMAL(10,2)
valor	1	cachorro	Djudi	3	300.00
valor	2	cachorro	Sula	5	300.00
valor	3	cachorro	Sarina	7	300.00
valor	4	gato	Pipa	2	500.00
valor	5	gato	Sarangue	2	500.00
valor	6	gato	Clarences	1	500.00
valor	7	coruja	Agnes	0	700.00
valor	8	coruja	Arabela	1	700.00
valor	9	sapo	Quash	1	100.00
valor	10	peixe	Fish	0	100.00

Resposta:

3. Escreva um comando que exiba todas as colunas e todas as linhas da tabela **ANIMAIS**.

Resposta:

4. Escreva um comando que exiba apenas as colunas **TIPO** e **NOME** da tabela **ANIMAIS**, apresentando todos os registros da tabela.

Resposta:

5. Escreva um comando que exiba apenas as colunas **TIPO**, **NOME** e **IDADE** da tabela **ANIMAIS**, apresentando todos os registros.

Resposta:

6. Escreva um comando que exiba apenas as colunas **TIPO** e **NOME** da tabela **ANIMAIS** apresentando todos os registros. Apresente a legenda da coluna **TIPO** com o alias **[Tipo de Animal]** e a coluna nome com o alias **[Nome do Animal]**.

Resposta:

7. Escreva um comando que exiba apenas as colunas **TIPO**, **NOME** e **IDADE** da tabela **ANIMAIS** apresentando todos os registros. Apresente a legenda da coluna **TIPO** com o alias **[Tipo de Animal]**, da coluna nome com o alias **[Nome do Animal]** e da coluna **IDADE** com o alias **[Tempo de Vida]**.

Resposta:

8. Escreva um comando que apresente os dados da tabela **ANIMAIS** da seguinte forma:

Procedência	Tipo	Nome	Tempo de Vida
Animal Doméstico	cachorro	Djudi	3
Animal Doméstico	cachorro	Sula	5
Animal Doméstico	cachorro	Sarina	7
Animal Doméstico	gato	Pipa	2
Animal Doméstico	gato	Sarangue	2
Animal Doméstico	gato	Clarences	1
Animal Doméstico	coruja	Agnes	0
Animal Doméstico	coruja	Arabela	1
Animal Doméstico	sapo	Quash	1
Animal Doméstico	peixe	Fish	0

Resposta:

9. Escreva um comando que apresente os dados da tabela **ANIMAIS** da seguinte forma:

Tipo	Nome	Idade	Valor Original	Valor de Venda
cachorro	Djudi	3	300.00	330.00
cachorro	Sula	5	300.00	330.00
cachorro	Sarina	7	300.00	330.00
gato	Pipa	2	500.00	550.00
gato	Sarangue	2	500.00	550.00
gato	Clarences	1	500.00	550.00
coruja	Agnes	0	700.00	770.00
coruja	Arabela	1	700.00	770.00
sapo	Quash	1	100.00	110.00
peixe	Fish	0	100.00	110.00

Resposta:

10. Escreva um comando que apresente os dados da tabela **ANIMAIS** como vemos a seguir, apresentando uma vez os dados repetidos.

Tipo	Valor Original	Valor de Venda
cachorro	300.00	330.00
gato	500.00	550.00
coruja	700.00	770.00
sapo	100.00	110.00
peixe	100.00	110.00

Resposta:

Capítulo 5

A Cláusula WHERE

Vimos no capítulo anterior um pouco dessa cláusula, nesse capítulo iremos nos aprofundar na forma como ela trata os dados. A utilidade da cláusula **WHERE** é determinar quais os dados de uma tabela que serão afetados pelos comandos **SELECT**, **UPDATE** ou **DELETE**. Podemos dizer, então, que essa cláusula determina o escopo de uma consulta a algumas linhas, realizando a filtragem dos dados que estejam de acordo com as condições definidas.

Para que possamos fazer um uso máximo dessa cláusula, iremos criar uma nova database, tabela e registros. Segue script:

```
CREATE TABLE PRODUTOS
(
    CODIGO      INT,
    NOME        VARCHAR(50),
    TIPO        VARCHAR(25),
    QUANTIDADE  INT,
    VALOR       DECIMAL(10, 2)
)
GO

INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 1,
'IMPRESSORA', 'INFORMATICA', 200, 600.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 2,
'CAMERA DIGITAL', 'DIGITAIS', 300, 400.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 3, 'DVD
PLAYER', 'ELETRONICOS', 250, 500.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 4,
'MONITOR', 'INFORMATICA', 400, 900.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 5,
'TELEVISOR', 'ELETRONICOS', 350, 650.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 6,
'FILMADORA DIGITAL', 'DIGITAIS', 500, 700.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 7,
'CELULAR', 'TELEFONE', 450, 850.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 8,
'TECLADO', 'INFORMATICA', 300, 450.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 9,
'VIDEOCASSETE', 'ELETRONICOS', 200, 300.00 )
INSERT INTO PRODUTOS ( CODIGO, NOME, TIPO, QUANTIDADE, VALOR ) VALUES ( 10,
'MOUSE', 'INFORMATICA', 400, 60.00 )
GO
```

Sintaxe Básica

A sintaxe básica de qualquer comando que utilize a cláusula **WHERE** é a seguinte:

[comando] WHERE [nome_campo] [operador_comparacao] [valor]

Os operadores de comparação podem ser: = (igual à), > (maior que), < (menor que), <> (diferente de), >= (maior ou igual à), <= (menor ou igual à). Há a possibilidade de se usar os operadores **IN**, **AND** e **OR** em conjunto com a cláusula **WHERE**, mas veremos isso mais a frente.

Selecionando Registros Específicos

Com a cláusula **WHERE**, podemos fazer uma seleção de registros específicos. Suponha que queremos retornar apenas os produtos os quais o **TIPO** é 'INFORMATICA'. Vejamos:

```
SELECT * FROM PRODUTOS WHERE TIPO = 'INFORMATICA'  
GO
```

O resultado que iremos obter será:

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, a database named 'MINHA_DB' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery2.sql' contains the following code:

```
SELECT * FROM PRODUTOS WHERE TIPO = 'INFORMATICA'  
GO
```

The results pane displays a table with four columns: CODIGO, NOME, TIPO, and QUANTIDADE. The 'TIPO' column is highlighted with a red border. The data is as follows:

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	600.00
2	MONITOR	INFORMATICA	400	900.00
3	TECLADO	INFORMATICA	300	450.00
4	MOUSE	INFORMATICA	400	60.00

Faça um teste e selecione apenas os registros com **QUANTIDADE** igual à 200.

Atualizando Registros Específicos

Para realizar uma alteração sobre os preços dos produtos desta tabela, utilizaremos a cláusula **WHERE**. Vejamos um exemplo que demonstra um acréscimo de 15% sobre os valores dos produtos do tipo 'INFORMATICA'.

```
UPDATE PRODUTOS SET VALOR = VALOR * 1.15  
WHERE TIPO = 'INFORMATICA'  
GO
```

Executando o comando, podemos observar que quatro registros foram alterados. Faça um **SELECT** somente nos itens os quais o tipo seja 'INFORMATICA' para visualizar a alteração.

A Cláusula WHERE Com os Operadores AND e OR

Os operadores lógicos **AND** e **OR** são utilizados em conjunto com a cláusula **WHERE** nas situações em que é necessário empregar mais de uma condição de comparação. Supondo um cenário, poderíamos elevar o valor dos produtos do tipo 'ELETRONICOS' em 5% cujo valor seja inferior a 600,00. Vejamos com isso seria usando o operador **AND**:

```
UPDATE PRODUTOS SET VALOR = VALOR * 1.05  
    WHERE TIPO = 'ELETRONICOS' AND VALOR < 600.00  
GO
```

Utilize um **SELECT** com a cláusula **WHERE** para retornar somente os produtos do tipo 'ELETRONICOS' e veja as alterações.

Para que vejamos o **OR** em operação, imaginemos que vamos diminuir o valor dos produtos, em 20%, que sejam maiores que 1000,00 ou que estejam do tipo 'DIGITAIS'. Então teríamos o seguinte:

```
UPDATE PRODUTOS SET VALOR = VALOR * 0.80  
    WHERE TIPO = 'DIGITAIS' OR VALOR > 1000.00  
GO
```

Utilize esse comando e veja qual o resultado obtido.

A cláusula WHERE com o operador IN

O operador **IN** poder ser utilizado em lugar do operador **OR** em determinadas situações. O **IN** permite verificar se o valor de uma coluna está presente em uma lista de elementos.

Considerando a coluna **TIPO**, da tabela **PRODUTOS**, podemos usar o **IN** ou **OR** para selecionar os produtos do tipo 'INFORMATICA' ou 'DIGITAIS'. Vejamos:

```
SELECT * FROM PRODUTOS WHERE TIPO IN ('INFORMATICA', 'DIGITAIS')  
GO
```

O comando **SELECT** equivalente, usando o operador **OR** seria:

```
SELECT * FROM PRODUTOS WHERE TIPO = 'INFORMATICA' OR TIPO = 'DIGITAIS'  
GO
```

Observe que a vantagem dessa forma de escrita, com o operador **IN**, é a legibilidade. Ao executarmos ambos os comandos a saída será a mesma. Vejamos com o operador **IN**:

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'MINHA_DB' is selected. In the Results pane, a query is being run:

```
PC-MARCO\SQLEXP... - SQLQuery2.sql
SELECT * FROM PRODUTOS WHERE TIPO IN ('INFORMATICA', 'DIGITAIS')
GO
```

The results show a table of products categorized as INFORMATICA or DIGITAIS. The 'TIPO' column is highlighted with a red box.

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	690.00
2	CAMERA DIGITAL	DIGITAIS	300	320.00
3	MONITOR	INFORMATICA	400	828.00
4	FILMADORA DIGITAL	DIGITAIS	500	560.00
5	TECLADO	INFORMATICA	300	517.50
6	MOUSE	INFORMATICA	400	69.00

A Cláusula WHERE Com os Operadores NOT IN

Os operadores **NOT IN**, ao contrário do **IN**, permitem obter como resultados o valor de uma coluna que não pertence a uma determina lista de elementos, conforme podemos observar a seguir:

```
SELECT * FROM PRODUTOS WHERE TIPO NOT IN ('INFORMATICA', 'DIGITAIS')
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'MINHA_DB' is selected. In the Results pane, a query is being run:

```
PC-MARCO\SQLEXP... - SQLQuery2.sql
SELECT * FROM PRODUTOS WHERE TIPO NOT IN ('INFORMATICA', 'DIGITAIS')
GO
```

The results show a table of products categorized as ELETRONICOS, TELEFONE, or VIDEOCASSETE. The 'TIPO' column is highlighted with a red box.

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	DVD PLAYER	ELETRONICOS	250	525.00
2	TELEVISOR	ELETRONICOS	350	650.00
3	CELULAR	TELEFONE	450	850.00
4	VIDEOCASSETE	ELETRONICOS	200	315.00

A Cláusula WHERE Com o Operador BETWEEN

O operador **BETWEEN** tem a finalidade de permitir uma consulta em determinada faixa de valores. Dessa forma, podemos utilizar este operador para selecionar todos os produtos cujos valores estejam entre 400,00 e 700,00 na tabela **PRODUTOS**. Vejamos:

```
SELECT * FROM PRODUTOS WHERE VALOR BETWEEN 400.00 AND 700.00
GO
```

```

File Edit View Query Tools Window Community Help
New Query Execute
Object Explorer Results Messages
PC-MARCO\SQLEXPRESS (SQL Server 9.0.3000.214)
  Databases
    MINHA_DB
      Tables
        System Tables
        dbo.AGENDA
        dbo.CADASTROS
        dbo.PRODUTOS
        dbo.TELEFONES
PC-MARCO\SQLEXPRESS - SQLQuery2.sql
SELECT * FROM PRODUTOS WHERE VALOR BETWEEN 400.00 AND 700.00
GO

```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	690.00
2	DVD PLAYER	ELETRONICOS	250	525.00
3	TELEVISOR	ELETRONICOS	350	650.00
4	FILMADORA DIGITAL	DIGITAIS	500	560.00
5	TECLADO	INFORMATICA	300	517.50

Por meio do operador **AND** e dos operadores relacionais **=**, **<** e **>** também é possível consultar uma determinada faixa de valores. No entanto, por meio do **BETWEEN** esta consulta tornar-se mais simples. Esse operador permite checar se o valor de uma coluna encontra-se em um determinado intervalo. Ele pode ser utilizado para verificar intervalos de data, caracteres, entre outros.

A Cláusula WHERE Com os Operadores NOT BETWEEN

Os operadores **NOT BETWEEN**, ao contrário do **BETWEEN**, permitem consultar os valores que não estão em uma faixa de valores determinados. Podemos consultar, na nossa tabela **PRODUTOS**, os valores que não estão entre 300,00 e 500,00. Vejamos:

```
SELECT * FROM PRODUTOS WHERE VALOR NOT BETWEEN 300.00 AND 500.00
GO
```

```

File Edit View Query Tools Window Community Help
New Query Execute
Object Explorer Results Messages
PC-MARCO\SQLEXPRESS (SQL Server 9.0.3000.214)
  Databases
    MINHA_DB
      Tables
        System Tables
        dbo.AGENDA
        dbo.CADASTROS
        dbo.PRODUTOS
        dbo.TELEFONES
        Views
        Synonyms
PC-MARCO\SQLEXPRESS - SQLQuery2.sql
SELECT * FROM PRODUTOS WHERE VALOR NOT BETWEEN 300.00 AND 500.00
GO

```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	690.00
2	DVD PLAYER	ELETRONICOS	250	525.00
3	MONITOR	INFORMATICA	400	828.00
4	TELEVISOR	ELETRONICOS	350	650.00
5	FILMADORA DIGITAL	DIGITAIS	500	560.00
6	CELULAR	TELEFONE	450	850.00
7	TECLADO	INFORMATICA	300	517.50
8	MOUSE	INFORMATICA	400	69.00

A Cláusula WHERE Com o Operador LIKE

O operador **LIKE** é empregado para situações em que se necessita retornar valores que se iniciam com um determinado caractere ou sentença de caracteres, terminem com um determinado caractere ou sentença de caracteres ou contenham um determinado caractere ou sentença desses. Por exemplo, desejamos retornar todos os produtos que se iniciem com a letra 'T', então vejamos:

```
SELECT * FROM PRODUTOS WHERE NOME LIKE 'T%'  
GO
```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	TELEVISOR	ELETRONICOS	350	650.00
2	TECLADO	INFORMATICA	300	517.50

Perceba que após o nosso caractere de busca, usamos um caractere coringa. Que no caso do SQL Server é o caractere porcentagem (%). Isso especifica ao SQL Server que desejamos retornar todos os valores que se iniciem com a letra 'T'. Caso quiséssemos todos os valores que terminassem com a letra 'T', usaríamos essa busca da seguinte forma:

```
SELECT * FROM PRODUTOS WHERE NOME LIKE '%T'  
GO
```

Ou, no caso da letra 'T' ser encontrada em qualquer posição do valor, usaríamos esta forma:

```
SELECT * FROM PRODUTOS WHERE NOME LIKE '%T%'  
GO
```

Com os caracteres coringas rodeando o nosso caractere de consulta. Vale lembrar que não é apenas o primeiro caractere que pode ser determinado para uma consulta. Caso desejamos obter como resultado todos os nomes iniciados com a letra 'M' e que contenha a letra 'S' em qualquer outra posição:

```
SELECT * FROM PRODUTOS WHERE NOME LIKE 'M%S%'  
GO
```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	MOUSE	INFORMATICA	400	69.00

Podemos restringir uma consulta com o operador **LIKE** determinando não apenas um caractere como, também, uma sílaba que deva estar presente no valor do campo. Agora, usaremos o operador **LIKE** em conjunto com o operador **AND** para obter como resultado os produtos cujos nomes possuam a sílaba 'RA' e os tipos que possuam a sílaba 'CA'.

```
SELECT * FROM PRODUTOS WHERE NOME LIKE '%RA%' AND TIPO LIKE '%CA%'
GO
```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	690.00

Nota: Caso em uma procura seja necessário localizar um registro que possua o caractere coringa (%), usa-se o **LIKE** nesta forma:

[comando] LIKE '%^%%' ESCAPE '^'

Isso informa ao SQL Server que nesta consulta o caractere que estiver à direita do caractere de **ESCAPE**, no caso o caractere '^', será o caractere a ser procurado.

A Cláusula WHERE Com os Operadores NOT LIKE

Os operadores **NOT LIKE** são utilizados da mesma forma que o operador **LIKE**, mas com eles obtemos com resultado inverso. Vejamos:

```
SELECT * FROM PRODUTOS WHERE NOME NOT LIKE '%RA%'
GO
```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	DVD PLAYER	ELETRONICOS	250	525.00
2	MONITOR	INFORMATICA	400	828.00
3	TELEVISOR	ELETRONICOS	350	650.00
4	CELULAR	TELEFONE	450	850.00
5	TECLADO	INFORMATICA	300	517.50
6	VIDEOCASSETE	ELETRONICOS	200	315.00
7	MOUSE	INFORMATICA	400	69.00

Exercícios

1. Qual a utilidade de cláusula **WHERE**?

Resposta:

2. Quais os operadores utilizados com a cláusula **WHERE** quando se deseja usar mais de uma condição de comparação?

Resposta:

3. Que operador é utilizado para selecionar uma faixa de valores a ser atingida por um determinado comando?

Resposta:

4. Para que é utilizado o operador **NOT LIKE**?

Resposta:

5. Que operador permite checar se o valor de uma coluna está presente em uma lista de elementos e pode ser utilizado no lugar do operador **OR** em determinadas situações?

Resposta:

Laboratório

Para fazer os exercícios do laboratório utilize a tabela **PRODUTOS**

1. Aumente em 12% o valor dos produtos cujos nomes iniciem com a letra 'F'

Resposta:

2. Aumentar em 50 unidades todos os produtos cujo valor seja maior que 400 e inferior a 600

Resposta:

3. Aplicar um desconto de 50% ($*0.5$) em todos os produtos que as unidades de estoque sejam maiores que 300

Resposta:

4. Exiba o produto de **CODIGO = 4**

Resposta:

5. Exibir todos os produtos que não tenham a letra 'Y'

Resposta:

6. Exibir todos os produtos que se iniciem com nome 'MO' e tenham como tipo as letras 'MA'

Resposta:

Capítulo 6

A Cláusula ORDER BY

Por vezes é necessário que o resultado de uma consulta seja mostrado em uma ordem específica, de acordo com determinado critério. Para tal, contamos com opções e cláusulas de ordenação. Uma delas é a cláusula **ORDER BY** (ordernar por). Utilizamos a cláusula em conjunto com o comando **SELECT** a fim de retornar resultados de uma consulta a uma tabela em determinada ordem.

Ordenando Por Colunas

É possível utilizar a cláusula **ORDER BY** para ordenar dados retornados utilizando como critério o nome ou número referente à posição, no **SELECT**. Para que os dados de uma tabela sejam retornados e exibidos de acordo com o nome da coluna, utilizamos o seguinte comando:

```
SELECT * FROM PRODUTOS ORDER BY TIPO  
GO
```

Isso irá nos retornar como resultado da consulta, todos os itens da tabela **PRODUTOS** ordenados pelo campo **TIPO**. Vejamos:

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, a database named 'MINHA_DB' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery1.sql' contains the following code:

```
SELECT * FROM PRODUTOS ORDER BY TIPO  
GO
```

The results pane displays a table with the following data, where the 'TIPO' column is highlighted with a red border:

CÓDIGO	NOME	TIPO	QUANTIDADE	VALOR
1	CAMERA DIGITAL	DIGITAIS	300	320.00
2	FILMADORA DIGITAL	DIGITAIS	500	560.00
3	VIDEOCASSETE	ELETRONICOS	200	315.00
4	DVD PLAYER	ELETRONICOS	250	525.00
5	TELEVISOR	ELETRONICOS	350	650.00
6	IMPRESSORA	INFORMATICA	200	690.00
7	MONITOR	INFORMATICA	400	828.00
8	MOUSE	INFORMATICA	400	69.00
9	TECLADO	INFORMATICA	300	517.50
10	CELULAR	TELEFONE	450	850.00

Veja que por padrão, a cláusula **ORDER BY** sempre ordena crescentemente.

Ordenando Por Mais de Uma Coluna

É possível utilizar a cláusula **ORDER BY** para ordenar os dados de várias colunas, isto com base nos nomes de várias colunas e nas posições das colunas no **SELECT**. A seguir um exemplo desse estilo de ordenação:

```
SELECT * FROM PRODUTOS ORDER BY NOME, TIPO  
GO
```

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	CAMERA DIGITAL	DIGITAIS	300	320.00
2	CELULAR	TELEFONE	450	850.00
3	DVD PLAYER	ELETRONICOS	250	525.00
4	FILMADORA DIGITAL	DIGITAIS	500	560.00
5	IMPRESSORA	INFORMATICA	200	690.00
6	MONITOR	INFORMATICA	400	828.00
7	MOUSE	INFORMATICA	400	69.00
8	TECLADO	INFORMATICA	300	517.50
9	TELEVISOR	ELETRONICOS	350	650.00
10	VIDEOCASSETE	ELETRONICOS	200	315.00

Veja que executado, ordena primeiramente pelo **NOME** e depois pela coluna **TIPO**.

ORDER BY ASC e DESC

A cláusula **ORDER BY** pode ser utilizada com as opções **ASC** e **DESC**, descritas a seguir:

- **ASC**: Esta opção faz com que as linhas sejam ordenadas na forma ascendente. (Padrão)
- **DESC**: Esta opção faz com que as linhas sejam ordenadas na forma descendente.

A seguir, um exemplo ordenando a tabela **PRODUTOS** pelo campo **VALOR** de forma descendente:

```
SELECT * FROM PRODUTOS ORDER BY VALOR DESC  
GO
```

Microsoft SQL Server Management Studio Express

File Edit View Query Tools Window Community Help

New Query Execute

Object Explorer

PC-MARCO\SQLEXPRESS (SQL Server 9.0.3000.218) MINHA_DB

Databases

- System Databases
- MINHA_DB
- Database Diagrams
- Tables
 - System Tables
 - dbo.AGENDA
 - dbo.CADASTROS
 - dbo.PRODUTOS
 - dbo.TELEFONES
- Views
- Synonyms
- Programmability
- Security

PC-MARCO\SQLEXPRESS - SQLQuery1.sql

```
SELECT * FROM PRODUTOS ORDER BY VALOR DESC
GO
```

Results

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	CELULAR	TELEFONE	450	850.00
2	MONITOR	INFORMATICA	400	828.00
3	IMPRESSORA	INFORMATICA	200	690.00
4	TELEVISOR	ELETRONICOS	350	650.00
5	FILMADORA DIGITAL	DIGITAIS	500	560.00
6	DVD PLAYER	ELETRONICOS	250	525.00
7	TECLADO	INFORMATICA	300	517.50
8	CAMERA DIGITAL	DIGITAIS	300	320.00
9	VIDEOCASSETE	ELETRONICOS	200	315.00
10	MOUSE	INFORMATICA	400	69.00

ASC e DESC

Também é possível mesclar as duas opções. Vejamos:

```
SELECT * FROM PRODUTOS ORDER BY NOME ASC, VALOR DESC
GO
```

Microsoft SQL Server Management Studio Express

File Edit View Query Tools Window Community Help

New Query Execute

Object Explorer

PC-MARCO\SQLEXPRESS (SQL Server 9.0.3000.218) MINHA_DB

Databases

- System Databases
- MINHA_DB
- Database Diagrams
- Tables
 - System Tables
 - dbo.AGENDA
 - dbo.CADASTROS
 - dbo.PRODUTOS
 - dbo.TELEFONES
- Views
- Synonyms
- Programmability
- Security

PC-MARCO\SQLEXPRESS - SQLQuery1.sql

```
SELECT * FROM PRODUTOS ORDER BY NOME ASC, VALOR DESC
GO
```

Results

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	CAMERA DIGITAL	DIGITAIS	300	320.00
2	CELULAR	TELEFONE	450	850.00
3	DVD PLAYER	ELETRONICOS	250	525.00
4	FILMADORA DIGITAL	DIGITAIS	500	560.00
5	IMPRESSORA	INFORMATICA	200	690.00
6	MONITOR	INFORMATICA	400	828.00
7	MOUSE	INFORMATICA	400	69.00
8	TECLADO	INFORMATICA	300	517.50
9	TELEVISOR	ELETRONICOS	350	650.00
10	VIDEOCASSETE	ELETRONICOS	200	315.00

A Cláusula TOP

Como resultado de uma consulta a uma tabela de um banco de dados, temos a opção de retornar a quantidade de linhas desejadas, a partir da primeira linha selecionada. Para isso, utilizamos a cláusula **TOP**.

Vamos novamente nos basear na tabela **PRODUTOS**. Para obter como resultado as primeiras cinco linhas da tabela, utilizamos o seguinte comando:

```
SELECT TOP 5 * FROM PRODUTOS
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, a database named 'MINHA_DB' is selected under 'PC-MARCO\SQLEXPRESS (SQL Server 9.0.30)'. In the center pane, a query window titled 'PC-MARCO\SQLEXPRESS... - SQLQuery1.sql' contains the following code:

```
SELECT TOP 5 * FROM PRODUTOS
GO
```

The 'Results' tab is selected, displaying the following table:

	CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	1	IMPRESSORA	INFORMATICA	200	690.00
2	2	CAMERA DIGITAL	DIGITAIS	300	320.00
3	3	DVD PLAYER	ELETRONICOS	250	525.00
4	4	MONITOR	INFORMATICA	400	828.00
5	5	TELEVISOR	ELETRONICOS	350	650.00

Observe que a consulta retornou apenas os primeiros cinco registros da tabela.

Considerações sobre a utilização de **TOP**:

- Em **Partitioned Views**, **TOP** não pode ser aplicada junto às instruções **UPDATE** e **INSERT**
- Não há ordem das linhas referenciadas na expressão **TOP** utilizada como as instruções **INSERT**, **DELETE** ou **UPDATE**, sendo que **TOP n** retorna linhas aleatórias **n** como resposta.

A Cláusula TOP Com ORDER BY

A cláusula **TOP** e a cláusula **ORDER BY** podem ser utilizadas de forma conjunta. Consideremos novamente a tabela **PRODUTOS**. Supondo que pretendemos exibir os três primeiros produtos de menor preço, dentre a relação de produtos da lista. Vejamos o comando utilizado:

```
SELECT TOP 3 * FROM PRODUTOS ORDER BY VALOR ASC
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, a database named 'MINHA_DB' is selected under 'PC-MARCO\SQLEXPRESS (SQL Server 9.0.30)'. In the center pane, a query window titled 'PC-MARCO\SQLEXPRESS... - SQLQuery1.sql' contains the following code:

```
SELECT TOP 3 * FROM PRODUTOS ORDER BY VALOR ASC
GO
```

The 'Results' tab is selected, displaying the following table:

	CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	10	MOUSE	INFORMATICA	400	69.00
2	9	VIDEOCASSETE	ELETRONICOS	200	315.00
3	2	CAMERA DIGITAL	DIGITAIS	300	320.00

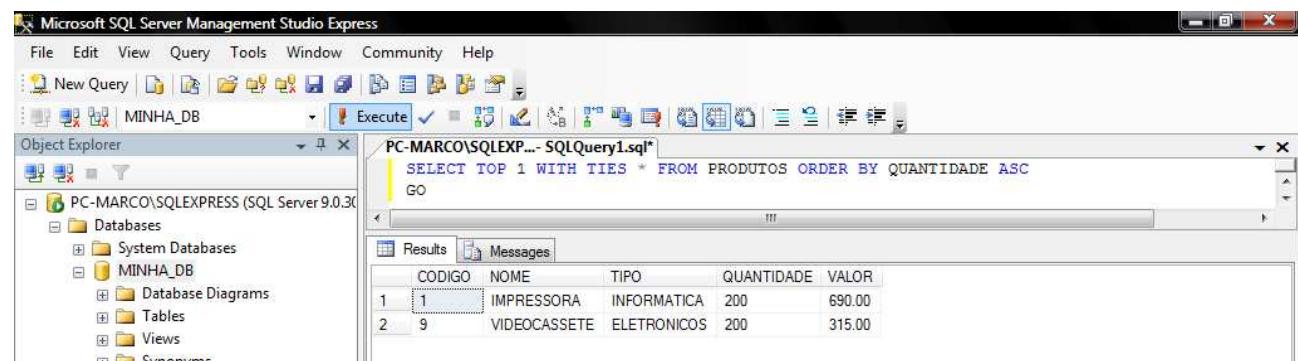
Isso também vale para a opção **DESC** do **ORDER BY**.

A Cláusula TOP WITH TIES Com ORDER BY

Permitida apenas em instruções de **SELECT**, e quando uma cláusula **ORDER BY** é especificada, a cláusula **TOP WITH TIES** determina que linhas adicionais sejam retornadas a partir do conjunto de resultados base com o mesmo valor apresentado nas colunas **ORDER BY**, sendo exibidas como as últimas das linhas **TOP n (PERCENT)**.

Vamos supor que precisamos obter como resultado o produto com a menor quantidade de unidades. Porém, é preciso considerar a existência de produtos com a mesma quantidade de unidades. Neste caso, executamos o seguinte comando para retornar os produtos com o menos número de unidades:

```
SELECT TOP 1 WITH TIES * FROM PRODUTOS ORDER BY QUANTIDADE ASC  
GO
```



The screenshot shows the Microsoft SQL Server Management Studio Express interface. The query window contains the following SQL code:

```
SELECT TOP 1 WITH TIES * FROM PRODUTOS ORDER BY QUANTIDADE ASC  
GO
```

The results pane displays the following data:

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	690.00
2	VIDECASSETE	ELETRONICOS	200	315.00

A cláusula **WITH TIES** retornou a primeira linha da tabela, além de todas as linhas que apresentam quantidades idênticas à quantidade do produto que a cláusula **TOP 1** selecionou.

Exercícios

1. Qual a função da cláusula **ORDER BY**?

Resposta:

2. Que comando utilizamos em conjunto com a cláusula **ORDER BY** a fim de permitir que os dados sejam exibidos na ordem desejada?

Resposta:

3. Qual a finalidade de utilizarmos as opções **ASC** e **DESC** com a cláusula **ORDER BY**?

Resposta:

4. Em que situações podemos utilizar as opções **ASC** e **DESC** em conjunto?

Resposta:

5. A cláusula **ORDER BY** permite que os dados sejam ordenados somente por uma coluna. Esta afirmação é verdadeira?

Resposta:

Laboratório

Utilize a tabela **PRODUTOS** para realizar os exercícios do laboratório.

1. Escreva uma consulta que exiba os dados da tabela **PRODUTOS** na forma crescente pelo campo **NOME**.

Resposta:

2. Escreva uma consulta que exiba o **NOME** dos produtos ordenado de forma decrescente e o **VALOR** de forma crescente.

Resposta:

3. Escreva uma consulta que exiba os três produtos mais caros.

Resposta:

4. Escreva uma consulta que exiba o **VALOR** do produto mais barato.

Resposta:

Capítulo 7

Integridade E Consistência - Parte 1

Para que um sistema de banco de dados sempre possa fornecer informações confiáveis aos usuários, o administrador deve filtrar algumas ações realizadas sobre essas informações a fim de evitar a ocorrência de possíveis erros relacionados às mesmas.

Além disso, esse sistema deve ser capaz de receber informações de forma constante sem ter que sofrer alterações com a mesma freqüência. O banco de dados deve possibilitar a inserção de uma grande quantidade de dados sem que o mesmo precise ser alterado em sua estrutura.

CONSTRAINTS

Constraints são objetos utilizados com a finalidade de estabelecer regras referentes à integridade e à consistência nas colunas das tabelas pertencentes a um sistema de banco de dados. Isso é muito importante porque, para planejar e criar tabelas, devemos garantir a integridade dos dados presentes nas colunas e identificar os valores válidos para tais dados.

A fim de assegurar a integridade dos dados de uma tabela, o SQL Server oferece cinco tipos diferentes de constraints, os quais são relacionados a cinco tipos diferentes de integridades. Segue abaixo:

Tipo de Integridade	Tipo de CONSTRAINT
Chave Primária	PRIMARY KEY
Chave Estrangeira	FOREIGN KEY / REFERENCES
Chave Única	UNIQUE
Regras de Validação	CHECK
Valor Padrão	DEFAULT

Chaves Primárias (PRIMARY KEY)

Normalmente, as tabelas possuem uma coluna contendo valores capazes de identificar uma linha de forma exclusiva. Essa coluna recebe o nome de chave primária, também conhecida como **PRIMARY KEY**, cuja finalidade é assegurar a integridade dos dados da tabela.

A chave primária pode ser gerada no momento da criação ou da alteração da tabela, bastando para isso definir uma constraint **PRIMARY KEY**. É importante ter em mente que cada tabela pode conter apenas uma constraint **PRIMARY KEY**, sendo que a coluna que a

representa não pode aceitar valores nulos. Esse tipo de constraint é capaz de assegurar que os dados sejam identificados de forma única e exclusiva, portanto, é comum que a coluna que a represente seja chamada de coluna de identidade. O **Database Engine** é responsável por assegurar essa exclusividade.

No nosso exemplo da tabela de **PRODUTOS**, temos um campo chamado **CODIGO** que serve como "chave" para que não existam linhas repetidas. Mas veremos que se quisermos inserir um novo produto com o código '5' isso é possível, pois esse campo não possui uma constraint do tipo **PRIMARY KEY**. Vejamos:

```
INSERT INTO PRODUTOS
(
    CODIGO,
    NOME,
    TIPO,
    QUANTIDADE,
    VALOR
)
VALUES
(
    5,
    'CAIXA DE SOM',
    'ELETRONICOS',
    150,
    200.00
)
GO
```

Após inserirmos o registro acima, realizamos uma consulta na tabela **PRODUTOS**. Observe que existem dois produtos com o mesmo **CODIGO**:

CODIGO	NOME	TIPO	QUANTIDADE	VALOR
1	IMPRESSORA	INFORMATICA	200	690.00
2	CAMERA DIGITAL	DIGITAIS	300	320.00
3	DVD PLAYER	ELETRONICOS	250	525.00
4	MONITOR	INFORMATICA	400	828.00
5	TELEVISOR	ELETRONICOS	350	650.00
6	CAIXA DE SOM	ELETRONICOS	150	200.00
7	FILMADORA DIGITAL	DIGITAIS	500	560.00
8	CELULAR	TELEFONE	450	850.00
9	TECLADO	INFORMATICA	300	517.50
10	VIDEOCASSETE	ELETRONICOS	200	315.00
11	MOUSE	INFORMATICA	400	69.00

Em um caso real, não poderíamos ter códigos idênticos para produtos diferentes. A constraint **PRIMARY KEY** evita isso, no momento em que tentamos inserir um registro com um código já existente, o SQL Server retorna um aviso de inconsistência para o usuário avisando-o que o processo não pôde ser realizado.

Nota: A chave primária deve fazer parte da integridade referencial que se estabelece entre duas tabelas relacionadas. Mais detalhes a respeito deste assunto serão vistos adiante.

Chaves Secundárias ou Únicas (UNIQUE)

Além da constraint **PRIMARY KEY**, também podemos utilizar constraints **UNIQUE**, as quais asseguram que dados duplicados não sejam inseridos em colunas que não fazem parte de chaves primárias. A **UNIQUE** é uma constraint que também é capaz de assegurar a exclusividade dos dados. Em uma tabela, várias colunas podem ser definidas como constraints **UNIQUE**.

As colunas nas quais são definidas constraints **UNIQUE** permitem a inclusão de valores nulos, desde que seja apenas um por coluna.

Nota: As constraints **UNIQUE** poder ser utilizadas para referenciar uma chave estrangeira.

Para entendermos melhor, tomemos um cenário em que tenhamos uma tabela de cadastro de pessoas, **CADASTROS**. Nessa tabela teremos a coluna **CODIGO** que será nossa coluna identidade, uma coluna **NOME**, **RG** e **DATA_NASCIMENTO**. Como já sabemos, o fato de existir uma coluna identidade na nossa tabela nos impede de acrescentar duas pessoas com o mesmo código, mas não nos impede de acrescentar a mesma pessoa mais de uma vez. Nesse caso usaríamos uma constraint do tipo **UNIQUE** para a coluna **RG**, já que não existem pessoas diferentes com o mesmo número de **RG**. Assim, a linha fica identificada pelo campo **CODIGO** e pelo campo **RG**. (Isso não nos impede de deixar o campo **RG** nulo, desde que só exista um nulo).

Chave Estrangeira (FOREIGN KEY / REFERENCES)

Colunas que representam chaves estrangeiras são utilizadas como a finalidade de estabelecer um vínculo entre os dados de tabelas distintas. A criação deste tipo de chave requer a utilização da constraint **FOREIGN KEY / REFERENCES**.

Para compreendermos este assunto de forma mais adequada, destacamos que para criarmos uma chave estrangeira é preciso que a coluna da primeira tabela, na qual se encontra a chave primária, seja referenciada pela coluna que se encontra na segunda tabela. Dessa forma, a coluna da segunda tabela torna-se a chave estrangeira.

Vale destacar, no entanto, que não é necessário que uma constraint **FOREIGN KEY / REFERENCES** em uma tabela seja vinculada apenas a uma constraint **PRIMARY KEY** em outra tabela. Além disso, embora a **FOREIGN KEY / REFERENCES** possa conter valores nulos, é possível que, nessa situação, a verificação dos valores que forma esta constraint não ocorra.

Podemos assegurar que a verificação dos valores de **FOREIGN KEY / REFERENCES** ocorra por meio da especificação do valor **NOT NULL** em todas as colunas que fazem parte deste tipo de constraint.

Cenário:

A nossa tabela do cenário anterior, **CADASTROS**, é referente a uma locadora de vídeo. Nessa locadora de vídeo não é necessário que parentes de uma primeira pessoa já cadastrada façam cadastro efetivo. Elas simplesmente podem ser adicionadas como **DEPENDENTES** da primeira pessoa. Portanto, seria criada uma nova tabela chamada **DEPENDENTES** com os campos **CODIGO**, **COD_CAD**, **NOME**, **RG** e **DATA_NASCIMENTO**. Onde **CODIGO** seria um campo de **PRIMARY KEY**, pois só podem existir registros exclusivos, **COD_CAD** seria um campo **FOREIGN KEY / REFERENCES** para o campo **CODIGO** na tabela **CADASTROS** e o campo **RG** seria um **UNIQUE**, para que fosse evitado o cadastro da mesma pessoa mais de uma vez.

Regras de Validação (CHECK)

Além de evitar que os usuários insiram valores inexistentes na tabela, as regras de validação dos dados evitam situações como as ilustradas a seguir:

CODIGO (PRIMARY KEY)	NOME	RG (UNIQUE)	SEXO	SALARIO
1	Maria Junqueira	23.456.789	F	-1900.00
2	José Almeida	98.765.432	M	900.00
3	Maristela Dias	45.678.912	X	2000.00
4	João Barbosa	32.654.987	M	1000.00

Como podemos observar, nas colunas **SEXO** e **SALARIO**, os valores inseridos para os cadastros **MARISTELA DIAS** e **MARIA JUNQUEIRA** estão inadequados, pois não existe sexo **X** e, tampouco, salário negativo. (Exceto aquelas pessoas que "pagam" para trabalhar).

Quando pesquisarmos pelas pessoas do sexo feminino, receberemos o seguinte resultado:

CODIGO (PRIMARY KEY)	NOME	RG (UNIQUE)	SEXO	SALARIO
1	Maria Junqueira	23.456.789	F	-1900.00

Ou, no caso de necessitarmos da soma de todos os salários, teríamos o seguinte:

CODIGO (PRIMARY KEY)	NOME	RG (UNIQUE)	SEXO	SALARIO
1	Maria Junqueira	23.456.789	F	-1900.00
2	José Almeida	98.765.432	M	900.00
3	Maristela Dias	45.678.912	X	2000.00
4	João Barbosa	32.654.987	M	1000.00
Soma dos salários				2000.00

Valor Padrão (DEFAULT)

Quando um valor padrão é estabelecido para uma coluna, o sistema assume que ele deve ser utilizado sempre que o valor que deveria ser inserido é suprimido, ou seja, quando o usuário deixa de preencher o valor manualmente. Vejamos a tabela:

CODIGO (PRIMARY KEY)	NOME	RG (UNIQUE)	SEXO (DEFAULT 'F')	SALARIO (DEFAULT 0)
1	Maria Junqueira	23.456.789	F	-1900.00
2	José Almeida	98.765.432	M	900.00
3	Maristela Dias	45.678.912	X	2000.00
4	João Barbosa	32.654.987	M	1000.00

Temos, então, que aplicar a constraint **DEFAULT** para a coluna **SEXO** e a coluna **SALARIO**. Com os valores padrões 'F' e 0 (zero) respectivamente. Se adicionarmos um novo registro à tabela, omitindo os valores para **SEXO** e **SALARIO**, o novo registro ficará dessa forma:

CODIGO (PRIMARY KEY)	NOME	RG (UNIQUE)	SEXO (DEFAULT 'F')	SALARIO (DEFAULT 0)
1	Maria Junqueira	23.456.789	F	-1900.00
2	José Almeida	98.765.432	M	900.00
3	Maristela Dias	45.678.912	X	2000.00
4	João Barbosa	32.654.987	M	1000.00
5	Fernanda Casta	56.987.123	F	0.00

Valores NULL

Além dos valores padrões, também é possível atribuir valores nulos a uma coluna. Por exemplo, na tabela de **FUNCIONARIOS**, podemos atribuir valores nulos a coluna **SALARIO**. Com isso, podemos inserir todas as informações da pessoa, mas, omitindo o valor do salário dessa pessoa. Vejamos:

CODIGO (PRIMARY KEY)	NOME	RG (UNIQUE)	SEXO (DEFAULT 'F')	SALARIO (DEFAULT 0 e admite valor NULL)
1	Maria Junqueira	23.456.789	F	-1900.00
2	José Almeida	98.765.432	M	900.00
3	Maristela Dias	45.678.912	X	2000.00
4	João Barbosa	32.654.987	M	1000.00
5	Fernanda Casta	56.987.123	F	0.00
6	Manuel Ferraz	25.897.664	M	NULL

Atribuir valor nulo a uma coluna significa que ela não terá valor algum. Assim, de acordo com as regras de integridade e consistência dos dados, é preciso atribuir **nulabilidade** das colunas de uma tabela a fim de determinar se elas aceitarão valores nulos (**NULL**) ou não (**NOT NULL**). Quando atribuímos um valor nulo a uma coluna, esta pode ocupar espaço no banco de dados, mas isso depende de seu datatype.

Regras de constraints

Cada tipo de constraint possui uma regra específica. Vejamos essas regras:

Tipo de constraint	Descrição
PRIMARY KEY	Uma coluna que é definida como chave primária não pode aceitar valores nulos. Em cada tabela, pode haver somente uma constraint de chave primária.
FOREIGN KEY / REFERENCES	Várias colunas podem ser definidas como chave estrangeira. No entanto, para que uma coluna seja definida dessa forma, é preciso que ela já tenha sido definida como chave primária em outra tabela. As colunas definidas como chaves estrangeiras podem aceitar valores nulos (apenas um), e os datatypes das colunas relacionadas devem ser iguais.
UNIQUE	Várias colunas de uma tabela podem ser definidas como sendo chave única e, ainda, aceitar valores nulos (apenas um).
CHECK	Diversas colunas de uma tabela podem ser definidas como constraint CHECK . Essas colunas podem aceitar valores nulos, mas isso depende das regras que são determinadas para elas.
DEFAULT	Várias colunas de uma tabela podem ser definidas como constraint DEFAULT . Essas colunas podem aceitar valores nulos.

Criando Tabelas Com Todas as Regras de Integridade e Consistência

Nessa parte, veremos a criação de um cadastro de pessoas simples. Nas tabelas aplicaremos as regras de integridade e consistência que acabamos de aprender, com o intuito de fixar o conhecimento. Segue script para criação da database, das tabelas e registros:

```
USE MASTER
GO

IF EXISTS (SELECT * FROM MASTER.DBO.SYSDATABASES WHERE NAME LIKE '%MINHA_DB%')
    DROP DATABASE MINHA_DB
GO

CREATE DATABASE MINHA_DB
GO

USE MINHA_DB
GO

CREATE TABLE CADASTROS
(
    CODIGO      INT          NOT NULL,
    NOME        VARCHAR(50),
    ENDERECO    VARCHAR(50),
```

```

        CEP          CHAR (9)           NOT NULL,
        BAIRRO      VARCHAR (25),
        CIDADE      VARCHAR (25),
        ESTADO      CHAR (2),
        RG          CHAR (12)          NOT NULL,
        SALARIO     DECIMAL (9,2)      DEFAULT 0,
        SEXO          CHAR (1)          DEFAULT 'F',

        CONSTRAINT PK_CADASTROS_CODIGO PRIMARY KEY (CODIGO),
        CONSTRAINT UQ_CADASTROS_RG UNIQUE (RG),
        CONSTRAINT CK_CADASTROS_SALARIO CHECK (SALARIO >= 0),
        CONSTRAINT CK_CADASTROS_SEXO CHECK (SEXO IN ('F', 'M'))
)
GO

CREATE TABLE DEPENDENTES
(
        CODIGO      INT             NOT NULL,
        COD_CAD     INT             NOT NULL,
        NOME        VARCHAR (50),
        RG          CHAR (12)         NOT NULL,
        DATA_NASCIMENTO  SMALLDATETIME,
        SEXO          CHAR (1)          DEFAULT 'F',

        CONSTRAINT PK_DEPENDENTES_CODIGO PRIMARY KEY (CODIGO),
        CONSTRAINT FK_DEPENDENTES_COD_CAD FOREIGN KEY (COD_CAD) REFERENCES CADASTROS (CODIGO),
        CONSTRAINT UQ_DEPENDENTES_RG UNIQUE (RG),
        CONSTRAINT CK_DEPENDENTES_SEXO CHECK (SEXO IN ('F', 'M'))
)
GO

INSERT INTO CADASTROS VALUES (1, 'MARIA JUNQUEIRA', 'RUA CAMBOJA, 100', '02345-123', 'AGUA LONGE', 'SÃO PAULO', 'SP', '23.456.789', 1900.00, 'F')
INSERT INTO CADASTROS VALUES (2, 'JOSE ALMEIRA', 'RUA BALBUCIA, 120', '05498-987', 'AGUA PERTO', 'SÃO PAULO', 'SP', '98.765.432', 1000.00, 'M')
INSERT INTO CADASTROS VALUES (3, 'MARISTELA DIAS', 'AVENIDA LINDOMAR, 520', '05412-369', 'BAIXAQUI', 'MINAS GERAIS', 'MG', '45.678.912', 2000.00, 'F')
INSERT INTO CADASTROS VALUES (4, 'JOAO BARBOSA', 'RUA CASA DO BARBEIRO, 220', '03526-333', 'BAIXALI', 'SÃO PAULO', 'SP', '32.654.987', 1500.00, 'M')
INSERT INTO CADASTROS VALUES (5, 'FERNANDA CASTA', 'ALAMEDA VICENTINO, 360', '05255-002', 'MORRO ALTO', 'RIO DE JANEIRO', 'RJ', '56.987.123', 900.00, 'F')
INSERT INTO CADASTROS VALUES (6, 'MANUEL FERRAZ', 'TRAVESSA PASSA TRES, 10', '01411-222', 'MORRO BAIXO', 'SÃO PAULO', 'SP', '25.897.664', 3000.00, 'M')
INSERT INTO CADASTROS VALUES (7, 'PRIMO ROSSI', 'RUA CONGLOMERADO, 500', '08744-001', 'MORRO MEDIO', 'MINAS GERAIS', 'MG', '52.986.741', 2100.00, 'M')
INSERT INTO CADASTROS VALUES (8, 'GLAUBER DOS SANTOS', 'RUA GRANADA, 1', '06352-544', 'FAZENDA PEQUENA', 'MINAS GERAIS', 'MG', '22.369.147', 2300.00, 'M')
INSERT INTO CADASTROS VALUES (9, 'MURILo BEBE BENICIO', 'AVENIDA MOTO ROLA, 650', '04144-547', 'FAZENDA GRANDE', 'SÃO PAULO', 'SP', '98.365.125', 800.00, 'M')
INSERT INTO CADASTROS VALUES (10, 'CACILDA CACILDES', 'AVENIDA SARGENTO BIGODAO, 522', '08524-963', 'CHACARA MEDIA', 'RIO DE JANEIRO', 'RJ', '41.411.441', 800.00, 'F')
INSERT INTO CADASTROS VALUES (11, 'VERONICA VENTANIA', 'RUA MORENA FURACAO, 666', '01472-564', 'VILA TORMENTA', 'SÃO PAULO', 'SP', '45.859.423', 350.00, 'F')
GO

INSERT INTO DEPENDENTES VALUES (1, 1, 'MARCIO JUNQUEIRA FERMAT', '54.698.321', '11/12/78', 'M')
INSERT INTO DEPENDENTES VALUES (2, 1, 'FERNANDO JUNQUEIRA FERMAT', '54.698.322', '25/5/2000', 'M')
INSERT INTO DEPENDENTES VALUES (3, 2, 'JOSE ALMEIDA JUNIOR', '12.369.741', '3/3/1983', 'M')

```

```

INSERT INTO DEPENDENTES VALUES (4, 3, 'CAROLINA DIAS
CACAPAVA', '56.454.654', '29/2/1992', 'F')
INSERT INTO DEPENDENTES VALUES (5, 7, 'FERNANDA BEATRIZ MONTEIRO
ROSSI', '98.754.548', '21/7/1987', 'F')
INSERT INTO DEPENDENTES VALUES (6, 8, 'GLOBENILDO MULERA DOS
SANTOS', '21.455.469', '23/9/2001', 'M')
INSERT INTO DEPENDENTES VALUES (7, 8, 'GLOBALINA MULERA DOS
SANTOS', '21.712.842', '11/7/2001', 'F')
INSERT INTO DEPENDENTES VALUES (8, 8, 'GLOBERTO MULERA DOS
SANTOS', '98.721.379', '18/11/1982', 'M')
INSERT INTO DEPENDENTES VALUES (9, 11, 'ROBERTO VENTANIA E
TEMPESTADE', '32.198.712', '28/10/1985', 'M')
GO

```

Modelo Entidade - Relacionamento (MER)

O **MER** (Modelo Entidade - Relacionamento) tem seu conceito baseado na teoria relacional de Codd. Esse conceito diz que a expressão da realizada baseia-se no relacionamento existente entre as entidades, uma vez que essa realidade é dirigida pelos fatos determinados por tais relacionamentos.

O conceito referente ao **MER**, que está relacionado principalmente aos bancos de dados, permite apresentar os tipos de relacionamentos existentes entre os dados de um sistema.

A evolução desse modelo, o **MER**, foi fundamentada em mecanismos de abstração, cujo conceito permite determinar quais partes da realidade são importantes para que o sistema de informações seja construído. Além disso, por meio dos conceitos de abstração também é possível determinar quais aspectos referentes à modelagem são importantes para modelar o ambiente.

Alguns dos mecanismos de abstração nos quais se baseia a evolução do **MER** são: agregação, classificação e generalização. Mais detalhes a respeito dos relacionamentos entre entidades serão abordados a seguir.

Relacionamento

Os relacionamentos são responsáveis por definir uma ligação entre dois objetos pertencentes ao mundo real. Quando trabalhamos com aplicações construídas e administradas por um **SGBD** (Sistema Gerenciador de Banco de Dados), o relacionamento é o responsável por unir duas ou mais tabelas de banco de dados.

Vale destacar que o grau de relacionamento entre duas entidades é determinado pela quantidade de ocorrências de uma entidade que está relacionada à outra. O grau de relacionamento entre entidades também é chamado de cardinalidade.

Relacionamento 1 : 1

No relacionamento 1 : 1 (um-para-um), cada um dos elementos de uma entidade está relacionado com um único elemento de outra entidade. Para compreendermos de forma adequada, considere duas tabelas **CONVIDADOS** e **CONJUGES**. Na tabela **CONVIDADOS**, temos a seguinte estrutura:

COD_CONV	NOME	SEXO
1	Carlos da Silva	M
2	Joaquim Nabuco	M
3	Carla Moreno Costa	F

E na tabela **CONJUGES** temos essa estrutura:

COD_CONV_CONJ	NOME	SEXO
1	Maria Teresa da Silva	F
3	Fernando Costa	M

Perceba que o **COD_CONV_CONJ** faz referência ao **COD_CONV**, já que cada convidado tem apenas um cônjuge. Esse é um caso de uma relacionamento 1 : 1.

Relacionamento 1 : N

Para compreendermos o relacionamento 1 : N (um-para-muitos), consideremos duas entidades, as quais serão chamadas de **A** e **B**. Nesse tipo de relacionamento, cada elemento da entidade **A** pode ter relacionamento com vários elementos da entidade **B**. Mas o inverso é falso. As tabelas **PAIS** e **FILHOS** são exemplos de relacionamentos 1 : N. Vejamos a tabela **PAIS**.

COD_PAIS	NOME
1	Marisléia dos Santos Braga
2	Esgrumilda Argentina
3	Fernando Porto
4	Pedro Pedreira

Agora a tabela **FILHOS**:

COD_FIL	COD_PAI	NOME
1	1	Marcelo Braga
2	1	Fernando Braga
3	4	Silva Monte Pedreira

Perceba que um pai pode ter vários filhos, mas um filho tem apenas um pai.

Para que se estabeleçam relacionamentos de 1 : N (um-para-muitos), devemos contar com duas tabelas, em que a primeira (1) obrigatoriamente deve ter uma coluna que utilize a chave primária. Vale recordar que colunas com chave primária não aceitam a inserção de valores repetidos.

Já na tabela que representa o relacionamento para muitos (N), deverá haver uma consulta referente à primeira tabela, a qual deve utilizar a chave estrangeira para que, dessa forma, seja estabelecido o relacionamento entre ambas as tabelas.

Devemos estar atentos ao fato de que, diferentemente das chaves primárias, colunas que utilizam chave estrangeira aceitam a inserção de valores duplicados.

Relacionamento N : N

O relacionamento N : N (muitos-para-muitos) possui uma característica diferente dos outros. Neste caso, os dados serão diretamente relacionados ao fato, e não às entidades, como observamos em outros tipos de relacionamento.

É importante destacar que pode não haver associação de fatos nas situações em que os relacionamentos são de caráter condicional. Neste caso, a cardinalidade deve ser determinada por meio de ampla análise quanto à possibilidade de ocorrerem relacionamentos.

Para compreendermos esse modo de relacionamento, tomemos como exemplo duas entidades: **ALUNOS** e **CURSOS**. Com elas, temos a seguinte situação: um aluno pode fazer diversos cursos e um curso pode ter diversos alunos. Vejamos a ilustração:

Tabela **ALUNOS**:

COD_ALUNO	NOME	SEXO
1	Fernando Fernandes	M
2	Carlos Carvalho	M
3	Maria Garcia	F
4	Inês Pereira	F

Tabela **CURSOS**:

COD_CURSO	NOME
1	Matemática
2	Física
3	Química
4	Biologia
5	História

Agora, a partir dessas duas tabelas podemos traçar o seguinte relacionamento:

COD_ALUNO	COD_CURSO
1	2
1	5
2	1
2	3
3	2
3	4
3	5
4	1
4	2

Para estabelecer este tipo de relacionamento, devemos ter três tabelas, sendo que a terceira é responsável por relacionar as outras duas. Para isso, é preciso que essas duas primeiras tabelas tenham uma coluna que seja chave primária.

As colunas que são chaves primárias na primeira e na segunda tabela devem ser colunas com chave estrangeira na terceira. Com isso, esta tabela terá duas chaves estrangeiras, as quais formam uma chave primária composta.

Exercícios

1. O que são constraints?

Resposta:

2. Quais são os tipos de constraints existentes?

Resposta:

3. Que são chaves primárias?

Resposta:

4. Qual a finalidade de utilização de datatypes?

Resposta:

5. Quais são os tipos de relacionamentos existentes?

Resposta:

Laboratório

Nesse laboratório iremos criar um sistema simplificado de venda de CDs. Para isso será necessário criar uma nova database.

1. Crie uma database com o nome **DB_CDS**

Resposta:

2. Conectar-se a database **DB_CDS**

Resposta:

3. Crie as tabelas especificadas a seguir com as respectivas constraints:

TABELA: ARTISTAS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_ART	Código do Artista	INT	NOT NULL	PRIMARY KEY
NOME_ART	Nome do Artista	VARCHAR(100)	NOT NULL	UNIQUE

TABELA: GRAVADORAS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_GRAV	Código da Gravadora	INT	NOT NULL	PRIMARY KEY
NOME_GRAV	Nome da Gravadora	VARCHAR(50)	NOT NULL	UNIQUE

TABELA: CATEGORIAS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_CAT	Código da Categoria	INT	NOT NULL	PRIMARY KEY
NOME_CAT	Nome da Categoria	VARCHAR(50)	NOT NULL	UNIQUE

TABELA: ESTADOS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
SIGLA_EST	Sigla do Estado	CHAR(2)	NOT NULL	PRIMARY KEY
NOME_EST	Nome do Estado	VARCHAR(50)	NOT NULL	UNIQUE

TABELA: CIDADES

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_CID	Código da Cidade	INT	NOT NULL	PRIMARY KEY
SIGLA_EST	Sigla do Estado	CHAR(2)	NOT NULL	FOREIGN KEY / REFERENCES (SIGLA_EST de ESTADOS)
NOME_CID	Nome da Cidade	VARCHAR(50)	NOT NULL	

TABELA: CLIENTES

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_CLI	Código do Cliente	INT	NOT NULL	PRIMARY KEY
COD_CID	Código da Cidade	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_CID de CIDADES)
NOME_CLI	Nome do Cliente	VARCHAR(50)	NOT NULL	
END_CLI	Endereço do Cliente	VARCHAR(100)	NOT NULL	
RENDA_CLI	Renda do Cliente	DECIMAL(9,2)	NOT NULL	CHECK >= 0 e DEFAULT 0
SEXO_CLI	Sexo do Cliente	CHAR(1)	NOT NULL	CHECK ('F','M') e DEFAULT 'F'

TABELA: CONJUGE

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_CLI	Código do Cliente	INT	NOT NULL	PRIMARY KEY e FOREIGN KEY / REFERENCES (COD_CLI de CLIENTES)
NOME_CONJ	Nome do Cônjuge	VARCHAR(50)	NOT NULL	
RENDAS_CONJ	Renda do Cônjuge	DECIMAL(9,2)	NOT NULL	CHECK >= 0 e DEFAULT 0
SEXO_CONJ	Sexo do Cônjuge	CHAR(1)	NOT NULL	CHECK ('F','M') e DEFAULT 'F'

TABELA: FUNCIONARIOS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_FUNC	Código do Funcionário	INT	NOT NULL	PRIMARY KEY
NOME_FUNC	Nome do Funcionário	VARCHAR(50)	NOT NULL	
END_FUNC	Endereço do Funcionário	VARCHAR(100)	NOT NULL	
SAL_FUNC	Renda do Funcionário	DECIMAL(9,2)	NOT NULL	CHECK >= 0 e DEFAULT 0
SEXO_FUNC	Sexo do Funcionário	CHAR(1)	NOT NULL	CHECK ('F','M') e DEFAULT 'F'

TABELA: DEPENDENTES

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_DEP	Código do Dependente	INT	NOT NULL	PRIMARY KEY
COD_FUNC	Código do Funcionário	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_FUNC de FUNCIONARIOS)
NOME_DEP	Nome do Dependente	VARCHAR(100)	NOT NULL	
SEXO_DEP	Sexo do Dependente	CHAR(1)	NOT NULL	CHECK ('F','M') e DEFAULT 'F'

TABELA: TITULOS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_TIT	Código do Título	INT	NOT NULL	PRIMARY KEY
COD_CAT	Código da Categoria	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_CAT de CATEGORIAS)
COD_GRAV	Código da Gravadora	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_GRAV de GRAVADORAS)
NOME_CD	Nome do CD	VARCHAR(50)	NOT NULL	UNIQUE
VAL_CD	Valor do CD	DECIMAL(9,2)	NOT NULL	CHECK > 0
QTD_ESTQ	Quantidade em Estoque	INT	NOT NULL	CHECK >= 0

TABELA: PEDIDOS

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
NUM_PED	Número do Pedido	INT	NOT NULL	PRIMARY KEY
COD_CLI	Código do Cliente	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_CLI de CLIENTES)
COD_FUNC	Código do Funcionário	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_FUNC de FUNCIONARIOS)
DATA_PED	Data da Emissão	SMALLDATETIME	NOT NULL	
VAL_PED	Valor do Pedido	DECIMAL(9,2)	NOT NULL	CHECK >= 0 e DEFAULT 0

TABELA: TITULOS_PEDIDO

Obs.: Esta tabela terá uma chave primária composta por duas colunas: NUM_PED e COD_TIT				
COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
NUM_PED	Número do Pedido	INT	NOT NULL	FOREIGN KEY / REFERENCES (NUM_PED de PEDIDOS)
COD_TIT	Código do Título	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_TIT de TITULOS)
QTD_CD	Quantidade de cada Título	INT	NOT NULL	CHECK >= 1
VAL_CD	Valor do CD	DECIMAL(9,2)	NOT NULL	CHECK > 0

TABELA: TITULOS_ARTISTA

Obs.: Esta tabela terá uma chave primária composta por duas colunas:
COD_TIT e COD_ART

COLUNA	DESCRÍÇÃO	DATATYPE	NULABILIDADE	CONSTRAINT
COD_TIT	Código do Título	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_TIT de TITULOS)
COD_ART	Código do Artista	INT	NOT NULL	FOREIGN KEY / REFERENCES (COD_ART de ARTISTAS)

4. Agora insira os seguintes dados nas respectivas tabelas:

TABELA: ARTISTAS

COD_ART	NOME_ART
1	MARISA MONTE
2	GILBERTO GIL
3	CAETANO VELOSO
4	MILTON NASCIMENTO
5	LEGIÃO URBANA
6	THE BEATLES
7	RITA LEE

TABELA: GRAVADORAS

COD_GRAV	NOME_GRAV
1	POLYGRAM
2	EMI
3	SOM LIVRE
4	SOM MUSIC

TABELA: CATEGORIAS

COD_CAT	NOME_CAT
1	MPB
2	TRILHA SONORA
3	ROCK INTERNACIONAL
4	ROCK NACIONAL

TABELA: ESTADOS

SIGLA_EST	NOME_EST
SP	SÃO PAULO
MG	MINAS GERAIS
RJ	RIO DE JANEIRO

TABELA: CIDADES

COD_CID	SIGLA_EST	NOME_CID
1	SP	SÃO PAULO
2	SP	SOROCABA
3	SP	JUNDIAÍ
4	SP	AMERICANA
5	SP	ARARAQUARA
6	MG	OURO PRETO
7	ES	CACHOEIRA DO ITAPEMIRIM

TABELA: CLIENTES

COD_CLI	COD_CID	NOME_CLI	END_CLI	RENDACLIENTE	SEXO_CLI
1	1	JOSÉ NOGUEIRA	RUA A	1500.00	M
2	1	ÂNGELO PEREIRA	RUA B	2000.00	M
3	1	ALÉM MAR PARANHOS	RUA C	1500.00	M
4	1	CATARINA SOUZA	RUA D	892.00	F
5	1	VAGNER COSTA	RUA E	950.00	M
6	2	ANTENOR DA COSTA	RUA F	1582.00	M
7	2	MARIA AMÉLIA DE SOUZA	RUA G	1152.00	F
8	2	PAULO ROBERTO SILVA	RUA H	3250.00	M
9	3	FÁTIMA SOUZA	RUA I	1632.00	F
10	3	JOEL DA ROCHA	RUA J	2000.00	M

TABELA: CONJUGE

COD_CLI	NOME_CONJ	RENDACLIENTE	SEXO_CLI
1	CARLA NOGUEIRA	2500.00	F
2	EMILIA PEREIRA	5500.00	F
6	ALTIVA DA COSTA	3000.00	F
7	CARLOS DE SOUZA	3250.00	M

TABELA: FUNCIONARIOS

COD_FUNC	NOME_FUNC	END_FUNC	SAL_FUNC	SEXO_FUNC
1	VÂNIA GABRIELA PEREIRA	RUA A	2500.00	F
2	NORBERTO PEREIRA DA SILVA	RUA B	300.00	M
3	OLAVO LINHARES	RUA C	580.00	M
4	PAULA DA SILVA	RUA D	3000.00	F
5	ROLANDO ROCHA	RUA E	2000.00	M

TABELA: DEPENDENTES

COD_DEP	COD_FUNC	NOME_DEP	SEXO_CLI
1	1	ANA PEREIRA	F
2	1	ROBERTO PEREIRA	M
3	1	CELSO PEREIRA	M
4	3	BRISA LINHARES	F
5	3	MARI SOL LINHARES	F
6	4	SONIA DA SILVA	F

TABELA: TITULOS

COD_TIT	COD_CAT	COD_GRAV	NOME_CD	VAL_CD	QTD_ESTQ
1	1	1	TRIBALISTAS	30.00	1500
2	1	2	TROPICÁLIA	50.00	500
3	1	1	AQUELE ABRAÇO	50.00	600
4	1	2	REFAZENDA	60.00	1000
5	1	3	TOTALMENTE DEMAIS	50.00	2000
6	1	3	TRAVESSIA	55.00	500
7	1	2	COURAGE	30.00	200
8	4	3	LEGIÃO URBANA	20.00	100
9	3	2	THE BEATLES	30.00	300
10	4	1	RITA LEE	30.00	500

TABELA: PEDIDOS

NUM_PED	COD_CID	COD_FUNC	DATA_PED	VAL_PED
1	1	2	02/05/02	1500.00
2	3	4	02/05/02	50.00
3	4	7	02/06/02	100.00
4	1	4	02/02/03	200.00
5	7	5	02/03/03	300.00
6	4	4	02/03/03	100.00
7	5	5	02/03/03	50.00
8	8	2	02/03/03	50.00
9	2	2	02/03/03	2000.00
10	7	1	02/03/03	3000.00

TABELA: TITULOS_ARTISTA

COD_TIT	COD_ART
1	1
2	2
3	2
4	2
5	3
6	4
7	4
8	5
9	6
10	7

TABELA: TITULOS_PEDIDO

NUM_PED	COD_TIT	QTD_CD	VAL_CD
1	1	2	30.00
1	2	3	20.00
2	1	1	50.00
2	2	3	30.00
3	1	2	40.00
4	2	3	20.00
5	1	2	25.00
6	2	3	30.00
6	3	1	35.00
7	4	2	55.00
8	1	4	60.00
9	2	3	15.00
10	7	2	15.00

Capítulo 8

Associando tabelas

A associação de tabelas pode ser realizada para diversas finalidades, por exemplo, converter em informações os dados encontrados em duas ou mais tabelas. Essa associação pode ser realizada por meio da cláusula **WHERE** e **JOIN**. É importante ressaltar que as tabelas devem ser associadas em pares, embora seja possível utilizar um único comando para combinar várias tabelas.

É importante ressaltar que as tabelas devem ser associadas em pares, embora seja possível utilizar um único comando para combinar várias tabelas. Um procedimento muito comum é a associação da chave primária da primeira tabela com a chave estrangeira da segunda tabela.

Aprenderemos a utilizar as cláusulas **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, **FULL JOIN** e **CROSS JOIN** para promover a associação das tabelas. Também falaremos sobre o uso dos comandos **UPDATE** e **DELETE** em parceria com a cláusula **JOIN** para alteração e eliminação de dados nas tabelas relacionadas.

JOIN

A cláusula **JOIN** permite que os dados de várias tabelas sejam combinados com base na relação existente entre elas. Por meio dessa cláusula, os dados de uma tabela são utilizados para selecionar os dados pertencentes à outra tabela.

A partir da cláusula **FROM**, devemos indicar as tabelas que serão combinadas. A seleção de linhas de dados dessas tabelas, por sua vez, será controlada por meio de uma ação conjunta entre as condições de associação e de buscas como **HAVING** e **WHERE**.

Veja a sintaxe básica para associação:

... FROM [nome_tabela1] [tipo_associacao] [nome_tabela2] ON (condicao_associacao)

INNER JOIN

A partir da database criada no laboratório anterior, veremos um exemplo. Associaremos a tabela **CLIENTES** e a tabela **CIDADES**, como na tabela **CLIENTES** existe um campo com o código da cidade, por esse campo retornaremos o nome da cidade. Vejamos:

```
SELECT
    NOME_CLI,
    END_CLI,
    CID.NOME_CID
FROM CLIENTES AS CLI
    INNER JOIN CIDADES AS CID ON CID.COD_CID = CLI.COD_CID
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. The left pane displays the Object Explorer with the database structure, including the DB_CDS database. The right pane shows a query window titled "PC-MARCO\SQLEXP... - SQLQuery1.sql" containing the provided SQL code. Below the query window is a results grid showing the output of the query. The results grid has three columns: NOME_CLI, END_CLI, and NOME_CID. The data is as follows:

	NOME_CLI	END_CLI	NOME_CID
1	JOSÉ NOGUEIRA	RUA A	SÃO PAULO
2	ANGELO FERREIRA	RUA B	SÃO PAULO
3	ALÉM MAR PARANHOS	RUA C	SÃO PAULO
4	CATARINA SOUZA	RUA D	SÃO PAULO
5	VAGNER COSTA	RUA E	SÃO PAULO
6	ANTENOR DA COSTA	RUA F	SOROCABA
7	MARIA AMÉLIA DE SOUZA	RUA G	SOROCABA
8	PAULO ROBERTO DA SILVA	RUA H	SOROCABA
9	FATIMA DE SOUZA	RUA I	JUNDIAÍ
10	JOEL DA ROCHA	RUA J	JUNDIAÍ

É comum que duas tabelas tenham campos com o mesmo nome, como é o caso da chave estrangeira da tabela **CLIENTES** e a chave primária da tabela **CIDADES**. Para que possamos informar ao SQL Server de quais tabelas os campos estão sendo relacionados, podemos "apelidar" as tabelas. No caso descrito acima, a tabela **CLIENTES** foi apelidada de **CLI** e a tabela **CIDADES** foi apelidada de **CID**. "Apelidar" uma tabela é muito parecido com a criação de **ALIAS** para campos da tabela, só que nesse caso estamos nos referenciando ao nome das tabelas.

O processamento de uma associação atende a uma seqüência lógica, sendo que, em primeiro lugar, são executadas as condições de associação da cláusula **FROM**. Depois são aplicadas as condições de busca e de associação encontradas na cláusula **WHERE**. Por último, são executadas as condições de busca da cláusula **HAVING**, que veremos mais adiante.

Os tipos de dados das colunas vindas de tabelas associadas não precisam ser necessariamente idênticos, mas sim compatíveis, de modo que o SQL Server possa realizar

uma conversão. A condição de associação pode, ainda, utilizar a função **CAST** para converter tipos de dados que não podem ser transformados facilmente.

LEFT JOIN

A cláusula **LEFT JOIN** permite obter não apenas dados relacionados de duas tabelas, mas também os dados não-relacionados encontrados na tabela à esquerda da cláusula **JOIN**. Caso não existam dados relacionados entre as tabelas à esquerda e à direita do **JOIN**, os valores resultantes de todas as colunas de lista de seleção da tabela à direita serão nulos.

Vejamos um exemplo aplicado ao nosso database **DB_CDS**. Iremos retornar o nome de todos os clientes, com seus respectivos cônjuges e também aqueles clientes que não possuem cônjuge.

```
SELECT CLIENTES.NOME_CLI, CONJUGES.NOME_CONJ FROM CLIENTES
    LEFT JOIN CONJUGES ON CLIENTES.COD_CLI = CONJUGES.COD_CLI
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. The left pane displays the Object Explorer with the database 'PC-MARCO\SQLEXPRESS (SQL Server 9.0.30)' selected, showing its structure including databases, tables, and security. The right pane contains a query window titled 'PC-MARCO\SQLEXPRESS - SQLQuery1.sql' with the following code:

```
SELECT CLIENTES.NOME_CLI, CONJUGES.NOME_CONJ FROM CLIENTES
    LEFT JOIN CONJUGES ON CLIENTES.COD_CLI = CONJUGES.COD_CLI
GO
```

The 'Results' tab is selected, showing the output of the query:

	NOME_CLI	NOME_CONJ
1	JOSÉ NOGUEIRA	CARLA NOGUEIRA
2	ANGELO PEREIRA	EMILIA PEREIRA
3	ALÉM MAR PARANHOS	NULL
4	CATARINA SOUZA	NULL
5	VAGNER COSTA	NULL
6	ANTENOR DA COSTA	ALTIVA DA COSTA
7	MARIA AMÉLIA DE SOUZA	CARLOS DE SOUZA
8	PAULO ROBERTO DA SILVA	NULL
9	FATIMA DE SOUZA	NULL
10	JOEL DA ROCHA	NULL

RIGHT JOIN

Ao contrário do **LEFT JOIN**, a cláusula **RIGHT JOIN** retorna todos os dados encontrados na tabela à direita de **JOIN**. Caso não existam dados associados entre as tabelas à esquerda e à direita de **JOIN**, serão retornados valores nulos. Aplicando o mesmo exemplo do **LEFT JOIN** para o **RIGHT JOIN** teremos o seguinte:

```
SELECT CLIENTES.NOME_CLI, CONJUGES.NOME_CONJ FROM CLIENTES
    RIGHT JOIN CONJUGES ON CLIENTES.COD_CLI = CONJUGES.COD_CLI
GO
```

```

SELECT CLIENTES.NOME_CLI, CONJUGES.NOME_CONJ FROM CLIENTES
RIGHT JOIN CONJUGES ON CLIENTES.COD_CLI = CONJUGES.COD_CLI
GO

```

NOME_CLI	NOME_CONJ
1 JOSÉ NOGUEIRA	CARLA NOGUEIRA
2 ANGELO PEREIRA	EMILIA PEREIRA
3 ANTENOR DA COSTA	ALTIVA DA COSTA
4 MARIA AMÉLIA DE SOUZA	CARLOS DE SOUZA

Observe que foram retornados todos os dados da tabela **CONJUGES** com os respectivos relacionamentos. Como não existem cadastros de cônjuges sem clientes, não foi apresentado nenhum valor nulo na tabela à esquerda do **JOIN**.

Associando Mais de Duas Tabelas

Como vimos anteriormente, podemos fazer a associação de duas tabelas distintas para que possamos obter dados de uma "mescla" das duas. Agora, mostraremos como associar mais de duas tabelas. Vale lembrar que esse tipo de associação não tem limite.

Agora iremos retornar os dados de três tabelas: **CLIENTES**, **CIDADES** e **CONJUGES**. Iremos ter como resultado o nome do cliente que possui cônjuge, junto com o nome deste, e a cidade onde moram. Vejamos:

```

SELECT CLI.NOME_CLI, CONJ.NOME_CONJ, CID.NOME_CID FROM CLIENTES AS CLI
INNER JOIN CONJUGES AS CONJ ON CONJ.COD_CLI = CLI.COD_CLI
INNER JOIN CIDADES AS CID ON CID.COD_CID = CLI.COD_CID
GO

```

```

SELECT CLI.NOME_CLI, CONJ.NOME_CONJ, CID.NOME_CID FROM CLIENTES AS CLI
INNER JOIN CONJUGES AS CONJ ON CONJ.COD_CLI = CLI.COD_CLI
INNER JOIN CIDADES AS CID ON CID.COD_CID = CLI.COD_CID
GO

```

NOME_CLI	NOME_CONJ	NOME_CID
1 JOSÉ NOGUEIRA	CARLA NOGUEIRA	SÃO PAULO
2 ANGELO PEREIRA	EMILIA PEREIRA	SÃO PAULO
3 ANTENOR DA COSTA	ALTIVA DA COSTA	SOROCABA
4 MARIA AMÉLIA DE SOUZA	CARLOS DE SOUZA	SOROCABA

FULL OUTER JOIN

Todas as linhas de dados da tabela à esquerda de **JOIN** e da tabela à direita serão retornadas pela cláusula **FULL OUTER JOIN**. Caso uma linha de dados não esteja associada a qualquer linha da outra tabela, os valores das colunas da lista de seleção serão nulos. Caso contrário, os valores obtidos serão baseados nas tabelas utilizadas como referência. Vejamos:

```
SELECT CLI.NOME_CLI, CONJ.NOME_CONJ FROM CLIENTES AS CLI  
    FULL OUTER JOIN CONJUGES AS CONJ ON CONJ.COD_CLI = CLI.COD_CLI  
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'DB_CDS' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery1.sql' contains the following code:

```
SELECT CLI.NOME_CLI, CONJ.NOME_CONJ FROM CLIENTES AS CLI  
    FULL OUTER JOIN CONJUGES AS CONJ ON CONJ.COD_CLI = CLI.COD_CLI  
GO
```

The 'Results' tab is selected, displaying the following data:

	NOME_CLI	NOME_CONJ
1	JOSÉ NOGUEIRA	CARLA NOGUEIRA
2	ANGELO PEREIRA	EMILIA PEREIRA
3	ALÉM MAR PARANHOS	NULL
4	CATARINA SOUZA	NULL
5	VAGNER COSTA	NULL
6	ANTENOR DA COSTA	ALTIVA DA COSTA
7	MARIA AMÉLIA DE SOUZA	CARLOS DE SOUZA
8	PAULO ROBERTO DA SILVA	NULL
9	FATIMA DE SOUZA	NULL
10	JOEL DA ROCHA	NULL

CROSS JOIN

Todos os dados da tabela à esquerda de **JOIN** são cruzados com os dados da tabela à direita de **JOIN** por meio do **CROSS JOIN**, também conhecido como produto cartesiano. É possível cruzar informações de duas ou mais tabelas.

Para facilitar a compreensão a respeito desse tipo de associação, utilizaremos as tabelas **CLIENTES** e **CONJUGES**. Vejamos:

```
SELECT CLI.NOME_CLI, CONJ.NOME_CONJ FROM CLIENTES AS CLI  
    CROSS JOIN CONJUGES AS CONJ ORDER BY CLI.NOME_CLI  
GO
```

Execute esta consulta e observe o resultado.

Os Comandos UPDATE e DELETE Com Utilização do JOIN

Podemos utilizar a cláusula **JOIN** em conjunto com os comandos **UPDATE** e **DELETE**. Para exemplificar, tomemos primeiramente o comando **UPDATE**. Suponha que desejamos dar um aumento de 10% para os funcionários que conseguiram fazer alguma venda de CDs. Antes de utilizarmos o comando **UPDATE**, precisamos ter certeza de encontrar esses funcionários. Portanto, vamos selecioná-los com um **JOIN** entre as tabelas **PEDIDOS** e **FUNCIONARIOS**. Vejamos:

```
SELECT DISTINCT FUNC.COD_FUNC, FUNC.NOME_FUNC FROM FUNCIONARIOS AS FUNC  
INNER JOIN PEDIDOS AS PED ON PED.COD_FUNC = FUNC.COD_FUNC  
GO
```

Isso irá nos trazer a seguinte listagem:

COD_FUNC	NOME_FUNC
1	VANIA GABRIELA PEREIRA
2	NORBERTO PEREIRA DA SILVA
3	PAULA DA SILVA
4	ROLANDO ROCHA

Perceba que existem quatro funcionários nesta listagem, mas existem cinco registros na tabela **FUNCIONARIOS**. Veja a tabela funcionários:

```
SELECT * FROM FUNCIONARIOS  
GO
```

COD_FUNC	NOME_FUNC	END_FUNC	SAL_FUNC	SEXO_FUNC
1	VANIA GABRIELA PEREIRA	RUA A	2500.00	F
2	NORBERTO PEREIRA DA SILVA	RUA B	300.00	M
3	OLAVO LINHARES	RUA C	580.00	M
4	PAULA DA SILVA	RUA D	3000.00	F
5	ROLANDO ROCHA	RUA E	2000.00	M

Observe o salário dos funcionários antes de utilizarmos o comando **UPDATE**. Iremos aumentar em 10% todos os salários, exceto o do registro número '3', 'OLAVO LINHARES', pois este funcionário não fez nenhuma venda.

Agora, usaremos o comando **UPDATE** para realizar o aumento de salário para esses funcionários. Para isso, iremos fazer o comando de **UPDATE** e utilizaremos a parte do **JOIN** do comando de **SELECT** para fazer nossa restrição. Vejamos:

```
UPDATE FUNCIONARIOS SET SAL_FUNC = SAL_FUNC * 1.1 FROM FUNCIONARIOS AS FUNC
    INNER JOIN PEDIDOS AS PED ON PED.COD_FUNC = FUNC.COD_FUNC
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, a database named 'DB_CDS' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery1.sql' contains the following code:

```
UPDATE FUNCIONARIOS SET SAL_FUNC = SAL_FUNC * 1.1 FROM FUNCIONARIOS AS FUNC
    INNER JOIN PEDIDOS AS PED ON PED.COD_FUNC = FUNC.COD_FUNC
GO
```

The 'Results' tab is selected, displaying a table with five rows of data. The rows represent employees and their updated salaries. The third row, which corresponds to the employee 'OLAVO LINHARES', has a red box around it, indicating that this record was not updated.

COD_FUNC	NOME_FUNC	END_FUNC	SAL_FUNC	SEXO_FUNC
1	VANIA GABRIELA PEREIRA	RUA A	2750.00	F
2	NORBERTO PEREIRA DA SILVA	RUA B	330.00	M
3	OLAVO LINHARES	RUA C	580.00	M
4	PAULA DA SILVA	RUA D	3300.00	F
5	ROLANDO ROCHA	RUA E	2200.00	M

Observe que todos os funcionários tiveram um aumento de 10% em seus salários, exceto o funcionário 'OLAVO LINHARES'.

Com o comando **DELETE** não há muitas diferenças. Para visualizarmos um exemplo, crie uma database chamada **IDIOMAS**, ponha-a em uso e crie as tabelas **ALUNOS** e **CURSOS**:

Tabela: **ALUNOS**

COD_ALUNO PRIMARY KEY	NOME NOT NULL	COD_CURSO INT
1	JOAQUIM MANUEL	1
2	MARIA MADALENA	1
3	CARLITOS AMARAL	3
4	JOSÉ DA BRAGA E TINO	3

Tabela: **CURSOS**

COD_CURSO PRIMARY KEY	NOME NOT NULL
1	INGLES
2	ESPAÑOL
3	ALEMAO

Após criadas as tabelas e inseridos os registros, iremos utilizar um comando de **SELECT** com a cláusula **JOIN** para retornar todos os cursos que tenham alunos matriculados. Vejamos:

```
SELECT AL.NOME, CR.NOME FROM ALUNOS AS AL
    JOIN CURSOS AS CR ON CR.COD_CURSO = AL.COD_CURSO
GO
```

	NOME	NOME
1	JOAQUIM MANUEL	INGLES
2	MARIA MADALENA	INGLES
3	CARLITOS AMARAL	ALEMAO
4	JOSÉ DA BRAGA E TINO	ALEMAO

Agora iremos excluir todos os alunos do curso de inglês e para esse caso usaremos a cláusula restritiva **WHERE**. Vejamos o comando:

```
DELETE ALUNOS FROM ALUNOS AS AL
    JOIN CURSOS AS CR ON CR.COD_CURSO = AL.COD_CURSO
    WHERE CR.NOME = 'INGLES'
GO
```

Como podemos observar, em nossa aba **Messages**, os alunos do curso de inglês foram excluídos. Utilize um **SELECT** para verificar a tabelas **ALUNOS** e confirmar a exclusão.

Exercícios

1. Quais são as cláusulas utilizadas para fazer a associação de tabelas?

Resposta:

2. Qual a função da cláusula **INNER JOIN**?

Resposta:

3. Que cláusula gera os dados relacionados entre duas tabelas e os dados não-relacionados localizados na tabela posicionada à direita de **JOIN**?

Resposta:

4. Que faz a cláusula **CROSS JOIN**?

Resposta:

5. Quais comandos utilizar com o **JOIN** para alterar ou remover informações de uma tabela, tendo como base os dados de uma segunda tabela e estando elas inter-relacionadas?

Resposta:

Laboratório

Utilizando o banco de dados **DB_CDS**:

1. Selecione o nome dos CDs e o nome da gravadora de cada CD.

Resposta:

2. Selecione o nome dos CDs e o nome da categoria de cada CD.

Resposta:

3. Selecione o nome dos CDs, o nome das gravadoras de cada CD e o nome da categoria de cada CD.

Resposta:

4. Selecione o nome dos clientes e os títulos dos CDs vendidos em cada pedido que o cliente fez.

Resposta:

5. Selecione o nome do funcionário, número, data e valor dos pedidos que este funcionário registrou, além do nome do cliente que está fazendo o pedido.

Resposta:

6. Selecione o nome dos funcionários e o nome de todos os dependentes de cada funcionário.

Resposta:

7. Selecione o nome dos clientes e o nome dos cônjuges de cada cliente.

Resposta:

8. Selecione o nome de todos os clientes. Se estes possuem cônjuges, mostrar os nomes de seus cônjuges também.

Resposta:

9. Selecione nome do cliente, nome do cônjuge, número do pedido que cada cliente fez, valor de cada pedido que cada cliente fez e código do funcionário que atendeu a cada pedido.

Resposta:

Capítulo 9

No SQL Server, podemos consultar informações pertencentes a mais de uma tabela e que são obtidas com a execução de mais de um comando **SELECT**. Isso pode ser feito por meio da cláusula **UNION ALL**, que facilita a exibição de dados de tabelas distintas.

Uma consulta aninhada em uma instrução **SELECT**, **INSERT**, **DELETE** ou **UPDATE** é chamada de subquery (subconsulta), ou **O-QUE-NÃO-SE-DEVE-FAZER-NO-SQL-SERVER**. O limite máximo de aninhamentos de uma subquery é de 32 níveis, limite este que varia de acordo com a complexidade de outras instruções que compõem a consulta e da quantidade de memória disponível.

UNION ALL

A **UNION ALL** é uma cláusula responsável por unir informações obtidas a partir de diversos comandos de **SELECT**. Basicamente o **UNION ALL** concatena vários **SELECTs** a fim de montar um único comando que traga os dados de várias tabelas distintas. Para entender o **UNION ALL** considere a tabela **CLIENTES** e **FUNCIONARIOS** do nosso banco de dados **DB_CDS**. Supondo que precisamos obter o código, nome, endereço, renda/salário e sexo de cada pessoa nessas tabelas, podemos proceder da seguinte forma:

```
SELECT COD_CLI, NOME_CLI, END_CLI, RENDA_CLI, SEXO_CLI FROM CLIENTES
UNION ALL
SELECT COD_FUNC, NOME_FUNC, END_FUNC, SAL_FUNC, SEXO_FUNC FROM FUNCIONARIOS
GO
```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. The top menu bar includes File, Edit, View, Query, Tools, Window, Community, and Help. The toolbar below has icons for New Query, Execute, and various database management functions. The Object Explorer on the left shows the database structure for 'PC-MARCO\SQLEXPRESS (SQL Server 9.0.30)'. It lists 'Databases' (System Databases, DB_CDS), 'Tables' (System Tables, dbo.ARTISTAS, dbo.CATEGORIAS, dbo.CIDADES, dbo.CLIENTES, dbo.CONJUGES, dbo.DEPENDENTES, dbo.ESTADOS, dbo.FUNCIONARIOS, dbo.GRAVADORAS, dbo.PEDIDOS, dbo.TITULOS, dbo.TITULOS_ARTISTA, dbo.TITULOS_PEDIDO), 'Views', 'Synonyms', 'Programmability', and 'Security'. The main query window titled 'PC-MARCO\SQLEXP... - SQLQuery1.sql' contains the following T-SQL code:

```
SELECT COD_CLI, NOME_CLI, END_CLI, RENDA_CLI, SEXO_CLI FROM CLIENTES
UNION ALL
SELECT COD_FUNC, NOME_FUNC, END_FUNC, SAL_FUNC, SEXO_FUNC FROM FUNCIONARIOS
GO
```

The results pane shows a table with 15 rows of data, mapping the columns from the two tables:

	COD_CLI	NOME_CLI	END_CLI	RENDACLIENTES	SEXO_CLI
1	1	JOSE NOGUEIRA	RUA A	1500.00	M
2	2	ANGELO PEREIRA	RUA B	2000.00	M
3	3	ALÉM MAR PARANHOS	RUA C	1500.00	F
4	4	CATARINA SOUZA	RUA D	892.00	F
5	5	VAGNER COSTA	RUA E	950.00	M
6	6	ANTENOR DA COSTA	RUA F	1582.00	M
7	7	MARIA AMÉLIA DE SOUZA	RUA G	1152.02	F
8	8	PAULO ROBERTO DA SILVA	RUA H	3250.00	M
9	9	FATIMA DE SOUZA	RUA I	1632.00	F
10	10	JOEL DA ROCHA	RUA J	2000.00	M
11	1	VANIA GABRIELA PEREIRA	RUA A	2750.00	F
12	2	NORBERTO PEREIRA DA SILVA	RUA B	330.00	M
13	3	OLAVO LINHARES	RUA C	580.00	M
14	4	PAULA DA SILVA	RUA D	3300.00	F
15	5	ROLANDO ROCHA	RUA E	2200.00	M

Nota: Para que a consulta possa ser realizada, é fundamental que as colunas sejam do mesmo tipo.

Regras de Utilização

- O nome (alias) das colunas, quando realmente necessário, deve ser incluído no primeiro **SELECT**.
- A inclusão de **WHERE** pode ser feita em qualquer comando **SELECT**.
- É possível escrever qualquer **SELECT** com **JOIN** ou subquery, caso seja necessário.
- É necessário que todos os comandos **SELECT** utilizados apresentem o mesmo número de colunas.
- É necessário que todas as colunas dos comandos **SELECT** tenham o mesmo datatype em seqüência. Por exemplo, uma vez que a segunda coluna do primeiro **SELECT** baseia-se no datatype decimal, é preciso que as segundas colunas dos outros **SELECTs** também apresentem o mesmo tipo de datatype decimal.
- Para que tenhamos dados ordenados, o último **SELECT** deve ter uma cláusula **ORDER BY** adicionada em seu final.
- Devemos utilizar a cláusula **UNION** sem **ALL** para exibição única de dados repetidos em mais de uma tabela.

Subquery

As principais características das subquerys são:

- Pelo fato de podermos trabalhar com consultas estruturadas, as subquerys permitem que partes de um comando sejam separadas das demais.
- Se uma instrução é permitida em um local, esse local aceita a utilização de uma subquery.
- Uma subquery, pode ser incluída dentro de uma outra subquery (aninhamento), é identificada por estar entre parênteses, o que a diferencia da consulta principal.
- Quando temos uma subquery aninhada na instrução **SELECT** externa, ela é formada por uma cláusula **FROM** regular com um ou mais nomes de visualização ou tabela, uma consulta **SELECT** regular junto dos componentes da lista de seleção regular e as cláusulas opcionais **WHERE**, **HAVING** e **GROUP BY**.
- Uma consulta **SELECT** de uma subquery não pode ter uma cláusula **FOR BROWSE** ou **COMPUTE** (veremos à frente) incluída. Além disso, caso a cláusula **TOP** seja especificada, tal consulta, que está sempre entre parênteses, pode incluir apenas uma cláusula **ORDER BY**.
- As subquerys pode ser classificadas em: subqueries incluídas junto de um operador de comparação não modificado e que devem retornar um valor único, subqueries que são

testes de existência acrescentados com **EXISTS**, e subqueries que operam em listas introduzidas com **IN** ou que foram alteradas por um operador de comparação com **ALL** ou **ANY**.

- Por oferecer diversas formas de obter resultados, as subqueries eliminam a necessidade de utilizarmos as cláusulas **JOIN** e **UNION** de maior complexidade. Mas isso tem um preço, a performance das subqueries são extremamente baixas.
- Uma subquery também é chamada de **INNER QUERY** ou **INNER SELECT**.
- Chamamos de **OUTER QUERY** ou **OUTER SELECT** a instrução que possui a subquery.
- Alternativamente, é possível formular muitas instruções Transact-SQL com subqueries como **JOINS**.
- Uma instrução que possui uma subquery, bem como uma versão semanticamente semelhante a essa instrução mas que não possui subquery, normalmente não apresenta muitas diferenças de performance. No entanto, uma **JOIN** apresenta melhor desempenho nas situações em que se precisa verificar a existência de um dado.
- As colunas de uma tabela não poderão ser incluídas na saída, ou seja, a lista de seleção da **OUTER QUERY**, caso essa tabela apareça apenas em uma subquery e não na **OUTER QUERY**.

Formatos Apresentados Comumente Pelas Subqueries

- **WHERE [expressao] [NOT] IN (subquery)**
- **WHERE [expressao] [operador_comparacao] [ANY | ALL] (subquery)**
- **WHERE [NOT] EXISTS (subquery)**

Exemplo de Subquery

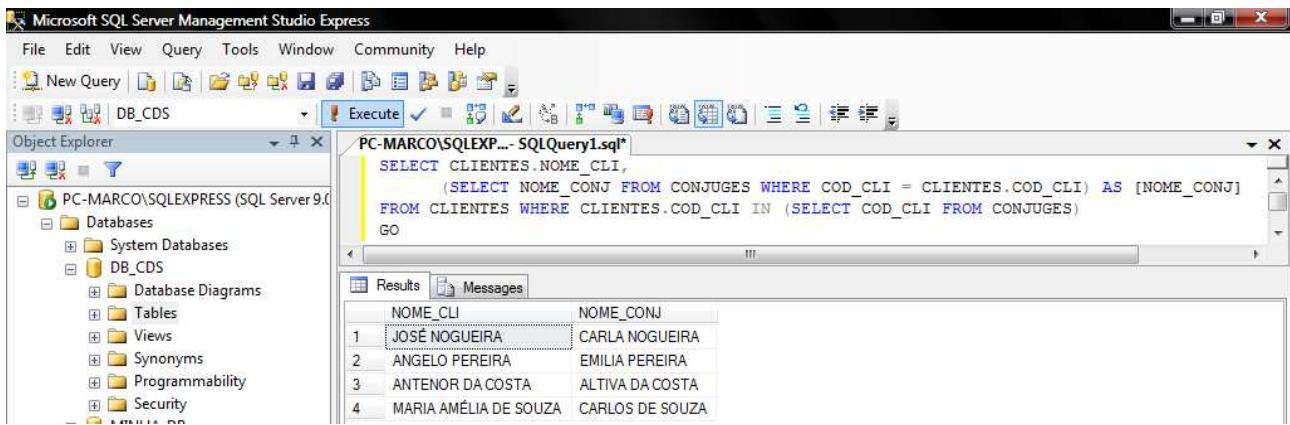
Como vimos em um exemplo anterior, com a cláusula **JOIN** podemos retornar todos os clientes que possuem cônjuge. Vejamos essa instrução:

```
SELECT CLI.NOME_CLI, CONJ.NOME_CONJ FROM CLIENTES AS CLI  
INNER JOIN CONJUGES AS CONJ ON CONJ.COD_CLI = CLI.COD_CLI  
GO
```

Fazendo uso de uma subquery, a nossa instrução SQL ficaria dessa forma:

```
SELECT CLIENTES.NOME_CLI, (SELECT NOME_CONJ FROM CONJUGES WHERE COD_CLI =  
CLIENTES.COD_CLI) AS [NOME_CONJ] FROM CLIENTES WHERE CLIENTES.COD_CLI IN (SELECT  
COD_CLI FROM CONJUGES)  
GO
```

E obteríamos o mesmo resultado. Observe a imagem abaixo:



```
PC-MARCO\SQLEXP... - SQLQuery1.sql
SELECT CLIENTES.NOME_CLI,
       (SELECT NOME_CONJ FROM CONJUGES WHERE COD_CLI = CLIENTES.COD_CLI) AS [NOME_CONJ]
FROM CLIENTES WHERE CLIENTES.COD_CLI IN (SELECT COD_CLI FROM CONJUGES)
GO
```

	NOME_CLI	NOME_CONJ
1	JOSE NOGUEIRA	CARLA NOGUEIRA
2	ANGELO PEREIRA	EMILIA PEREIRA
3	ANTENOR DA COSTA	ALTIVA DA COSTA
4	MARIA AMÉLIA DE SOUZA	CARLOS DE SOUZA

Como podemos ver, as duas nos retornaram os mesmos dados. E como podemos ver, também, a solução mais elegante é utilizar o **JOIN**.

Regras de Utilização

- As subqueries devem ser utilizadas entre parênteses.
- Ao utilizarmos subqueries, podemos obter apenas uma coluna por subquery.
- Um único valor será retornado ao utilizarmos o sinal (=) no início da subquery.

Exercícios

1. Qual o objetivo da cláusula **UNION ALL**?

Resposta:

2. Qual a diferença entre a cláusula **UNION ALL** e **UNION**?

Resposta:

3. Explique o conceito de subquery.

Resposta:

4. Cite as regras básicas para que as subqueries retornem o valor esperado.

Resposta:

5. Qual vantagem que podemos obter com o uso de subquery?

Resposta:

Capítulo 10

Totalizando Dados

Veremos como a cláusula **GROUP BY** pode ser utilizada para agrupar vários dados, tornando mais prática a totalização dos mesmos. Também conhceremos outras cláusulas que podem ser empregadas em parceria com o **GROUP BY**, como **HAVING**, **WITH ROLLUP** e **JOIN**.

GROUP BY

Com a cláusula **GROUP BY** é possível agrupar diversos registros com base em uma ou mais colunas. Por exemplo, na nossa database **DB_CDS**, mais de um título é produzido por uma gravadora, caso seja necessário saber qual a quantidade de títulos que essa gravadora produziu, podemos usar o **GROUP BY** em associação com o **JOIN** para obtermos esses dados.

É importante lembrar que o **GROUP BY** é freqüentemente utilizado em conjunto com as funções:

- **SUM()**: Função que permite a soma dos registros.
- **COUNT()**: Permite a contagem de registros.
- **MAX()**: Retorna o maior valor de um conjunto de valores.
- **MIN()**: Retorna o menor valor de um conjunto de valores.
- **AVG()**: Calcula a média dos valores de um conjunto.

Nas situações em que se especifica a cláusula **GROUP BY**, deve ocorrer uma das seguintes situações: a expressão **GROUP BY** deve ser correspondente à expressão da lista de seleção ou cada uma das colunas presentes em uma expressão não-agregada na lista de seleção deve ser adicionada à lista de **GROUP BY**.

Quando utilizamos o **GROUP BY**, nos é retornado grupos em ordem aleatória, nesse caso podemos utilizar a cláusula **ORDER BY** em conjunto para ordená-los.

A sintaxe básica da cláusula **GROUP BY** é a seguinte:

[comando] [GROUP BY [ALL] expressao_group_by [,...n] [WITH { CUBE | ROLLUP }]]

Em que:

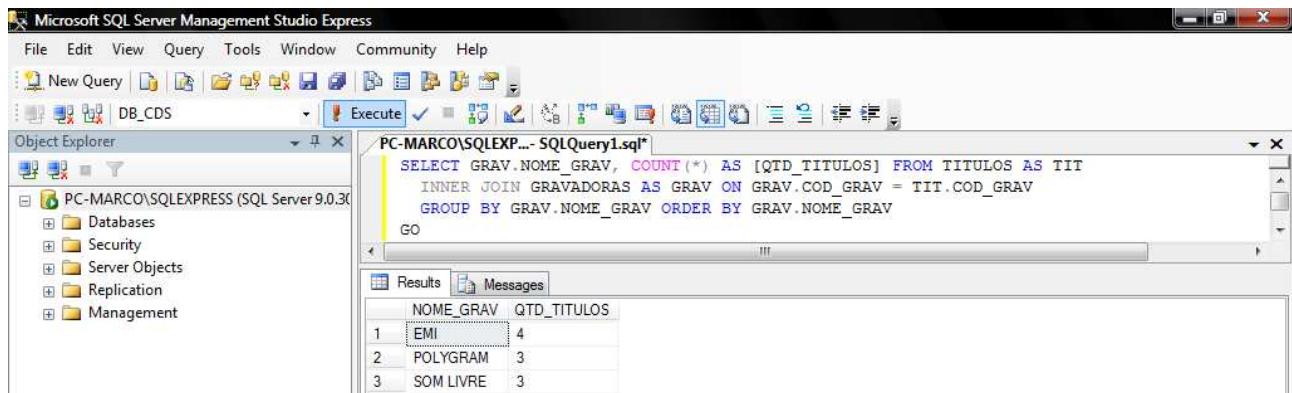
- **ALL**: Trata-se do argumento que determina a inclusão de todos os grupos e conjuntos de resultados. Vale destacar que valores nulos são retornados às colunas resultantes dos grupos que não correspondem aos critérios de busca quando o argumento **ALL** é especificado.

- **expressao_group_by:** Também conhecida como coluna agrupada, trata-se de uma expressão na qual o argumento é realizado. Ela pode ser especificada como uma coluna ou como uma expressão não-agregada que faz referência à coluna que a cláusula **FROM** retornou, mas não é possível especificá-la com uma alias de coluna determinado na lista de seleção. Além disso, não podemos utilizar a **expressao_group_by** as colunas dos seguintes tipos: **IMAGE**, **TEXT** e **NTEXT**.

Nota: É importante considerar que as queries que acessam tabelas remotas não são capazes de suportar a cláusula **GROUP BY ALL** caso essas queries também possam uma cláusula **WHERE**. Devemos ter em mente, ainda, que o argumento **ALL** não pode ser utilizado em conjunto com os operadores **CUBE** e **ROLLUP**. A quantidade de itens da **expressao_group_by** é limitada de acordo com o tamanho apresentado pelas colunas **GROUP BY**, com as colunas agregadas e com os valores referentes à query nas situações em que a cláusula **GROUP BY** não possuem os operadores **CUBE** e **ROLLUP**. Nas situações em que tais operadores são especificados, são permitidas até 10 expressões de agrupamento.

Para compreender, vejamos o exemplo citado acima:

```
SELECT GRAV.NOME_GRAV, COUNT(*) AS [QTD_TITULOS] FROM TITULOS AS TIT
INNER JOIN GRAVADORAS AS GRAV ON GRAV.COD_GRAV = TIT.COD_GRAV
GROUP BY GRAV.NOME_GRAV ORDER BY GRAV.NOME_GRAV
GO
```



HAVING Com GROUP BY

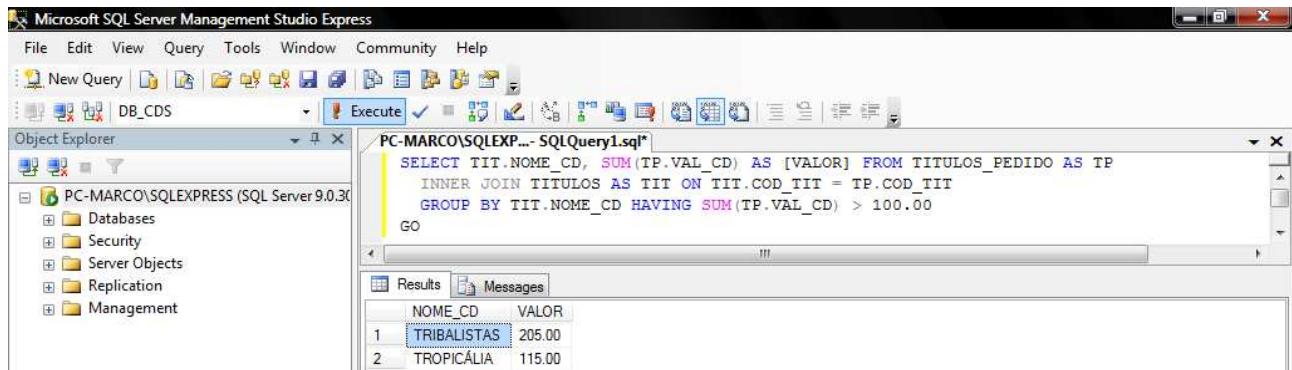
A cláusula **HAVING** determina uma condição de busca para um grupo ou um conjunto de registros, definindo critérios para limitar os resultados obtidos a partir do agrupamento de registros. É importante lembrar que essa cláusula só pode ser utilizada em parceria com **GROUP BY**.

Nota: A cláusula **HAVING** é diferente da cláusula **WHERE**. Esta última

restringe os resultados obtidos após a aplicação da cláusula **FROM**, ao passo que a cláusula **HAVING** filtra o retorno do agrupamento.

Como exemplo, iremos agrupar todos os títulos de CDs que venderam mais de 100.00:

```
SELECT TIT.NOME_CD, SUM(TP.VAL_CD) AS [VALOR] FROM TITULOS_PEDIDO AS TP
INNER JOIN TITULOS AS TIT ON TIT.COD_TIT = TP.COD_TIT
GROUP BY TIT.NOME_CD HAVING SUM(TP.VAL_CD) > 100.00
GO
```



WITH ROLLUP Com GROUP BY

Esta cláusula determina que, além das linhas normalmente oferecidas por **GROUP BY**, também sejam obtidas como resultado as linhas de sumário. O sumário dos grupos é feito em uma ordem hierárquica, a partir do nível mais baixo até o mais alto. A ordem que define a hierarquia do grupo é determinada pela ordem na qual são definidas as colunas agrupadas. Caso esta ordem seja alterada, a quantidade de linhas produzidas pode ser afetada.

Utilizada em parceria com a cláusula **GROUP BY**, a cláusula **WITH ROLLUP** acrescentar uma linha na qual são exibidos os subtotais e totais dos registros já distribuídos em colunas agrupadas.

Dessa forma, é possível analisar diversas informações em uma só coluna, por exemplo, a gravadora que possui a menor quantidade de títulos em estoque. Vejamos:

```
SELECT
    GRAV.NOME_GRAV,
    TIT.NOME_CD, TIT.QTD_ESTQ,
    SUM(TIT.QTD_ESTQ) AS [TOTAL_ESTQ]
    FROM TITULOS AS TIT
    INNER JOIN GRAVADORAS AS GRAV ON GRAV.COD_GRAV = TIT.COD_GRAV
    GROUP BY GRAV.NOME_GRAV, TIT.NOME_CD, TIT.QTD_ESTQ WITH ROLLUP
GO
```

Observe a imagem abaixo:

The screenshot shows the Microsoft SQL Server Management Studio Express interface. The Object Explorer pane on the left displays the database structure for 'PC-MARCO\SQLEXPRESS (SQL Server 9.0)'. The central pane shows a query window titled 'PC-MARCO\SQLEXPRESS... - SQLQuery1.sql*' containing the following T-SQL code:

```
SELECT
    GRAV.NOME_GRAV,
    TIT.NOME_CD, TIT.QTD_ESTQ,
    SUM(TIT.QTD_ESTQ) AS [TOTAL_ESTQ]
    FROM TITULOS AS TIT
    INNER JOIN GRAVADORAS AS GRAV ON GRAV.COD_GRAV = TIT.COD_GRAV
    GROUP BY GRAV.NOME_GRAV, TIT.NOME_CD, TIT.QTD_ESTQ WITH ROLLUP
GO
```

The Results pane below the query window displays the output of the query. The results are as follows:

	NOME_GRAV	NOME_CD	QTD_ESTQ	TOTAL_ESTQ
1	EMI	COURAGE	200	200
2	EMI	COURAGE	NULL	200
3	EMI	REFAZENDA	1000	1000
4	EMI	REFAZENDA	NULL	1000
5	EMI	THE BEATLES	300	300
6	EMI	THE BEATLES	NULL	300
7	EMI	TROPICÁLIA	500	500
8	EMI	TROPICÁLIA	NULL	500
9	EMI	NULL	NULL	2000
10	POLYGRAM	AQUELE ABRÃO	600	600
11	POLYGRAM	AQUELE ABRÃO	NULL	600
12	POLYGRAM	RITA LEE	500	500
13	POLYGRAM	RITA LEE	NULL	500
14	POLYGRAM	TRIBALISTAS	1500	1500
15	POLYGRAM	TRIBALISTAS	NULL	1500
16	POLYGRAM	NULL	NULL	2600
17	SOM LIVRE	LEGIÃO URBANA	100	100
18	SOM LIVRE	LEGIÃO URBANA	NULL	100
19	SOM LIVRE	TOTALMENTE ...	2000	2000
20	SOM LIVRE	TOTALMENTE ...	NULL	2000

The status bar at the bottom indicates 'Query executed successfully.' and other details.

WITH CUBE Com GROUP BY

A cláusula **CUBE** tem a finalidade de determinar que as linhas de sumário sejam inseridas no conjunto de resultados. A linha de sumário é retornada para cada combinação possível e de subgrupos no conjunto de resultados.

Visto que a cláusula **CUBE** é responsável por retornar todas as combinações possíveis de grupos e subgrupos, a quantidade de linhas não está relacionada à ordem em que são determinadas as colunas de agrupamento, sendo, portanto, mantida a quantidade de linhas já apresentadas.

A quantidade de linhas de sumário no conjunto de resultados é especificada de acordo com a quantidade de colunas incluídas na cláusula **GROUP BY**. Cada uma dessas colunas é vinculada sob o valor **NULL** do agrupamento, o qual é aplicado a todas as outras colunas.

A cláusula **WITH CUBE**, em conjunto à **GROUP BY**, gera totais e subtotais, apresentando vários agrupamentos de acordo com as colunas definidas com o **GROUP BY**. Vejamos o mesmo exemplo anterior, só que com **WITH CUBE**:

```

SELECT
    GRAV.NOME_GRAV,
    TIT.NOME_CD, TIT.QTD_ESTQ,
    SUM(TIT.QTD_ESTQ) AS [TOTAL_ESTQ]
    FROM TITULOS AS TIT
INNER JOIN GRAVADORAS AS GRAV ON GRAV.COD_GRAV = TIT.COD_GRAV
GROUP BY GRAV.NOME_GRAV, TIT.NOME_CD, TIT.QTD_ESTQ WITH CUBE
GO

```

The screenshot shows the Microsoft SQL Server Management Studio Express interface. The Object Explorer on the left shows the database structure, including the DB_CDS database which contains tables like GRAVADORAS and TITULOS. The central pane displays the SQL query:

```

SELECT
    GRAV.NOME_GRAV,
    TIT.NOME_CD, TIT.QTD_ESTQ,
    SUM(TIT.QTD_ESTQ) AS [TOTAL_ESTQ]
    FROM TITULOS AS TIT
INNER JOIN GRAVADORAS AS GRAV ON GRAV.COD_GRAV = TIT.COD_GRAV
GROUP BY GRAV.NOME_GRAV, TIT.NOME_CD, TIT.QTD_ESTQ WITH CUBE
GO

```

The Results pane shows the output of the query, which includes 62 rows of data from the TITULOS and GRAVADORAS tables, grouped by GRAV.NOME_GRAV, TIT.NOME_CD, and TIT.QTD_ESTQ, and including a calculated column [TOTAL_ESTQ]. The results are as follows:

	NOME_GRAV	NOME_CD	QTD_ESTQ	TOTAL_ESTQ
1	EMI	COURAGE	200	200
2	EMI	COURAGE	NULL	200
3	EMI	REFAZENDA	1000	1000
4	EMI	REFAZENDA	NULL	1000
5	EMI	THE BEATLES	300	300
6	EMI	THE BEATLES	NULL	300
7	EMI	TROPICÁLIA	500	500
8	EMI	TROPICÁLIA	NULL	500
9	EMI	NULL	NULL	2000
10	POLYGRAM	AQUELE ABRAÇO	600	600
11	POLYGRAM	AQUELE ABRAÇO	NULL	600
12	POLYGRAM	RITA LEE	500	500
13	POLYGRAM	RITA LEE	NULL	500
14	POLYGRAM	TRIBALISTAS	1500	1500
15	POLYGRAM	TRIBALISTAS	NULL	1500
16	POLYGRAM	NULL	NULL	2600
17	SOM LIVRE	LEGIÃO URBANA	100	100
18	SOM LIVRE	LEGIÃO URBANA	NULL	100
19	SOM LIVRE	TOTALMENTE DEMAIS	2000	2000
20	SOM LIVRE	TOTALMENTE DEMAIS	NULL	2000

The status bar at the bottom right indicates "62 rows".

Observe que o retorno de linhas é superior ao exemplo anterior, pois nas linhas abaixo, a cláusula **WITH CUBE** fez todas as combinações possíveis com os dados passados.

Exercícios

1. Qual o objetivo da cláusula **GROUP BY**?

Resposta:

2. Explique a cláusula **HAVING**.

Resposta:

3. Qual é a cláusula que, utilizada em parceria com a cláusula **GROUP BY**, acrescenta uma linha na qual são exibidos os subtotais e totais dos registros já distribuídos em colunas agrupadas?

Resposta:

4. Que faz a cláusula **JOIN**?

Resposta:

5. Qual a diferença entre as cláusulas **HAVING** e **WHERE**?

Resposta:

Laboratório

Para realizar esse laboratório, utilize o banco de dados **DB_CDS**.

1. Exiba quantos pedidos cada cliente fez.

Resposta:

2. Exiba quantos CDs possui cada categoria.

Resposta:

3. Exiba quantos CDs possui cada gravadora.

Resposta:

4. Exiba quantos pedidos cada funcionário atendeu.

Resposta:

5. Exiba quantos dependentes tem cada funcionário.

Resposta:

6. Exiba quantos pedidos cada cliente fez, mostrando o nome dos clientes.

Resposta:

7. Exiba quantos CDs possui cada categoria, mostrando o nome das mesmas.

Resposta:

8. Exiba quantos CDs possui cada gravadora, mostrando o nome das mesmas.

Resposta:

9. Exiba quantos pedidos cada funcionário atendeu, mostrando o nome dos funcionários.

Resposta:

10. Exiba quantos dependentes cada funcionário possui, mostrando seus nomes.

Resposta:

Capítulo 11

A partir deste capítulo, iremos mais afundo nos aspectos técnicos do SQL Server 2005 e veremos a complementação de comandos vistos anteriormente.

Entendendo os Arquivos de Dados

Um database criado pelo SQL Server 2005 é formado por, no mínimo, dois arquivos de sistema: um arquivo de dados e um arquivo de armazenamento de log. Porém , tal database pode ter três ou mais arquivos dependendo da necessidade.

Arquivo Primário (PRIMARY DATA FILE)

Apresenta a extensão **.mdf** como padrão. O **PRIMARY DATA FILE** aponta para os demais arquivos do database. É no arquivo de dados primário, além do database **MASTER**, que é registrada a localização de todos os arquivos no database. Porém, na maioria das vezes, o **DATABASE ENGINE** utiliza as informações de localização provenientes do database **MASTER**.

Essas informações, do **PRIMARY DATA FILE**, são usadas pelo **DATABASE ENGINE** quando:

- O database **MASTER** está sendo restaurado.
- Quando é realizada uma atualização do SQL Server 7.0 ou 2000 para versão 2005. (Não necessariamente o Express).
- Quando se utiliza a instrução **CREATE DATABASE** com a opção **FOR ATTACH** ou **FOR_ATTACH_REBUILD_LOG** para anexar databases.

Arquivo Secundário (SECONDARY DATA FILE)

Apresenta a extensão **.ndf**, representa todos os arquivos de dados, exceto o primário. Um database pode ter um, vários ou nenhum **SECONDARY DATA FILE**.

Arquivo de Log (LOG DATA FILE)

Nesse tipo de arquivo são armazenadas as informações necessárias para a recuperação de transações em um banco de dados. O database deve possuir pelo menos um arquivo de log. A extensão padrão é **.ldf**.

Nota: Quando "dropamos" um database, esses arquivos são excluídos automaticamente.

Nome Físico e Lógico

Um arquivo do SQL Server é formado por dois nomes: um nome físico e outro lógico. O nome físico é utilizado para se referir ao arquivo físico no disco, enquanto o nome lógico é o nome utilizado para nos referirmos aos arquivos físicos em todas as instruções do Transact-SQL.

Grupos de Arquivos (FILEGROUP)

Os grupos de arquivos, também chamados de **FILEGROUPs**, são utilizados para melhorar o gerenciamento e a localização dos arquivos e objetos do database. Por meio de **FILEGROUPs**, objetos específicos podem ser inseridos em arquivos específicos.

Um database possui pelo menos um **FILEGROUP**. Trata-se do **FILEGROUP** primário. Outros **FILEGROUPs** adicionais podem ser criados pelo usuário para atender necessidades específicas, como alocação e administração de dados.

Outra facilidade oferecida pelo **FILEGROUPs** é a de backup: podemos fazer cópias de segurança de arquivos e grupos de arquivos específicos sem precisar criar o backup de todo o database.

Alguns sistemas podem ter um ganho de performance por meio de **FILEGROUPs**, os quais, nesta situação, auxiliam o controle da localização de índices e arquivos de dados em discos específicos.

A utilização de **FILEGROUPs** pode ser útil caso estejamos trabalhando com uma configuração de hardware caracterizada por vários discos rígidos (disk arrays). Neste caso, os discos podem ser preparados para alocar objetos e arquivos específicos, que por sua vez, estarão contidos em **FILEGROUPs**.

Existem dois tipos de **FILEGROUPs**: primário (padrão) e definidos (secundários) pelo usuário. O **FILEGROUP** primário contém os arquivos de dados primários, além de arquivos

que não foram atribuídos a outros **FILEGROUPs** e aloca as páginas das tabelas do sistema. Vale ressaltar que é possível alterar o **FILEGROUP** padrão, porém a alocação de tabelas e objetos de sistema é mantida no **FILEGROUP** primário. Já os **FILEGROUPs** definidos pelo usuário tem como objetivo armazenar os dados, para fins administrativos, de posicionamento ou performance de dados.

Regras dos FILES e FILEGROUPs

- Pelo fato de os espaços de log e de dados serem controlados separadamente, os arquivos de log nunca ocupam um **FILEGROUP**.
- Mais de um database não pode utilizar um mesmo arquivo ou **FILEGROUP**.
- Nunca pode haver, simultaneamente, mais de um **FILEGROUP** como padrão.
- Um arquivo não pode ser membro de mais de um **FILEGROUP**.

Recomendações de Uso

Tamanho dos Arquivos

Ao definirmos um arquivo no SQL Server, podemos especificar um incremento para o seu crescimento. Isso porque os arquivos do SQL Server possuem a capacidade de aumentar seu tamanho, de forma automática.

O arquivo aumentará de tamanho de acordo com o incremento especificado e toda vez que seu espaço for preenchido. Mas é possível especificar o tamanho do crescimento.

Múltiplos Arquivos

- Armazenamento de tabelas e objetos do sistema no **PRIMARY DATA FILE**.
- Armazenamento de objetos criados pelo usuário no **SECONDARY DATA FILE**.

Tabelas

Grandes dados de objetos, assim como tabelas e índices, podem ser associados a um **FILEGROUP** específico. Essas tabelas e índices podem ser particionados. Quando isso é feito, os dados são separados em unidades, sendo que cada uma das unidades pode ser colocada em um **FILEGROUP** distinto, no database.

Para obter uma performance melhor, recomenda-se separar em **FILEGROUPs** distintos as diferentes tabelas utilizadas nas mesmas consultas **JOIN**, para se obter um acesso paralelo aos dados. Também é aconselhável separar em **FILEGROUPs** distintos as tabelas de maior acesso e os índices **NONCLUSTERED** pertencentes a essas tabelas. Com os arquivos localizados em discos diferentes, temos um acesso I/O paralelo, o que melhora em muito a performance do sistema.

Separando o Log e Arquivo de Dados

É recomendável manter em discos rígidos separados os arquivos de log dos outros arquivos de dados. Podemos também manter tabelas específicas, muito pouco utilizadas, em arquivos de **FILEGROUPs** distintos e posicionar cada um desses **FILEGROUPs** em discos distintos. Dessa forma as operações de I/O de tabelas específicas ficam de forma paralela.

Exemplo de Criação de Database

```
CREATE DATABASE CAP_11
    ON PRIMARY
(
    NAME = 'CAP_11_DADOS_1',
    FILENAME = 'C:\CAP_11_DADOS_1.MDF',
    SIZE = 10MB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 10MB
),
(
    NAME = 'CAP_11_DADOS_2',
    FILENAME = 'D:\CAP_11_DADOS_2.NDF',
    SIZE = 20MB,
    MAXSIZE = 500MB,
    FILEGROWTH = 20MB
),
FILEGROUP FG_01
(
    NAME = 'CAP_11_DADOS_3',
    FILENAME = 'E:\CAP_11_DADOS_3.NDF',
    SIZE = 10MB,
    MAXSIZE = 250MB,
    FILEGROWTH = 20%
),
(
    NAME = 'CAP_11_DADOS_4',
    FILENAME = 'F:\CAP_11_DADOS_4.NDF',
    SIZE = 10MB,
    MAXSIZE = 250MB,
    FILEGROWTH = 20%
),
```

```

FILEGROUP FG_02
(
    NAME = 'CAP_11_DADOS_5',
    FILENAME = 'G:\CAP_11_DADOS_5.NDF',
    SIZE = 10MB,
    MAXSIZE = 1000MB,
    FILEGROWTH = 20%
)

LOG ON
(
    NAME = 'CAP_11_LOG_1',
    FILENAME = 'H:\CAP_11_LOG_1.LDF',
    SIZE = 50MB,
    MAXSIZE = 250MB,
    FILEGROWTH = 50MB
),
(
    NAME = 'CAP_11_LOG_2',
    FILENAME = 'I:\CAP_11_LOG_2.LDF',
    SIZE = 50MB,
    MAXSIZE = 250MB,
    FILEGROWTH = 50MB
)
GO

```

Criando Tabelas Dentro de FILEGROUPs

Como mencionamos anteriormente, é possível colocar tabelas específicas dentro de **FILEGROUPs** definidos pelo usuário, para isso, basta que no final do script de criação da tabela especifiquemos em qual **FILEGROUP** essa tabela deve ficar. Vejamos:

```

CREATE TABLE MEU_TESTE
(
    CODIGO INT IDENTITY(1,1) NOT NULL PRIMARY KEY,
    DADOS VARCHAR(8000) NOT NULL
)
ON FG_01
GO

```

Também há a possibilidade de se alterar o **FILEGROUP** de uma tabela já criada, veja o exemplo:

```

ALTER TABLE MEU_TESTE
    MODIFY FILEGROUP FG_01 DEFAULT

```

Transaction Log

O Transaction Log, ou log de transações, é definido como um registro seria de todas as alterações sofridas pelo database. O log de transações é mantido no arquivo de log (**LOG DATA FILE**) do database.

O procedimento a seguir demonstra o que acontece no Transaction Log no momento em que uma transação é iniciada:

1. A aplicação envia uma transação de dados
2. No momento em que a transação é executada, as páginas de dados afetadas são carregadas do disco para o **DATA CACHE** (memória). Caso uma query anterior já tenha carregado essas páginas no **DATA CACHE**, o carregamento não irá ocorrer.
3. O **WRITE-AHEAD LOG** registra as alterações antes delas serem encaminhadas para o database.
4. Por um processo denominada **CHECKPOINT**, as alterações registradas no Transaction Log são escritas nas tabelas correspondentes, no database.

Caso exista algum problema no meio de uma transação, quando o SQL Server é iniciado novamente, ele procura pelo Transaction Log as transações não finalizadas (transações não marcadas com o **CHECKPOINT**). Esse processo é denominado **RECOVERY**. Para que seja possível identificar uma transação, é necessário informar o SQL Server. Para tal, usamos as instruções **BEGIN TRANSACTION**, para marcar o início da transação e **COMMIT TRANSACTION** para finalizar a transação com sucesso. Caso seja necessário abortar a transação, usamos o comando **ROLLBACK TRANSACTION**.

Exercícios

1. Que significa nome físico e nome lógico de um arquivo?

Resposta:

2. Que são **FILEGROUPs**?

Resposta:

3. Quais são os tipos de **FILEGROUPs** existentes?

Resposta:

4. Qual a função do procedimento **RECOVERY**?

Resposta:

5. Quais das alternativas a seguir são características dos **FILEGROUPs**?
- a. Melhorar o gerenciamento e a localização de tabelas e objetos.
 - b. Podemos ter simultaneamente mais de um **FILEGROUP** como padrão.
 - c. Um database possui pelo menos um **FILEGROUP**, trata-se do **FILEGROUP** primário. Outros **FILEGROUPs** adicionais podem ser criados pelo usuário para atender a necessidades específicas, como alocação e administração de dados.
 - d. Um arquivo pode ser membro de mais de um **FILEGROUP**.
 - e. Podemos fazer cópias de segurança de arquivos e grupos de arquivos específicos sem precisar fazer o backup de todo o database.
6. Qual das alternativas a seguir é uma afirmativa correta com relação às regras atribuídas aos files e **FILEGROUPs**?
- a. Pelo fato de os espaços de log e de dados serem controlados juntos, os arquivos de log sempre ocupam um **FILEGROUP**.
 - b. Podemos ter simultaneamente mais de um **FILEGROUP** como padrão.
 - c. Um arquivo não pode ser membro de mais de um **FILEGROUP**
 - d. Dois ou mais databases não podem utilizar o mesmo arquivo, porém, podem utilizar o mesmo **FILEGROUP**.
 - e. Todas as alternativas anteriores estão erradas.
7. Qual das alternativas a seguir é correta com relação aos arquivos físicos gerados pelo SQL Server?
- a. Ao definirmos um arquivo no SQL Server 2005, não podemos especificar um incremento para o crescimento de um arquivo.
 - b. O arquivo poderá aumentar de tamanho de acordo com o incremento especificado pelo administrador do banco de dados.
 - c. Um banco de dados criado pelo SQL Server deve obrigatoriamente ter como nome físico o mesmo nome atribuído ao nome lógico.
 - d. É aconselhável manter em um mesmo **FILEGROUP** tabelas que são muito acessadas.
 - e. Para que os arquivos aumentem automaticamente o seu tamanho, não é necessário que eles sejam totalmente preenchidos.

8. Qual a definição de Transaction Log?

- a. O Transaction Log ou log de transações, é definido como um local no qual é registrada a localização de todos os arquivos no database
- b. O Transaction Log é um arquivo que guarda somente as tabelas de um database.
- c. O Transaction Log, ou log de transações, é definido como um registro serial de todas as alterações que o database sofreu. O log de transações é mantido no arquivo de log de transações do database.
- d. O Transaction Log é uma tabela que guarda informações sobre erros ocorridos em transações.
- e. Transaction Log é um **TRIGGER** que dispara quando ocorre erro nas transações correntes.

9. Qual o procedimento correto de registro no Transaction Log?

A.

- i. A aplicação envia uma transação de dados
- ii. No momento em que a transação é executada, as páginas de dados afetadas são carregadas do disco para o **DATA CACHE** (memória). Caso uma query anterior já tenha carregado essas páginas no **DATA CACHE**, o carregamento não ocorrerá.
- iii. Ocorre o **WRITE-AHEAD LOG**
- iv. Ocorre o **CHECKPOINT**

B.

- i. A aplicação envia uma transação de dados
- ii. No momento em que a transação é executada, as tabelas afetadas são carregadas do disco para o Transaction Log. Caso uma query anterior já tenha carregado essas páginas no **DATA CACHE**, o carregamento não ocorrerá.
- iii. Ocorre o **WRITE-AHEAD LOG**
- iv. Ocorre o **CHECKPOINT**

C.

- i. A aplicação envia uma transação de dados
- ii. No momento em que a transação é executada, as páginas de dados afetadas são carregadas do disco para o **DATA CACHE** (memória). Caso uma query anterior já tenha carregado essas páginas no **DATA CACHE**, o carregamento não ocorrerá.
- iii. Ocorre o **CHECKPOINT**
- iv. Ocorre o **WRITE-AHEAD LOG**

D.

- i. A aplicação envia uma transação de dados
- ii. No momento em que a transação é executada, as páginas de dados afetadas são carregadas do disco para o **DATA CACHE** (memória). Caso uma query anterior já tenha carregado essas páginas no **DATA CACHE**, o carregamento não ocorrerá.
- iii. Ocorre o **RECOVERY**
- iv. Ocorre o **WRITE-AHEAD LOG**

E.

- i. A aplicação envia uma transação de dados
- ii. No momento em que a transação é executada, as páginas de dados afetadas são carregadas do disco para o **DATA CACHE** (memória). Caso uma query anterior já tenha carregado essas páginas no **DATA CACHE**, o carregamento não ocorrerá.
- iii. Ocorre o **WRITE-AHEAD LOG**
- iv. Ocorre o **RECOVERY**

Laboratório

1. Crie um database chamado **CARROS** com quatro arquivos: um para os dados no **FILEGROUP** primário, outro arquivo de dados em um **FILEGROUP** chamado **FG_CARROS_1** e outros dois para o log. Os valores de **SIZE**, **MAXSIZE** e **FILEGROWTH** dos arquivos devem ser 6Mb, 12Mb e 2Mb respectivamente.

Resposta:

Capítulo 12

Integridade e Consistência - Parte 2

UDDT (User-Defined Datatypes)

O usuário pode definir seus próprios datatypes com base nos tipos de dados fornecidos pelo SQL Server. Conhecidos como **UDDTs**, o tipo de dado definido pelo usuário também pode ser considerado sinônimo de um tipo já disponível.

Para trabalharmos com os **UDDTs** devemos empregar o uso de duas **STORED PROCEDURES**.

- **SP_ADDTYPE**: responsável pela criação do tipo de dado. Sendo que contém os seguintes parâmetros:
 - **@TYPENAME**: define o nome do datatype.
 - **@PHYTYPE**: define o tipo de dado do **UDDT**.
 - **@NULLTYPE**: define a aceitação de valores nulos pelo **UDDT**.
- **SP_DROPTYPE**: responsável pela remoção do **UDDT**. O único parâmetro que devemos informar é **@TYPENAME**, para que o SQL Server identifique o tipo de dado a ser removido.

Observe a utilização desse método de **UDDT** no script abaixo:

```
EXEC SP_ADDTYPE 'MOEDA', 'DECIMAL(9,2)', 'NULL'
GO

CREATE TABLE BANCO
(
    COD_CONTA INT,
    NOME_CORRENTISTA VARCHAR(50),
    SALDO MOEDA
)
GO
```

Para descartar a nossa **UDDT**, basta fazer:

```
EXEC SP_DROPTYPE MOEDA
GO
```

Para que possamos descartá-la, é necessário que nenhuma tabela esteja usando esse **UDDT**. Caso contrário é impossível.

NULL / NOT NULL

Como havíamos visto rapidamente, no capítulo 7, é possível definir se a coluna de uma tabela aceita ou não valores nulos. Para tal, colocamos uma instrução, no momento da criação de uma tabela, ao lado do tipo de dado da coluna. Essa instrução pode ser **NULL** ou **NOT NULL**. Vejamos um exemplo:

```
CREATE TABLE TEMP
(
    CODIGO      INT      NOT NULL,
    VALOR       VARCHAR(20) NULL
)
GO
```

PRIMARY KEY

Para que uma linha seja identificada como única dentro de uma tabela, devemos garantir a integridade de entidades por meio da constraint **PRIMARY KEY**. Assim que essa constraint é definida no momento da criação ou da alteração de uma tabela, podemos definir também a chave primária.

É importante lembrar que cada tabela pode apresentar apenas uma constraint deste tipo. Uma vez definida como chave primária, uma coluna não deve aceitar valores nulos. Caso essa constraint seja atribuída para mais de uma coluna, a combinação de valores dessas duas colunas deverá ser única.

FOREIGN KEY / REFERENCES

Para garantir a integridade referencial entre duas tabelas, é preciso definir a constraint **FOREIGN KEY / REFERENCES** quando ambas são criadas ou alteradas. Uma consulta definida como **PRIMARY KEY** na tabela de origem poderá receber o atributo de chave estrangeira em outra coluna, de modo que seja criada uma ligação entre as duas.

Valores nulos são aceitos nas constraints **FOREIGN KEY / REFERENCES**, porém, sua presença faz com que ela interrompa a verificação dos demais valores existentes. Portanto devemos especificar o valor **NOT NULL** nas colunas definidas como **FOREIGN KEY / REFERENCES** para que a verificação seja completa.

DEFAULT

As colunas que fazem parte de um registro de dados devem conter valores independentes se eles são nulos ou não. Porém, nem sempre é recomendável trabalhar com colunas que aceitam esses valores. Nesse caso, precisamos definir valores padrões que preencheram essa coluna. Como vimos anteriormente, no capítulo 7, podemos adicionar um valor padrão no momento de criação de uma tabela, mas também é possível criar um objeto DEFAULT para ser utilizado nas tabelas. Vejamos:

```
CREATE DEFAULT DF_AUTOR AS 'DESCONHECIDO'  
GO
```

O procedimento acima cria um objeto chamado **AUTOR** como um objeto **DEFAULT**. Depois de criado, é necessário ligá-lo a uma coluna que o utilize. Para efeito de exemplo, temos uma tabela chamada **AUTORES** e nessa tabela existe a coluna **NOME_AUTOR** que não tem um valor padrão. Portanto, utilizaremos o objeto **DEFAULT DF_AUTOR**, que acabamos de criar, como **DEFAULT** dessa coluna. Para isso, fazemos uso da stored procedure **SP_BINDEFAULT**. Essa stored procedure recebe dois parâmetros:

- **@DEFNAME**: Nome dado ao DEFAULT criado.
- **@OBJNAME**: Coluna da tabela que receberá os valores padrões.

Vejamos a utilização da **SP_BINDEFAULT**:

```
EXEC SP_BINDEFAULT DF_AUTOR, 'AUTORES.NOME_AUTOR'  
GO
```

A diferença entre essa utilização e a utilização no momento da criação da tabela, é que o objeto criado por esse método fica disponível para o uso em qualquer outra tabela no database em que ele foi criado.

Caso queiramos remover o **DEFAULT** de uma coluna, usamos a stored procedure **SP_UNBINDEFAULT**, com o parâmetro **@OBJNAME**. Vejamos:

```
EXEC SP_UNBINDEFAULT 'AUTORES.NOME_AUTOR'  
GO
```

RULEs

Similares a constraint **CHECK**, os **RULEs** não fazem parte da estrutura da tabela. Primeiramente, é preciso criá-los para que depois estes objetos sejam ligados à coluna de uma tabela. O objeto **RULE** deve ser associado a uma coluna ou a um **UDDT**.

Para criar um objeto RULE, devemos proceder da seguinte forma:

```
CREATE RULE R_SIM_NAO
    AS @VAR IN ('S', 'N')
GO
```

Depois de criá-la, devemos associá-la a uma coluna de uma tabela. Vejamos:

```
EXEC SP_BINDRULE R_SIM_NAO, 'AUTORES.ATIVO'
GO
```

Caso seja necessário desligá-la da coluna, fazemos da seguinte forma:

```
EXEC SP_UNBINDRULE 'AUTORES.ATIVO'
GO
```

Ação Em Cadeia

A ação em cadeia abrange a utilização de dois comandos diferentes: **ON DELETE CASCADE** e **ON UPDATE CASCADE**. Para compreendê-los, utilizaremos o exemplo de duas tabelas: tabela **PAIS** e tabela **FILHOS**. Como já podemos imaginar, o tipo de relacionamento dessas duas tabelas será **1 : N**, ou seja, um-para-muitos, já que um pai pode ter vários filhos, mas o contrário não é possível. Vejamos:

```
CREATE TABLE PAIS
(
    COD_PAIS INT NOT NULL,
    NOME_PAIS VARCHAR(30) NOT NULL,
    CONSTRAINT PK_PAIS_COD_PAIS PRIMARY KEY (COD_PAIS)
)
GO

CREATE TABLE FILHOS
(
    COD_FILHO INT NOT NULL,
    COD_PAIS INT NOT NULL,
    NOME_FILHO VARCHAR(30) NOT NULL,
    CONSTRAINT PK_FILHOS_COD_FILHO PRIMARY KEY (COD_FILHO),
    CONSTRAINT FK_FILHOS_COD_PAIS FOREIGN KEY (COD_PAIS) REFERENCES
PAIS(COD_PAIS)
    ON DELETE CASCADE
)
GO

INSERT INTO PAIS VALUES (1, 'JOSE MACHADO')
INSERT INTO PAIS VALUES (2, 'CARLOS SERROTE')
INSERT INTO PAIS VALUES (3, 'ASTROBALDO MARTELLO')
GO

INSERT INTO FILHOS VALUES (1,1, 'JOSIAS MACHADO')
INSERT INTO FILHOS VALUES (2,1, 'CARLITA MACHADO')
INSERT INTO FILHOS VALUES (3,1, 'FERROLHO MACHADO')
INSERT INTO FILHOS VALUES (4,1, 'JOSE MACHADO JUNIOR')
INSERT INTO FILHOS VALUES (5,2, 'CARLITOS SERROTE')
```

```
INSERT INTO FILHOS VALUES (6,2,'CAROLINA SERROTE')
INSERT INTO FILHOS VALUES (7,3,'GUMERCINA MARTELLO')
GO
```

Como as tabelas **PAIS** e **FILHOS** estão relacionadas, caso façamos a exclusão de um registro na tabela **PAIS**, os registros da tabela **FILHOS** correspondentes também serão excluídos em cascata. Faça o teste: Exiba o nome do pai com os respectivos filhos, após, exclua o registro número '1' da tabela **PAIS** e exiba novamente todos os pais com os respectivos filhos e veja o que foi alterado.

Com o comando **ON UPDATE CASCADE** a idéia é a mesma, só que nesse caso, os registros serão atualizados em cascata.

Removendo Uma Constraint Com o Comando ALTER TABLE

Para remover uma constraint de uma tabela, utilizamos o comando **ALTER TABLE** da seguinte forma:

```
ALTER TABLE [nome_tabela] DROP CONSTRAINT [nome_constraint]
```

Devemos lembrar que caso seja necessário remover uma constraint do tipo **PRIMARY KEY**, ela não pode estar referenciada em nenhuma outra tabela como **FOREIGN KEY / REFERENCES**, ou seja, essa **PRIMARY KEY** não pode ser uma chave estrangeira para outra tabela. Para que possamos removê-la, então, é necessário primeiramente remover a **FOREIGN KEY / REFERENCES** da tabela à qual ela faz referência e depois removê-la.

Adicionando Uma Constraint Com o Comando ALTER TABLE

Para adicionar uma constraint a uma tabela, utilizamos o comando **ALTER TABLE** da seguinte forma:

```
ALTER TABLE [nome_tabela] ADD CONSTRAINT [nome_constraint] [tipo_constraint]
```

Adicionando Uma Coluna a Tabela

Para adicionarmos uma coluna a tabela já existente, fazemos da seguinte maneira:

```
ALTER TABLE [nome_tabela] ADD [nome_coluna] [tipo_dado_coluna] { [referencias] }
```

Removendo Uma Coluna da Tabela

É possível remover uma coluna de uma tabela, caso ela não esteja mais sendo usada ou pode ser descartada sem afetar o sistema. Façamos:

```
ALTER TABLE [nome_tabela] DROP COLUMN [nome_coluna]
```

Habilitando e/ou Desabilitando TRIGGERS de Uma Tabela

Para que um **TRIGGER** seja habilitado ou desabilitado de uma tabela, podemos proceder da seguinte forma:

```
ALTER TABLE [nome_tabela] { ENABLE | DISABLE } TRIGGER ALL
```

Habilitando e/ou Desabilitando Constraints

Caso os constraints **FOREIGN KEY / REFERENCES** e **CHECK** não estejam habilitados para uma determinada coluna, essa pode apresentar valores inconsistentes. É possível habilitar constraints para que dados posteriores não se tornem inconsistentes. Para tal:

- Habilitando: **ALTER TABLE [nome_tabela] CHECK CONSTRAINT [nome_constraint]**
- Desabilitando: **ALTER TABLE [nome_tabela] NOCHECK CONSTRAINT [nome_constraint]**

É possível a inclusão de constraints do tipo **FOREIGN KEY / REFERENCES** e **CHECK** em colunas que já apresentam dados inconsistentes. Para isso usamos o comando **ALTER TABLE** da seguinte forma:

Para a constraint **FOREIGN KEY / REFERENCES**:

```
ALTER TABLE [nome_tabela] WITH NOCHECK ADD CONSTRAINT [nome_constraint]
FOREIGN KEY [nome_coluna] REFERENCES [nome_tabela]([nome_coluna])
```

Para a constraint **CHECK**:

```
ALTER TABLE [nome_tabela] WITH NOCHECK ADD CONSTRAINT [nome_constraint]
CHECK ([nome_coluna][operador_comparacao][valor])
```

IDENTITY

IDENTITY é uma propriedade que permite a criação de números a partir de um número inteiro. Trata-se, portanto, de um contador automático capaz de gerar números ao longo de uma coluna.

Para que **IDENTITY** seja utilizado, devemos atribuir uma origem, também chamado de **SEED** ou semente, e um incremento ou **INCREMENT**. Qualquer número inteiro por ser definido para representar a semente ou o incremento. Vejamos um exemplo:

```
CREATE TABLE CARROS
(
    CODIGO      INT IDENTITY(1,1) NOT NULL,
    MARCA       VARCHAR(20),
    MODELO      VARCHAR(20),

    CONSTRAINT PK_CARROS_CODIGO PRIMARY KEY (CODIGO)
)
GO
```

Adicionando Uma Coluna IDENTITY Com ALTER TABLE

Podemos adicionar uma coluna **IDENTITY** a uma tabela já criada da seguinte forma:

ALTER TABLE [nome_tabela] ADD [nome_coluna] [datatype] IDENTITY

Apesar de este método poder inserir uma coluna **IDENTITY**, ele não modifica a propriedade de uma coluna em específico.

SET IDENTITY_INSERT

A propriedade **SET IDENTITY_INSERT** pode ser definida como **ON** apenas para uma tabela que possua uma coluna **IDENTITY** de um database. Caso essa propriedade já esteja configurada como **ON** em uma tabela e o comando **SET IDENTITY_INSERT ON** esteja relacionado à outra tabela uma mensagem de erro será retornada pelo SQL Server.

Sintaxe: **SET IDENTITY_INSERT [nome_tabela] { ON | OFF }**

Obtendo Informações da Coluna IDENTITY

O SQL Server oferece diversas opções para a verificação de informações relacionadas ao **IDENTITY** de uma tabela, dependendo das especificações de parâmetros listados a seguir:

Opções	Como Utilizá-las
DBCC CHECKIDENT	DBCC CHECKIDENT ('[nome_tabela]')
IDENTITYCOL	SELECT IDENTITYCOL FROM [nome_tabela]
IDENT_CURRENT	SELECT IDENT_CURRENT('[nome_tabela]')
IDENT_INCR	SELECT IDENT_INCR('[nome_tabela]')
IDENT_SEED	SELECT IDENT_SEED('[nome_tabela]')
@@IDENTITY	SELECT @@IDENTITY
SCOPE_IDENTITY	SELECT SCOPE_IDENTITY() AS [alias]

DBCC CHECKIDENT

O valor corrente do **IDENTITY** da tabela é corrigido caso necessário. A sintaxe completa de utilização é a seguinte: **DBCC CHECKIDENT ([nome_tabela], { NORESEED | RESEED }, [novo_valor_reseed]) [WITH NO_INFOMSGS]**

Caso o parâmetro **RESEED** seja informado, isso determinará que será preciso reinicializar o valor do **IDENTITY**. Já o parâmetro **NORESEED** informa que não será preciso reinicializar o valor do **IDENTITY**.

IDENTITYCOL

Exibi a coluna que possui o **IDENTITY**, bem como seus valores.

IDENT_CURRENT

Similar ao **@@IDENTITY** e ao **SCOPE_IDENTITY**, exibi o último valor gerado pelo **IDENTITY**.

IDENTITY_INCR

Exibi o valor do incremento da coluna identidade especificada.

IDENTITY_SEED

Exibi o valor da semente da coluna identidade especificada.

@@IDENTITY

Retorna o último valor gerado pelo **IDENTITY** vindo de qualquer tabela.

SCOPE_IDENTITY

Retorna o valor mais recente atribuído a uma coluna **IDENTITY**. Esta coluna pode pertencer a qualquer tabela que faça parte de um escopo de uma **PROCEDURE** ou qualquer **TRIGGER**.

Exercícios

1. Que é User Defined Datatype?

Resposta:

2. Que é **IDENTITY** e como devemos utilizá-la?

Resposta:

3. Que é constraint **FOREIGN KEY / REFERENCES** e qual a sua função?

Resposta:

4. Quando a constraint **UNIQUE** deve ser utilizada?

Resposta:

5. Quanto à integridade e consistência de dados é incorreto afirmar:

- a. **NULL** (o mesmo que nulo) determina que uma coluna não deverá receber valor algum, enquanto **NOT NULL** (não nulo) indica que uma coluna deverá sempre receber um valor no momento em que uma linha é acrescentada.
- b. A integridade de entidade é determinada pelas chaves estrangeiras, enquanto a integridade referencial é definida pelas chaves primárias.
- c. Para preservar tanto a integridade como a consistência dos dados no SQL Server, podemos utilizar objetos criados justamente para esta finalidade: são as chamadas constraints.
- d. Outros objetos, como **TRIGGERS, RULES** e default também podem ser utilizados, porém, as constraints podem ser definidas no momento em que a tabela é criada.
- e. Quando limitamos os valores de uma tabela estamos restringindo os dados que serão inseridos. Por exemplo, uma coluna com dados sobre Sexo só deve aceitar valores como F e M. Para que esta limitação seja definida, devemos implementar a integridade de domínio.

6. Qual das alternativas a seguir possui Built-in Datatypes baseados em caracteres?
 - a. char, nchar, bigint
 - b. varchar, char, bit
 - c. ntext, char, nvarchar
 - d. varchar, char, text
 - e. bit, int, text
7. Que alternativa apresenta as diferentes constraints que podem ser utilizadas no SQL Server para otimizar a integridade dos dados encontrados nas colunas de uma tabela?
 - a. PRIMARY KEY, FOREIGN KEY/REFERENCES, UNIQUE, CHECK E DEFAULT
 - b. IDENTITY, UNIQUE, CHECK, ON DELETE CASCADE, DEFAULT
 - c. PRIMARY KEY, IDENTITY, DEFAULT, FOREIGN KEY/REFERENCES, CHECK
 - d. ON DELETE CASCADE, PRIMARY KEY, FOREIGN KEY/REFERENCES, CHECK
 - e. CHECK, REFERENCES, IDENTITY, PRIMARY KEY, UNIQUE
8. Que alternativa contém o comando que exibe o valor mais recente que foi definido para uma coluna Identity pelo SQL Server?
 - a. IDENTITY_SEED
 - b. IDENTITYCOL
 - c. SCOPE_IDENTITY
 - d. SET IDENTITY_INSERT_ON
 - e. IDENT_INCR
9. Que alternativa apresenta as palavras que devem ser inseridas nos espaços em branco da afirmativa a seguir? Derivados dos system datatypes, os _____ são voltados especificamente para o banco de dados no qual esses estão sendo criados. Portanto, não é possível utilizá-los em outros bancos de dados. A exceção fica por conta dos _____ criados no database _____, pois os databases subsequentes também possuirão esses datatypes.
 - a. Rules / UDDTs / TempDB
 - b. Defaults / Rules / TempDB
 - c. User Definided Datatypes / Defautls / Master
 - d. User Definided Datatypes / UDDTs / Model
 - e. Rules / Defaults / Model

Laboratório

No database **MINHA_DB**, crie os seguintes **UDDTs**:

Nome UDDTs	datatypes	Nulabilidade
CODIGO	INT	NOT NULL
NOME	CHAR(100)	NOT NULL
MOEDA	DECIMAL(9,2)	NOT NULL
FOTO	IMAGE	NULL
OBSERVACAO	TEXT	NULL
DATA	SMALLDATETIME	NOT NULL

Resposta:

Capítulo 13

O Que é Uma Views?

Definimos uma **VIEW** como sendo uma tabela virtual composta por linhas e colunas de dados, os quais são provenientes de tabelas referenciadas em uma query que define a **VIEW**. Essas linhas e colunas são geradas de forma dinâmica no momento em que é feita uma referência a uma **VIEW**.

A query que determina a **VIEW** pode ser proveniente de uma ou mais tabelas ou de outras **VIEWs**. Para determinar uma **VIEW** que utiliza dados provenientes de fontes distintas, podemos utilizar as queries distribuídas.

Ao criarmos uma **VIEW**, podemos filtrar o conteúdo de uma tabela a ser exibido, visto que ela é um filtro de tabelas que pode auxiliar no agrupamento de dados das tabelas, protegendo certas colunas e simplificando o código de uma programação.

Mesmo que o SQL Server seja desligado, depois de criada, a **VIEW** não deixa de existir no sistema. Embora sejam internamente compostas por **SELECTs**, as **VIEWs** não ocupam espaço no banco de dados.

Criando Uma VIEW

O código para criar uma **VIEW** é o seguinte: **CREATE VIEW [nome_view] AS [comando]**

Para que possamos visualizar um exemplo funcional, utilizando o database **MINHA_DB**, crie duas tabelas, **FUNCIONARIOS** e **CARGOS**, com os seguintes dados:

Tabela: **FUNCIONARIOS**

CODIGO (INT IDENTITY)	NOME (VARCHAR(30))	COD_CARGO (INT)
1	CORNELIO MARISTOS	1
2	CLEIDI CORONHA	1
3	PAULO FERNANDO SERRA	2
4	CASIMIRO CASADO	3
5	FERDINANDO FERNADEZ	4

Tabela: CARGOS

COD_CARGO (INT IDENTITY)	NOME_CARGO (VARCHAR(15))
1	FAXINEIRO
2	ESTAGIARIO
3	GERENTE
4	PRESIDENTE

Iremos criar uma **VIEW** que nos retornará o nome do funcionário e o cargo ocupado por este. Criando nossa **VIEW**, essa fica da seguinte forma:

```
CREATE VIEW V_NOME_FUNC_CARGO AS
    SELECT F.NOME, C.NOME_CARGO FROM FUNCIONARIOS AS F
        INNER JOIN CARGOS AS C ON F.COD_CARGO = C.COD_CARGO
GO
```

Depois de criada, para executá-la procedemos da mesma forma que procederíamos caso existisse uma tabela com o nome do funcionário e o cargo ocupado por este. Vejamos:

```
SELECT * FROM V_NOME_FUNC_CARGO
GO
```

Crie e execute esta **VIEW** e observe o resultado.

Vantagem das VIEWS

- Reutilização: As VIEWS são objetos de caráter permanente, portanto, elas podem ser lidas por vários usuários de forma simultânea.
- Segurança: As VIEWS permitem ocultar determinadas colunas de uma tabela. Para tanto, basta criar uma VIEW e disponibilizar apenas ela para o usuário.
- Simplificação de código: As VIEWS permitem criar um código de programação muito mais limpo na medida em que podem conter um SELECT complexo. Dessa forma, podemos criar essas VIEWS para programadores a fim de poupar-lhos do trabalho de escrever SELECTs complexos.

WITH ENCRYPTION

A cláusula **WITH ENCRYPTION** permite realizar a criptografia das entradas da tabela **SYSCOMMENTS** (ver Apêndice) que contenham o texto do comando **CREATE VIEW**. Ao utilizarmos esta cláusula, a VIEW não pode ser publicada como parte da replicação do SQL Server.

Quando uma **VIEW**, **STORED PROCEDURE** ou **TRIGGER** é criptografado, o usuário deixa de ter acesso ao código que as gerou. A sintaxe a seguir, mostra a utilização da cláusula **WITH ENCRYPTION**, vejamos:

CREATE VIEW [nome_view] WITH ENCRYPTION AS [comando]

No caso do nosso exemplo anterior, esse ficaria desta forma:

```
CREATE VIEW V_NOME_FUNC_CARGO WITH ENCRYPTION AS
    SELECT F.NOME, C.NOME_CARGO FROM FUNCIONARIOS AS F
        INNER JOIN CARGOS AS C ON F.COD_CARGO = C.COD_CARGO
GO
```

WITH CHECK OPTION

Esta cláusula faz com que os critérios definidos em um **select_statement** sejam seguidos no momento em que são executados os comandos que realizam a alteração de dados com relação às **VIEWS**. Dessa forma, a cláusula **WITH CHECK OPTION** assegura que os dados continuem visíveis por meio da **VIEW** mesmo após uma linha ter sido alterada. Para exemplificar, criaremos uma **VIEW** que nos retorne somente os produtos, da tabela **PRODUTOS** e database **MINHA_DB**, que tenham valor igual ou inferior a 400,00:

```
CREATE VIEW V_PRODUTOS_VALOR400 AS
    SELECT * FROM PRODUTOS WHERE VALOR <= 400
GO
```

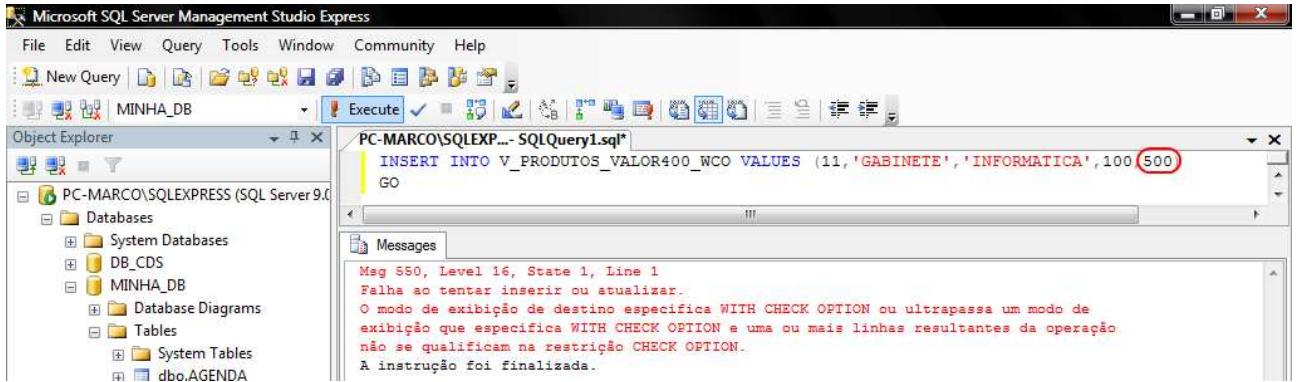
Caso o usuário tente inserir, por meio da **VIEW** um produto com o valor superior a 400,00, o SQL Server irá permitir, pois não há nenhum tipo de checagem com relação ao valor do produto. Mas caso usássemos a cláusula **WITH CHECK OPTION**, isso impediria o usuário de, por meio da **VIEW**, incluir um produto com o valor superior a 400,00. Vejamos:

```
CREATE VIEW V_PRODUTOS_VALOR400_WCO AS
    SELECT * FROM PRODUTOS WHERE VALOR <= 400
WITH CHECK OPTION
GO
```

Agora tentaremos incluir um produto com o valor acima de 400,00 por meio da **VIEW**. Vejamos:

```
INSERT INTO V_PRODUTOS_VALOR400_WCO VALUES (11, 'GABINETE', 'INFORMATICA', 100, 500)
GO
```

Observe a mensagem gerada pelo SQL Server:



The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'MINHA_DB' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXP... - SQLQuery1.sql*' contains the following code:

```
INSERT INTO V_PRODUTOS_VALOR400_WCO VALUES (11, 'GABINETE', 'INFORMATICA', 100, 500)
GO
```

In the 'Messages' pane, an error message is displayed:

```
Msg 550, Level 16, State 1, Line 1
Falta ao tentar inserir ou atualizar.
O modo de exibição de destino especifica WITH CHECK OPTION ou ultrapassa um modo de
exibição que especifica WITH CHECK OPTION e uma ou mais linhas resultantes da operação
não se qualificam na restrição CHECK OPTION.
A instrução foi finalizada.
```

Excluindo Uma VIEW

Para excluirmos uma **VIEW**, basta utilizar o comando **DROP VIEW [nome_view]**.

Alterando Uma VIEW

Caso seja necessário alterar a estrutura de uma **VIEW**, devemos utilizar o comando **ALTER VIEW**, da seguinte forma:

ALTER VIEW [{ WITH ENCRYPTION }] [nome_view] AS [comando] [{ WITH CHECK OPTION }]

VIEWS Indexadas

As **VIEWS** indexadas otimizam o desempenho. Os índices criados sobre **VIEWS** ocupam um certo espaço em disco, visto que eles têm a finalidade de armazenar os dados no banco de dados.

Diretrizes Para Criação de Índices Sobre Uma VIEW

A criação de índices sobre uma **VIEW** é uma tarefa que requer a observação de algumas regras. Vejamos:

- A **VIEW** dever ser criada com a cláusula **WITH SCHEMABINDING**.
- A **VIEW** sobre a qual o índice será criado não poderá fazer referência a outra **VIEW**.

- O primeiro índice a ser criado sobre uma VIEW deve ser do tipo CLUSTERED e UNIQUE.
- Os nomes de tabelas e funções que se encontram dentro de uma VIEW devem ser formados por duas partes.

Criando e Indexando VIEWS

Observemos o código a seguir, que demonstra como criar uma VIEW:

```
CREATE VIEW V_FUNC WITH SCHEMABINDING AS
    SELECT CODIGO, NOME, COD_CARGO FROM DBO.FUNCIONARIOS
GO
```

Já este outro demonstra como indexá-la:

```
CREATE UNIQUE CLUSTERED INDEX IDX_FUNC1 ON V_FUNC(CODIGO)
CREATE NONCLUSTERED INDEX IDX_FUNC2 ON V_FUNC(NOME)
CREATE NONCLUSTERED INDEX IDX_FUNC3 ON V_FUNC(COD_CARGO)
GO
```

Exercícios

1. O que é uma **VIEW**?

Resposta:

2. Quais são as vantagens das **VIEWS**?

Resposta:

3. Quais são as diretrizes para a criação de índices sobre uma **VIEW**?

Resposta:

4. Qual comando podemos utilizar para alterar uma **VIEW** sem excluir as permissões já atribuídas a ela?

Resposta:

Capítulo 14

Programando Com Transact-SQL

A programação por meio da linguagem Transact-SQL envolve a criação de variáveis, a utilização de elementos de controle de fluxo, como os comandos **IF** e **WHILE**, e a utilização de **STORED PROCEDUREs**.

Variáveis de Usuário

Uma variável local do Transact-SQL é um objeto nos scripts e batches que mantém um valor de dado. Por meio do comando **DECLARE**, podemos criar variáveis locais, sendo isto feito no corpo de um procedimento ou batch.

Vejamos alguns exemplos de declaração de variáveis:

DECLARE @VAR1 INT

DECLARE @VAR2 CHAR(10)

ou

DECLARE @VAR1 INT, @VAR2 CHAR(10)

Como podemos observar, todo nome de variável é precedido do caractere '@' (arroba). A sintaxe básica de declaração é a seguinte:

DECLARE @[nome_variavel] [datatype]

Atribuindo Valor a Uma Variável

Para atribuirmos um valor a uma variável criada, utilizamos a seguinte sintaxe:

SET @[nome_variavel] = [valor]

Existe a possibilidade de atribuir um valor a uma variável por meio da instrução **SELECT**. Para isso procedemos da seguinte maneira:

SELECT @[nome_variavel] = [nome_coluna] FROM [nome_tabela] WHERE [nome_campo] [operador_comparacao] [valor]

Controle de Fluxo

O SQL Server trabalha com elementos de linguagem denominados "elementos de controle de fluxo", cuja função é possibilitar a criação de programas de grande utilidade para o sistema.

Os elementos de controle de fluxo são:

BEGIN / END: Têm a finalidade de iniciar e finalizar, respectivamente, um bloco de comandos.

IF / ELSE: Estrutura de decisão. A sintaxe básica é:

```
IF [expressao_booleana]
{ comando_sql | bloco_comando }
[ ELSE
  { comando_sql | bloco_comando } ]
```

Caso o bloco de comando tenha mais de duas linhas, é necessário o uso de **BEGIN / END**. Similar a linguagem **PASCAL** e **DELPHI**, com exceção da não utilização do ';' ou '.' no final do **END**.

CASE / WHEN / THEN / END: O elemento CASE é utilizado para conferir uma lista de condições e, então, retornar uma entre as várias expressões de resultados possíveis. É especialmente útil quando queremos evitar **IFs** aninhados. Sintaxe básica:

SELECT

```
CASE [nome_coluna]
  WHEN [valor1] THEN [novo_valor1]
  ...
  WHEN [valorN] THEN [novo_valorN]
  ELSE [novo_valor]
END
```

FROM [nome_tabela]

Um exemplo prático é o caso em que desejamos exibir o sexo, por extenso, dos funcionários da nossa loja de CDs. Usando o database **DB_CDS**, procederemos da seguinte forma:

```

SELECT COD_FUNC,
       NOME_FUNC,
       END_FUNC,
       SAL_FUNC,
       'SEXO_FUNC' = CASE SEXO_FUNC
                         WHEN 'F' THEN 'FEMININO'
                         WHEN 'M' THEN 'MASCULINO'
                     END
FROM FUNCIONARIOS
GO

```

Execute esse comando e observe o resultado.

WHILE: Este comando faz com que o comando SQL ou bloco de comando seja executado repetidamente ou até o momento em quem uma condição é verdadeira. Vejamos a sintaxe:

WHILE [condicao_booleana]

BEGIN

[comando1]

...

[comandoN]

END

Nota: Por meio dos comandos **BREAK** e **CONTINUE**, é possível controlar, de dentro da estrutura de loop, a execução do comando **WHILE**.

TRY ... CATCH: Esses comandos são utilizados para tratar e controlar erros em grupos de comandos do SQL Server. Vejamos a sintaxe:

BEGIN TRY

{ comando_sql | bloco_comando }

END TRY

BEGIN CATCH

{ comando_sql | bloco_comando }

END CATCH [;]

A seguir, destacamos algumas considerações relacionadas aos comandos **TRY** e **CATCH**:

- Qualquer erro de execução com o valor de severidade superior a 10 e que não finaliza a conexão com o banco de dados é capturada por **TRY / CATCH**.
- Qualquer comando entre os comandos **END TRY** e **BEGIN CATCH** irá provocar um erro de sintaxe. Portanto é imprescindível que o bloco **TRY** seja imediatamente seguido por um bloco **CATCH** associado.
- Uma das limitações dos comandos **TRY / CATCH** é não poder transpor muitos blocos de comandos do Transact-SQL e diversos batches.
- A aplicação não terá os erros identificados por um bloco **CATCH**. Porém, por meio de alguns mecanismos utilizados no bloco **CATCH**, é possível retornar essas informações de erro à aplicação. Dentre esses mecanismos, temos os comandos **RAISERROR** e **PRINT**, e conjuntos de resultados **SELECT**.
- O comando que vem logo após o comando **END CATCH** irá assumir o controle assim que o código do bloco **CATCH** tiver sido finalizado.
- Blocos **TRY / CATCH** podem conter comandos **TRY / CATCH** aninhados.
- Em um bloco **CATCH** que possui um instrução **TRY / CATCH** aninhada, caso haja erro no bloco **TRY** aninhado, ele irá passar o controle para o bloco **CATCH** aninhado. O erro será retornado à aplicação que fez a chamada caso o bloco **CATCH** não possua uma construção **TRY / CATCH** aninhada.
- **TRIGGERS** e **STORED PROCEDURES** podem alternativamente ter construções **TRY / CATCH** próprias, cuja função é controlar erros criados pelos **TRIGGERS** e **STORED PROCEDURES**.
- **STORED PROCEDUREs** ou **TRIGGERS** executados pelo código de um bloco **TRY** podem conter erros não controlados. Estes podem ser identificados por construções **TRY / CATCH**.
- Caso a **STORED PROCEDURE** não tenha uma construção **TRY / CATCH** própria, o erro dessa retornará o controle para o bloco **CATCH** ligado ao bloco **TRY** que possui o controle **EXECUTE**. Se tiver o controle transferido pelo erro do bloco **CATCH** na **STORED PROCEDURE**. Então, o controle é retornado ao controle localizado logo após o comando **EXECUTE** responsável por chamar a **STORED PROCEDURE**, assim que o código do bloco **CATCH** tiver sido executado.
- O SQL Server possui comandos chamados **GOTO**, que podem ser utilizados para ir de uma etiqueta a outra no mesmo bloco **TRY / CATCH**. Os comandos **GOTO** também servem para sair desses mesmos blocos.
- Em uma função definida pelo usuário, não é possível utilizar a construção **TRY / CATCH**.

A fim de obter informações sobre o erro que provocou a execução do bloco **CATCH**, dispomos de várias funções de sistema que devem ser utilizadas em qualquer parte do escopo de um bloco **CATCH**. Caso não sejam utilizadas nesse escopo, as funções retornam **NULL**.

A seguir, descrevemos as funções de sistema disponíveis para obter informações de erro a partir do bloco **CATCH**:

Função de Sistema	Descrição
ERROR_NUMBER()	Retorna o número do erro.
ERROR_SEVERITY()	Retorna a severidade do erro.
ERROR_STATE()	Retorna o estado do erro.
ERROR_PROCEDURE()	Retorna o nome da PROCEDURE que gerou o erro.
ERROR_LINE()	Retorna a linha em que ocorreu o erro.
ERROR_MESSAGE()	Retorna uma mensagem descritiva sobre o erro ocorrido. Pode incluir outros atributos como tempo, nome de objetos, extensões.

Porém, existem certos tipos de informações de erro que não são identificados por uma construção **TRY / CATCH**, tipos estes descritos abaixo:

- Aviso de atenção, dentre os quais são incluídos aqueles referentes a conexão de cliente interrompidas e pedidos de interrupção de cliente.
- Mensagens informativas ou avisos que apresentam um valor de severidade igual ou inferior a 10.
- Sessões finalizadas por meio do comando **KILL**.
- Erros que finalizam o processamento de tarefas do SQL Server Database Engine para a sessão e que apresentam um valor de severidade igual ou superior a 20. Caso a conexão com o banco de dados não seja afetada, enquanto um erro com severidade igual ou superior a 20 ocorrer, esse erro será controlado pela construção **TRY / CATCH**.

EXECUTE

A seguir, temos duas sintaxes da utilização do comando **EXECUTE**:

EXECUTE [@retorno =] {nome_procedure}

ou

EXECUTE ({ @variavel_string | [N]'tsql_string' } [+....n])

Na primeira sintaxe, temos **@retorno** e **nome_procedure**. **@retorno** representa uma variável que armazena o estado de uma **STORED PROCEDURE**, e deve ser declarada antes de ser utilizada em **EXECUTE**. **@retorno** é inteira (**INT**) e opcional.

Já **nome_procedure** representa o nome da **STORED PROCEDURE** a ser executada, nome este que pode ser parcial ou totalmente qualificado.

Um aspecto importante em relação à nomenclatura das **STORED PROCEDURES** é que os nomes devem estar de acordo com as regras definidas pelos identificadores, como **EXEC @retorno = SP_TESTE**.

Já na segunda sintaxe exibida, temos **@variavel_string** e **[N]'tsql_string'**. **@variavel_string** representa o nome de uma variável local. Esta variável por ser do tipo: **VARCHAR, CHAR, NCHAR, NVARCHAR**. Já **[N]'tsql_string'** representa uma string constante e por ser do seguinte tipo: **VARCHAR, NVARCHAR** (este será o tipo caso **[N]** seja utilizado.)

STORED PROCEDURE

Uma coleção de comando SQL criada para utilização permanente ou temporária em uma sessão de usuário ou por todos os usuários é chamada de **STORED PROCEDURE**. Podemos programar a execução de **STORED PROCEDURES** das seguintes formas:

- Execução no momento em que o SQL Server é inicializado.
- Execução em períodos específicos do dia.
- Execução em um horário específico do dia.

Aplicações criadas por meio do SQL Server podem ser armazenadas como **STORED PROCEDURES**. Assim, podemos criar aplicações com a finalidade de executar essas **STORED PROCEDURES** e fazer o processamento dos resultados.

Tipos de STORED PROCEDURES

As **STORED PROCEDURES** do SQL Server podem ser de tipos diferentes: System Stored Procedures, Stored Procedures Locais, Stored Procedures Remotas, Stored Procedures Temporárias Locais e Globais, Extended Stored Procedure. A seguir, estudaremos cada um dos tipos citados.

System Stored Procedure

As **STORED PROCEDUREs** criadas quando o SQL Server é instalado nos bancos de dados do sistema são denominadas System Stored Procedures. Quando o valor retornado por uma delas é igual a 0 (zero) significa que a operação por ela realizada foi completada com sucesso. Caso contrário, houve falha.

Características:

- Podem ser executadas de qualquer banco de dados.
- Permite ao administrador do sistema executar tarefas administrativas. Isso é possível mesmo que o administrador não tenha direito de acesso às tabelas internas das **STORED PROCEDUREs**.
- São iniciadas pelo prefixo **SP**.

São classificadas em diversos tipos:

- **Active Directory Stored Procedures:** A fim de registrar bancos de dados do SQL Server e instâncias do SQL Server no Active Directory do Windows Server, utilizamos esta categoria de System Stored Procedures.
- **Catalog Stored Procedures:** Aplicar funções de dicionário de dados ODBC. Separar aplicações ODBC de alterações para tabelas de sistema básicas.
- **Cursor Stored Procedures:** Acrescenta funcionalidades de variável de cursor.
- **Database Engine Stored Procedures:** Para a tarefa de manutenção geral do SQL Server Database Engine.
- **Database Mail And SQL Mail Stored Procedures:** Operações de envio e recebimento de e-mails a partir de uma instância do SQL Server.
- **Database Maintenance Plan Stored Procedures:** Tarefas de manutenção central exigidas para gerenciar o desempenho do banco de dados.
- **Distributed Queries Stored Procedures:** Aplicar e gerenciar Distributed Queries.
- **Full-Text Search Stored Procedures:** Aplicar e consultar índices FULL-TEXT.
- **Log Shipping Stored Procedures:** Ajustar, monitorar e alterar as configurações de log shipping.
- **Automation Stored Procedures:** É possível utilizar objetos Automation padrão em um batch padrão do Transact-SQL.
- **Notification Services Stored Procedures:** Para gerenciar os Notification Services do SQL Server.
- **Replication Stored Procedures:** Gerenciamento de replicação.
- **Security Stored Procedures:** Gerenciamento de segurança.
- **SQL Server Profiler Stored Procedures:** Desempenho e da atividade.
- **SQL Server Agent Stored Procedures:** Controlar atividades agendadas ou orientadas por eventos.
- **Web Task Stored Procedures:** Criação de páginas Web.
- **XML Stored Procedures:** Gerenciamento de textos XML.

- **General Extended Stored Procedures:** A partir de uma instância do SQL Server, oferece uma interface para programas de diversas atividades de manutenção.

Extended Stored Procedures

Estas **STORED PROCEDUREs** compreendem **DLLs** que podem ser carregadas e executadas dinamicamente por uma instância do SQL Server. São executadas externamente ao SQL Server como uma extensão Extended Stored Procedure, que está ao alcance do SQL Server mas fora do seu próprio ambiente.

As extended stored procedures apresentam as seguintes características:

- Podem ser executadas similarmente as STORED PROCEDUREs comuns.
- Permitem a criação de rotinas externas próprias em linguagens de programação. Uma dessas linguagens é o C.
- Podem ser adicionadas apenas ao banco de dados MASTER.
- São executadas diretamente no espaço de endereço de uma instância do SQL Server.
- Membros do fixed server role **sysadmin** poderão utilizar a instância do SQL Server para registrar a extended stored procedure, depois que ela é escrita, e dar a permissão de execução da procedure para outros usuários.
- São programadas por meio da API do **SQL Server Extended Stored Procedures**.
- Não devem ser utilizadas para instanciar o tempo de execução de linguagem comum do MS .Net Framework e para a execução do código gerado, já que versões futuras do SQL Server não suportarão mais este tipo de tarefa.
- Uma das desvantagens das extended stored procedures é que elas podem prejudicar a segurança e performance do servidor, como provocar a perda de memória. Por causa desse aspecto, é uma recomendação manter extended stored procedures e os dados referenciados em instâncias diferentes do SQL Server. Outra medida que pode ajudar no desempenho do servidor é fazer o acesso ao banco de dados com consultas distribuídas.

Exemplo de Extended Stored Procedure

```
EXEC MASTER..XP_CMDSHELL 'DIR *.*'
GO
```

O código acima executa o comando **DIR** por meio do SQL Server.

Stored Procedures Temporárias

As stored procedures temporárias têm como característica apresentar o mesmo tempo de duração que a conexão da sessão que criou as procedures, ou seja, se a conexão for encerrada, a stored procedure temporária também será.

Podem ser locais ou globais. A forma dessa identificação é feita por meio do caractere '#' (sharp) precedendo o nome da procedure.

- Stored Procedure Temporárias Locais: Precedida por #. Exemplo: **EXEC #*[nome_procedure]**
- Stored Procedure Temporárias Globais: Precedida por ##. Exemplo: **EXEC ##*[nome_procedure]**

Stored Procedures Remotas

As **STORED PROCEDUREs** chamadas a partir de um servidor remoto ou de um cliente conectado a um servidor distinto são chamadas de stored procedures remotas. A execução de **STORED PROCEDUREs** a partir de servidores locais ou remotos pode ser feita por meio da opção **REMOTE ACCESS**. A seguir, destacamos quais os ajustes que **REMOTE ACCESS** pode receber e seus respectivos resultados.

Valor 1: Ao ajustarmos **REMOTE ACCESS** para este valor, será permitido executar **STORED PROCEDUREs** locais a partir de servidores remotos e vice-versa.

Valor 0: Não será possível executar **STORED PROCEDUREs** locais em servidores remotos e vice-versa.

Nota: Nas versões posteriores do SQL Server essa opção não será mais possível. É aconselhável não utilizá-lo para desenvolver novos projetos.

Stored Procedures Locais

As stored procedures locais são criadas em bancos de dados individuais de usuários. Um exemplo de criação de uma stored procedure local é o seguinte:

```
CREATE PROCEDURE SP_SOMA AS
BEGIN
    DECLARE @SOMA INT

    SET @SOMA = 10 + 50

    RETURN
END
GO
```

Para executá-la:

```
EXEC SP_SOMA
GO
```

Para que possamos visualizar essa **STORED PROCEDURE**, expanda o nó **MINHA_DB -> PROGRAMMABILITY -> STORED PROCEDURES**.

Parâmetros

As **STORED PROCEDUREs** criadas no SQL Server podem receber parâmetros tanto de entrada como de saída. A fim de indicar um sucesso ou falha, bem como o motivo da falha, as stored procedures do SQL Server retornam um valor de status para a aplicação (batch ou procedure).

As **STORED PROCEDUREs** não retornam valores no lugar de seus nomes. Além disso, não podem ser utilizadas no lugar de expressões. Isso difere as **STORED PROCEDUREs** das **FUNCTIONs**. A seguir, temos um exemplo de **STORED PROCEDURE** que recebe dois valores e os soma:

```
CREATE PROCEDURE SP_SOMA2
( @A INT, @B INT ) AS
BEGIN
    DECLARE @SOMA INT

    SET @SOMA = @A + @B

    RETURN
END
GO
```

Para executá-la:

```
EXEC SP_SOMA2 10, 50
GO
```

Passagem de Parâmetro Por Referência

Um tipo de passagem de parâmetro que pode ser utilizado é a passagem por referência. Nesse tipo de passagem é utilizada a palavra chave **OUTPUT** a fim de retornar um parâmetro de saída.

Na passagem de parâmetro por referência, tanto o comando **EXECUTE** como o comando **CREATE PROCEDURE** utilizam **OUTPUT**.

Vejamos um exemplo:

```
CREATE PROCEDURE SP_MULTIPLICA
    ( @A INT, @B INT, @RESULT INT OUTPUT ) AS
BEGIN
    SET @RESULT = @A * @B
    RETURN
END
GO
```

Para que possamos receber a saída dessa procedure, devemos proceder da seguinte forma:

```
DECLARE @X INT
EXEC SP_MULTIPLICA 20, 20, @X OUTPUT
SELECT @X
```

Retornando Um Valor

Por meio do comando **RETURN**, é possível fazer com que a procedure retorne um valor, que deve ser obrigatoriamente um valor inteiro. Vejamos:

```
CREATE PROCEDURE SP_DIVIDE
    ( @A INT, @B INT ) AS
BEGIN
    DECLARE @C INT
    SET @C = @A / @B
    RETURN @C
END
GO
```

Para que possamos receber o retorno dessa procedure, temos o código dessa forma:

```
DECLARE @X INT
EXEC @X = SP_DIVIDE 100, 10
PRINT @X
```

A Criação de Uma STORED PROCEDURE

Na criação de uma **STORED PROCEDURE**, temos o seguinte processo:

1. Os comandos de criação são analisados para correção de erros de sintaxe.
2. Caso não existam erros de sintaxe, o SQL Server realiza as seguintes ações:
 - Armazena o nome da **STORED PROCEDURE** em uma tabela de sistema chamada **SYSOBJECTS**.
 - Armazena o texto da criação (comandos) da procedure em uma tabela chamada **SYSCOMMENTS** do banco de dados atual.

Assim que uma **STORED PROCEDURE** é criada, podemos referenciá-la a objetos ainda inexistentes no banco de dados. Neste caso, os objetos deverão existir apenas quando a procedure for executada. Essa referência antecipada a objetos é possível pelo processo chamado **delayed name resolution**.

Execução de Uma Procedure

Diante da recompilação ou primeira execução da **STORED PROCEDURE**, um procedimento adotado pelo processador de consultas é recorrer à stored procedure de processos **resolution** para fazer a leitura da mesma.

Existem algumas situações em que as **STORED PROCEDURES** são recompiladas de maneira automática. Uma delas é quando há alteração das estatísticas para um índice ou tabela que a procedure referenciou.

Outros motivos que ocasionam a recompilação automática das **STORED PROCEDURES** são a alteração da versão do esquema e quando há diferença entre o ambiente que a procedure é executada e o ambiente em que ela foi compilada.

Otimização da Procedure

Uma procedure sofre um processo de otimização caso ela tenha atingido o estágio **resolution** de maneira bem sucedida. O SQL Server possui um otimizador de queries, que desenvolve para a procedure um plano de execução que possui métodos de acesso rápido a dados. Para que o otimizador possa realizar essa tarefa, ele analisa os comando internos da procedure. Os índices, a quantidade de dados que as tabelas apresentam, os **JOINS**, **UNIONS**, **GROUP BYs** e **ORDER BYs**, e os valores e operadores utilizados na cláusula **WHERE** são os fatores que o processador de queries considera para criar o plano de otimização para a procedure.

Compilação da Procedure

Os processos referentes à analise e ao desenvolvimento do plano de query da **STORED PROCEDURE** compreendem a sua compilação. A procedure será executada assim que o otimizador de queries inserir o plano já compilado no cache da procedure.

As Próximas Execuções da Procedure

A partir do momento em que o plano de query otimizado tiver sido inserido na procedure cache, o SQL Server passará a utilizá-lo para realizar as próximas execuções da **STORED PROCEDURE**. Por esta razão, as próximas execuções da procedure serão mais rápidas do que a primeira.

Vantagem das Procedures

As principais vantagens de se utilizar **STORED PROCEDURES** do que comandos SQL armazenados no **Client** são:

- Execução rápida: A execução das **STORED PROCEDURES** são mais rápidas do que comandos SQL **Client** porque elas já tiveram sua sintaxe previamente verificada e foram otimizadas durante sua criação.
- Tráfego na rede: São capazes de reduzir a quantidade de dados que trafegam pela rede.
- Segurança: Podem ser aproveitadas como um mecanismo de segurança.
- Programação modular: Podem ser chamadas a partir de qualquer aplicação.

Considerações na Criação de STORED PROCEDURES

É importante considerar:

- Os seguintes comandos não podem ser utilizados dentro de **STORED PROCEDURES**: **CREATE PROCEDURE**, **CREATE DEFAULT**, **CREATE RULE**, **CREATE TRIGGER** e **CREATE VIEW**.
- Outras **STORED PROCEDURES**, tabelas temporárias, tabelas permanentes e **VIEWS** podem ser referenciadas por **STORED PROCEDURES**.
- Uma tabela temporária criada por uma **STORED PROCEDURE** existirá enquanto a procedure estiver sendo executada.

Excluindo Uma Procedure

Para excluir uma procedure, utilizamos o comando **DROP PROCEDURE [nome_procedure]**.

Alterando Uma STORED PROCEDURE

Podemos alterar a estrutura de uma **STORED PROCEDURE** por meio do comando **ALTER PROCEDURE**, da seguinte forma:

```
ALTER PROCEDURE [nome_procedure] { [parametro1] [datatype1], ..., [parametroN]
[datatypeN] } AS
```

```
BEGIN
```

```
{ comando_sql | bloco_comando }
```

```
RETURN
```

```
END
```

Recompilando Uma STORED PROCEDURE

Podemos recompilar uma procedure utilizando as seguintes opções:

- **WITH RECOMPILE**: Esta opção força a **STORED PROCEDURE** a se recompilar todas as vezes que for executada.
- **WITH ENCRYPTION**: Essa função criptografa o código da procedure.
- **WITH RECOMPILE, ENCRYPTION**: Força a recompilação todas as vezes que é executada e criptografa.
- **SP_RECOMPILE**: Essa **STORED PROCEDURE** recompila a procedure que indicarmos. Exemplo: **EXEC SP_RECOMPILE '[nome_procedure]'**

Observações Quando ao Uso de STORED PROCEDURES

A utilização de **STORED PROCEDURES** merece algumas observações, descritas a seguir:

- O nível atual de aninhamento é inserido em **@@nestlevel**, que é uma variável de sistema.

- Os níveis de aninhamento suportados pelas **STORED PROCEDUREs** são no máximo 32.
- Uma **STORED PROCEDURE** pode chamar outra **STORED PROCEDURE**. Quando isso ocorre, a segunda procedure tem direito a acessar todos os objetos criados na primeira **STORED PROCEDURE**. As tabelas temporárias estão entre esses objetos.

Exercícios

1. Que é uma variável local e como podemos criá-la?

Resposta:

2. Qual a função dos elementos de controle de fluxo? Dê exemplos.

Resposta:

3. Que é uma STORED PROCEDURE? Quais são os tipos de SP existentes?

Resposta:

4. Quais são as vantagens das procedures?

Resposta:

5. Cite alguns aspectos que devemos levar em consideração na criação de uma STORED PROCEDURE.

Resposta:

6. A qual tipo de STORED PROCEDURE o parágrafo a seguir se refere? **São criadas quando o SQL Server é instalado nos bancos de dados de sistema. Quando retornam valor 0 (ZERO), isto significa que a operação foi realizada com sucesso. Valores diferentes de 0 (ZERO) indicam falha.**
- Stored Procedures Remotas.
 - Stored Procedures Locais.
 - Extended Stored Procedures.
 - System Stored Procedures.
 - Stored Procedures Temporárias Locais e Globais.
7. Qual das alternativas a seguir NÃO é uma característica das Extend Stored Procedures
- Podem ser adicionadas apenas ao banco de dados master.
 - Não permitem a criação de rotinas externas próprias em linguagens de programação. Uma dessas linguagens é a C.
 - São programadas por meio da API do SQL Server Extended Stored Procedure.
 - São executadas diretamente no espaço de endereço de uma instância do SQL Server.
 - Não devem ser utilizadas para instanciar o tempo de execução de linguagem comum do Microsoft .NET FrameWork e para execução do código gerenciado, já que versões futuras do SQL Server não suportarão mais esse tipo de tarefa.
8. Qual das alternativas as seguir apresenta a descrição correta de um valor que pode ser atribuído à opção Remote Access?
- Valor 0: Ao ajustarmos Remote Access para esse valor, será permitido executar STORED PROCEDUREs locais a partir de servidores remotos ou STORED PROCEDUREs remotas a partir do servidor local.
 - Valor 1: Ao ajustarmos Remote Access para este valor, STORED PROCEDUREs locais não serão executadas a partir de um servidor remoto e STORED PROCEDUREs remotas não serão executadas no servidor local.
 - Valor 1: Ao ajustarmos Remote Access para este valor, não será permitido executar STORED PROCEDUREs locais a partir de servidores remotos mas poderão ser executadas STORED PROCEDUREs remotas a partir do servidor local.
 - Valor 0: Ao ajustarmos Remote Access para este valor, STORED PROCEDUREs locais serão executadas a partir de um servidor remoto e STORED PROCEDUREs remotas não serão executadas no servidor local.
 - Valor 0: Ao ajustarmos Remote Access para este valor, STORED PROCEDUREs locais não serão executadas a partir de um servidor remoto e STORED PROCEDUREs remotas não serão executadas no servidor local.

9. Que palavra é utilizada na passagem de parâmetros por referência?
- a. EXECUTE.
 - b. OUTPUT
 - c. EXISTS
 - d. RETURN
 - e. UNION
10. Que acontece quando executamos a STORED PROCEDURE **SP_RECOMPILE**?
- a. Ela criptografa o código de criação da procedure, assim como foi feito com as VIEWS.
 - b. Ela criptografa e recomplia o código antes da execução da procedure.
 - c. Ela força a recompilação do código da procedure toda vez que esta for executada.
 - d. Ela recompila a procedure em sua próxima execução.
 - e. Ela altera o código de criação de uma STORED PROCEDURE.

Laboratório

Para realizar este laboratório, crie o database SISCOM com o seguinte script:

```
USE MASTER
GO
IF EXISTS (SELECT * FROM MASTER.DBO.SYSDATABASES
            WHERE NAME LIKE '%SISCOM%')
    DROP DATABASE SISCOM
GO
CREATE DATABASE SISCOM
ON PRIMARY
(
    NAME = 'SISCOM_DADOS1',
    FILENAME = 'C:\DADOS\SISCOM_DADOS1.MDF',
    SIZE = 3MB,
    MAXSIZE= 3MB,
    FILEGROWTH = 1MB
),
FILEGROUP TABELAS
(
    NAME = 'SISCOM_DADOS2',
    FILENAME = 'C:\DADOS\SISCOM_DADOS2.NDF',
    SIZE = 10MB,
    MAXSIZE= 100MB,
    FILEGROWTH = 10MB
),
FILEGROUP INDICES
(
    NAME = 'SISCOM_DADOS3',
    FILENAME = 'C:\DADOS\SISCOM_DADOS3.NDF',
    SIZE = 10MB,
    MAXSIZE= 100MB,
    FILEGROWTH = 10MB
)
LOG ON
(
    NAME = 'SISCOM_LOG',
    FILENAME = 'C:\DADOS\SISCOM_LOG.LDF',
    SIZE = 1MB,
    MAXSIZE= 10MB,
    FILEGROWTH = 1MB
)
GO
/* **** */
USE SISCOM
GO
EXEC SP_ADDTYPE 'CODIGO', 'INT', 'NOT NULL'
EXEC SP_ADDTYPE 'NOME', 'CHAR(100)', 'NOT NULL'
EXEC SP_ADDTYPE 'MOEDA', 'DECIMAL(10,2)', 'NOT NULL'
EXEC SP_ADDTYPE 'SEXO', 'CHAR(1)', 'NOT NULL'
EXEC SP_ADDTYPE 'DDD', 'CHAR(3)', 'NOT NULL'
EXEC SP_ADDTYPE 'FONE', 'CHAR(10)', 'NOT NULL'
EXEC SP_ADDTYPE 'RG', 'CHAR(15)', 'NOT NULL'
EXEC SP_ADDTYPE 'CPF', 'CHAR(20)', 'NOT NULL'
EXEC SP_ADDTYPE 'DATA', 'SMALLDATETIME', 'NOT NULL'
EXEC SP_ADDTYPE 'VALOR', 'DECIMAL(10,2)', 'NOT NULL'
GO
/* **** */
CREATE RULE R_MOEDA
AS @MOEDA >= 200.00
```

```

GO
CREATE RULE R_SEXO
AS @SEXO IN ('F', 'M')
GO
/* **** */
CREATE DEFAULT D_MOEDA
AS 200.00
GO

CREATE DEFAULT D_SEXO
AS 'F'
GO
/* **** */
EXEC SP_BINDRULE 'R_MOEDA', 'MOEDA'

EXEC SP_BINDRULE 'R_SEXO', 'SEXO'

EXEC SP_BINDEFUALT 'D_MOEDA', 'MOEDA'

EXEC SP_BINDEFUALT 'D_SEXO', 'SEXO'
GO
/* **** */
CREATE TABLE TIPOFONE
(
    COD_TIPOFONE      INT      IDENTITY      NOT NULL,
    NOME_TIPOFONE     CHAR(50)      NOT NULL,

    CONSTRAINT PK_FONE PRIMARY KEY(COD_TIPOFONE),
    CONSTRAINT UQ_FONE UNIQUE(NOME_TIPOFONE)
) ON TABELAS
GO

INSERT TIPOFONE VALUES('RESIDENCIAL')
INSERT TIPOFONE VALUES('COMERCIAL')
INSERT TIPOFONE VALUES('RECADO')
INSERT TIPOFONE VALUES('CELULAR')
GO
/* **** */
CREATE TABLE TIPOPROD
(
    COD_TIPOPROD      INT IDENTITY NOT NULL,
    NOME_TIPOPROD     CHAR(50)      NOT NULL,

    CONSTRAINT PK_PROD PRIMARY KEY(COD_TIPOPROD),
    CONSTRAINT UQ_TIPOPROD UNIQUE(NOME_TIPOPROD)
) ON TABELAS
GO

INSERT TIPOPROD VALUES('ESPECIAL')
INSERT TIPOPROD VALUES('CLASSICO')
INSERT TIPOPROD VALUES('NORMAL')
INSERT TIPOPROD VALUES('REVISADO')
INSERT TIPOPROD VALUES('RECICLADO')
GO
/* **** */
CREATE TABLE CLIENTE
(
    COD_CLI           CODIGO IDENTITY,
    NOME_CLI          NOME,
    RENDA_CLI         MOEDA,
    DATA_NASCCLI      DATA,
    SEXO_CLI          SEXO
    CONSTRAINT PK_CLI PRIMARY KEY(COD_CLI)
)

```

```

) ON TABELAS
GO

INSERT CLIENTE VALUES('OLGA CRISTINA BONFIGLIOLI',240.00,GETDATE(),'F')
INSERT CLIENTE VALUES('MARIA CRISTINA BONFIGLIOLI MARTINS DE SOUZA
SANTOS',240.00,GETDATE(),'F')
INSERT CLIENTE VALUES('SALVADOR ENEAS FEREDICO',240.00 *
@@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('DOLORES GERREIRO MARTINS',240.00 *
@@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('FABIANA BATAGLIN',240.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('APARECIDA RIBEIRO',240.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('REGINALDO RIBEIRO',240.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('SUELLEN M NUNES',240.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('CARLOS ALBERTO',240.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('ROBERTO ARRUDA',240.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('SANDRA MEDEIROS',240.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('ALICE SANTOS',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('VALTER SANCHES',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('PASCOAL BABIERA',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('LUCIA BACALLA',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('MARIA BELIDO',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('HAMILTON BELICO',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('ALBERTO BELLINI',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('MARCIA BUENO',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('MARIA CATTA',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('CARLOS CATTANEO',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('ANDRE CAULA',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('FABIA DÁVELLO',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('AFONSO FERRARO',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('AKEMI FUKAMIZU',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('BERNADINO GOMES',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('REGIONI HOKI',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('VALTER KOSZURA',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('ALEXANDRE KOZEKI',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('VITTORIO LANNOCCA',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('DOMINGOS LANINI',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('PAULO MELLO',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('ZILDA MELLONE',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('MARLENE MOURA',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('FRANCISCA OLIVEIRA',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('MARLENE PEREIRA',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('MILTON PEREIRA',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('LIGIA RAMOS',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('MARIANGELA RAMOS',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('DORA ROMARIZ',100.00 * @@IDENTITY,GETDATE(),'F')
INSERT CLIENTE VALUES('PAULINO ROMELLI',1000.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('FERNANDO SAMPAIO',1000.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('JOSÉ SAMPAIO',1000.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('VICENZO SENATORI',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('GERALDO SENEDEZE',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('MAURO SOARES',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('PAULO SOUZA',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('EMIDIO TRIFONI',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('HEITOR VERNILE',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('CARLOS SAURA',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('ANGELINO SAULLO',100.00 * @@IDENTITY,GETDATE(),'M')
INSERT CLIENTE VALUES('ALDO SAVAZZONI',100.00 * @@IDENTITY,GETDATE(),'M')

GO

SELECT * FROM CLIENTE
/* **** */
UPDATE CLIENTE

```

```

SET DATA_NASCCLI =
    CONVERT(SMALLDATETIME, CONVERT(CHAR(2), DATEPART(MM, '01/01/01')) + '/' +
    CONVERT(CHAR(2), DATEPART(DD, '01/01/01')) + '/' +
    CONVERT(CHAR(4), DATEPART(YY, '01/01/01')) - COD_CLI - 30)
GO
/* **** */
CREATE TABLE CONJUGE
(
    COD_CLI           CODIGO,
    NOME_CONJ         NOME,
    RENDA_CONJ        MOEDA,
    SEXO_CONJ         SEXO,
    CONSTRAINT PK_CONJ PRIMARY KEY(COD_CLI),
    CONSTRAINT FK_CONJ FOREIGN KEY (COD_CLI) REFERENCES CLIENTE(COD_CLI)
) ON TABELAS
GO

/* VOCE INSERE CONJUGE PARA OS 5 PRIMEIROS CLIENTES */
INSERT CONJUGE VALUES(1, 'MARINALVA', 1200, 'F')
INSERT CONJUGE VALUES(2, 'MARIANO', 1500, 'M')
INSERT CONJUGE VALUES(3, 'JOSUALDO', 2200, 'M')
INSERT CONJUGE VALUES(4, 'CREUZA', 1350, 'F')
INSERT CONJUGE VALUES(5, 'DAMARIS', 1200, 'F')
GO
/* **** */
CREATE TABLE FONE
(
    COD_FONE          CODIGO IDENTITY,
    COD_TIPOFONE      CODIGO,
    COD_CLI            CODIGO,
    NUM_FONE           FONE,
    DDD_FONE           DDD,
    CONSTRAINT PK_F PRIMARY KEY(COD_FONE),
    CONSTRAINT FK_F_1 FOREIGN KEY(COD_TIPOFONE) REFERENCES
TIPOFONE(COD_TIPOFONE),
    CONSTRAINT FK_F_2 FOREIGN KEY(COD_CLI) REFERENCES CLIENTE(COD_CLI)
) ON TABELAS
GO

INSERT FONE VALUES(1, 1, '434-2356', '011')
INSERT FONE VALUES(2, 1, '256-4578', '011')
INSERT FONE VALUES(3, 1, '256-5623', '011')
INSERT FONE VALUES(4, 2, '242-9865', '011')
INSERT FONE VALUES(1, 2, '323-8945', '011')
INSERT FONE VALUES(2, 2, '232-7845', '011')
INSERT FONE VALUES(3, 3, '565-2365', '011')
INSERT FONE VALUES(4, 3, '454-1254', '011')
INSERT FONE VALUES(1, 3, '898-2345', '011')
INSERT FONE VALUES(2, 4, '454-1223', '011')
INSERT FONE VALUES(3, 4, '787-4512', '011')
INSERT FONE VALUES(4, 5, '525-4578', '011')
INSERT FONE VALUES(1, 5, '252-9887', '011')
INSERT FONE VALUES(2, 6, '578-6521', '011')
INSERT FONE VALUES(3, 6, '568-5421', '011')
INSERT FONE VALUES(1, 8, '568-2154', '011')
INSERT FONE VALUES(1, 9, '587-3221', '011')
INSERT FONE VALUES(1, 10, '863-6598', '011')
INSERT FONE VALUES(1, 11, '138-8754', '011')
INSERT FONE VALUES(2, 12, '123-6598', '011')
INSERT FONE VALUES(2, 13, '321-6357', '011')
INSERT FONE VALUES(2, 14, '301-1232', '011')
INSERT FONE VALUES(2, 15, '321-4512', '011')
INSERT FONE VALUES(3, 16, '333-3221', '011')

```

```

INSERT FONE VALUES (3,17,'555-4578','011')
INSERT FONE VALUES (3,18,'666-1245','011')
INSERT FONE VALUES (3,19,'777-3265','011')
INSERT FONE VALUES (4,20,'888-2154','011')
INSERT FONE VALUES (4,21,'999-1111','015')
INSERT FONE VALUES (4,21,'202-1222','015')
INSERT FONE VALUES (4,22,'254-3333','015')
INSERT FONE VALUES (1,23,'458-4444','015')
INSERT FONE VALUES (1,23,'874-5555','015')
INSERT FONE VALUES (1,24,'313-6666','015')
INSERT FONE VALUES (1,24,'587-7777','015')
INSERT FONE VALUES (2,25,'589-8888','015')
INSERT FONE VALUES (2,26,'999-9999','015')
INSERT FONE VALUES (2,27,'999-1010','015')
INSERT FONE VALUES (2,27,'111-1111','015')
INSERT FONE VALUES (3,28,'222-1212','015')
INSERT FONE VALUES (3,28,'333-1313','015')
INSERT FONE VALUES (3,28,'444-1414','015')
INSERT FONE VALUES (3,29,'555-1515','015')
INSERT FONE VALUES (4,29,'666-1616','015')
INSERT FONE VALUES (4,30,'777-1717','015')
INSERT FONE VALUES (4,31,'888-1818','015')
INSERT FONE VALUES (4,32,'999-1919','015')
INSERT FONE VALUES (4,33,'101-2020','015')
INSERT FONE VALUES (4,34,'555-2121','021')
INSERT FONE VALUES (4,35,'333-2222','021')
INSERT FONE VALUES (4,36,'717-2323','021')
INSERT FONE VALUES (3,37,'656-2424','021')
INSERT FONE VALUES (3,38,'374-2525','021')
INSERT FONE VALUES (3,39,'859-2626','021')
INSERT FONE VALUES (3,40,'222-2727','021')
INSERT FONE VALUES (3,41,'256-2828','021')
INSERT FONE VALUES (3,42,'542-2929','021')
INSERT FONE VALUES (2,43,'578-3030','021')
INSERT FONE VALUES (2,44,'896-4041','021')
INSERT FONE VALUES (2,45,'369-5050','021')
INSERT FONE VALUES (2,46,'132-5151','021')
INSERT FONE VALUES (1,47,'321-6161','021')
INSERT FONE VALUES (1,48,'542-7171','011')
INSERT FONE VALUES (1,49,'201-8181','011')
INSERT FONE VALUES (1,50,'301-9191','011')
INSERT FONE VALUES (1,50,'401-1919','011')
INSERT FONE VALUES (1,50,'501-1818','011')
INSERT FONE VALUES (1,51,'601-1212','011')
INSERT FONE VALUES (1,52,'701-1313','011')

GO
/* **** */
CREATE TABLE FUNCIONARIO
(
    COD_FUNC          CODIGO IDENTITY,
    NOME_FUNC         NOME,
    DATA_CADFUNC     DATA NOT NULL DEFAULT GETDATE(),
    SEXOFUNC          SEXO,
    SAL_FUNC          MOEDA,
    END_FUNC          VARCHAR(100) NOT NULL,
    CONSTRAINT PK_FUNC PRIMARY KEY(COD_FUNC)
) ON TABELAS
GO

INSERT FUNCIONARIO VALUES ('ANTONIO ANTONINO ANTONES', '01/02/00', 'M', 1500.00, 'RUA A')
INSERT FUNCIONARIO VALUES ('AMARO MERICO VESPUCIO', '02/02/00', 'M', 2500.00, 'RUA B')

```

```

INSERT FUNCIONARIO VALUES ('ABÍLIO ABEL GARCIA', '03/02/01', 'M', 1000.00, 'RUA C')
INSERT FUNCIONARIO VALUES ('BIA BIANCA BONES', '04/03/01', 'F', 5000.25, 'RUA D')
INSERT FUNCIONARIO VALUES ('BEATRIZ BERTIOGA', '05/05/01', 'F', 300.00, 'RUA E')
INSERT FUNCIONARIO VALUES ('CAIO CESAR CEAREZ', '06/05/01', 'M', 250.00, 'RUA F')
INSERT FUNCIONARIO VALUES ('CELSO CESARE', '07/06/01', 'M', 1542.36, 'RUA J')
INSERT FUNCIONARIO VALUES ('DANILO DOUGLAS', '08/06/01', 'M', 1524.56, 'RUA K')
INSERT FUNCIONARIO VALUES ('DENIS DENILO', '09/07/01', 'M', 5235.56, 'RUA L')
INSERT FUNCIONARIO VALUES ('EVERTON EVARISTO', '10/07/01', 'M', 2542.25, 'RUA M')
INSERT FUNCIONARIO VALUES ('EVANIR EVA', '11/08/01', 'M', 4523.54, 'RUA N')
INSERT FUNCIONARIO VALUES ('FABIO FABRICIO', '12/08/01', 'M', 1524.25, 'RUA O')
INSERT FUNCIONARIO VALUES ('FABIOLA FABIOL', '02/01/02', 'F', 2554.25, 'RUA P')
INSERT FUNCIONARIO VALUES ('GERALDO GOMES', '03/01/02', 'M', 1542.25, 'RUA Q')
INSERT FUNCIONARIO VALUES ('HELIOS HELIÓPOLIS', '04/01/02', 'M', 1542.23, 'RUA R')
INSERT FUNCIONARIO VALUES ('IRINEU IRENE', '05/02/02', 'M', 2523.00, 'RUA S')
INSERT FUNCIONARIO VALUES ('JONAS JACKES', '05/02/02', 'M', 2500.00, 'RUA T')
INSERT FUNCIONARIO VALUES ('LEANDRO LAGO', '06/02/02', 'M', 1500.00, 'RUA U')
INSERT FUNCIONARIO VALUES ('LUCIO LACIO', '07/03/02', 'M', 2500.00, 'RUA V')
INSERT FUNCIONARIO VALUES ('LECIO LICIO', '08/04/02', 'M', 1420.00, 'RUA X')
INSERT FUNCIONARIO VALUES ('MARIO MENDES', '06/02/02', 'M', 1262.00, 'RUA W')
INSERT FUNCIONARIO VALUES ('OLAVO ODAVLAS', '07/07/02', 'M', 1540.00, 'RUA Y')
GO
/* **** */
CREATE TABLE DEPENDENTE
(
    COD_DEP           CODIGO  IDENTITY,
    COD_FUNC          CODIGO,
    NOME_DEP          NOME,
    SEXO_DEP          SEXO,
    DATA_NASCDEP     DATA,
    CONSTRAINT PK_DEP PRIMARY KEY (COD_DEP),
    CONSTRAINT FK_DEP FOREIGN KEY (COD_FUNC) REFERENCES FUNCIONARIO(COD_FUNC)
) ON TABELAS
GO

--INserir 3 dependentes para o FUNCIONARIO 1
INSERT DEPENDENTE VALUES(1, 'JOSEFINA', 'F', '10/15/2001')
INSERT DEPENDENTE VALUES(1, 'RENILDA', 'F', '02/20/2002')
INSERT DEPENDENTE VALUES(1, 'KATRINA', 'F', '12/30/2003')

--INserir 2 dependentes para o FUNCIONARIO 2
INSERT DEPENDENTE VALUES(2, 'CARLOS', 'M', '9/15/2001')
INSERT DEPENDENTE VALUES(2, 'JOANA', 'F', '6/15/2001')

--INserir 1 dependentes para o FUNCIONARIO 3
INSERT DEPENDENTE VALUES(3, 'CÉLIA', 'F', '7/19/2003')
GO
/* **** */
CREATE TABLE PREMIO
(
    COD_PREMIO        CODIGO IDENTITY,
    COD_FUNC          CODIGO,
    DATA_PREMIO       DATA,
    VAL_PREMIO        VALOR,
    STATUS_PREMIO    CHAR(1),

    CONSTRAINT PK_PREMIO PRIMARY KEY (COD_PREMIO),
    CONSTRAINT FK_PREMIO FOREIGN KEY (COD_FUNC) REFERENCES FUNCIONARIO(COD_FUNC),
    CONSTRAINT CH_PREMIO CHECK (STATUS_PREMIO IN ('0', '1'))
) ON TABELAS
GO
/* **** */
CREATE TABLE PEDIDO
(

```

```

NUM_PED      CODIGO IDENTITY NOT NULL,
COD_CLI       CODIGO          NOT NULL,
COD_FUNC      CODIGO          NOT NULL,
DATA_PED      DATA            NOT NULL,
VAL_PED       VALOR           NOT NULL,
CONSTRAINT PK_PED PRIMARY KEY (NUM_PED),
CONSTRAINT FK_PED1 FOREIGN KEY (COD_CLI) REFERENCES CLIENTE(COD_CLI),
CONSTRAINT FK_PED2 FOREIGN KEY (COD_FUNC) REFERENCES FUNCIONARIO(COD_FUNC)
) ON TABELAS
GO
/* **** */
CREATE PROCEDURE P_ENCHEPEDIDO
AS
DECLARE @COD_FUNC INT,
        @COD_CLI    INT,
        @COD_STA    INT,
        @DATA VARCHAR(255),
        @DATAPED SMALLDATETIME,
        @VALOR   DECIMAL(10,2)
SET @COD_FUNC = 1
SET @COD_CLI = 1
SET @COD_STA = 1

WHILE @COD_FUNC <= 20
BEGIN
    WHILE @COD_CLI < 50
    BEGIN
        SET @DATA =
CONVERT(CHAR(02),MONTH(DATEADD(MM,@COD_STA*2,GETDATE())))
        SET @DATA = @DATA + '/' + CONVERT(CHAR(02),DAY(GETDATE()) -
@COD_STA*3))
        SET @DATA = @DATA + '/' + CONVERT(CHAR(04),YEAR(DATEADD(YY,-
1*@COD_STA,GETDATE())))
        SET @DATAPED = CONVERT(SMALLDATETIME,@DATA)

        SET @VALOR = @COD_FUNC * 10
        IF @VALOR <= 240.00
            SET @VALOR = 300.00
        INSERT PEDIDO VALUES(@COD_CLI,@COD_FUNC,@DATA,@VALOR)
        SET @COD_STA = @COD_STA + 1
        SET @COD_CLI = @COD_CLI + 1
        SET @COD_STA = 1
    END
    SET @COD_FUNC = @COD_FUNC + 1
    SET @COD_CLI = 1
END
GO
/* **** */
EXEC P_ENCHEPEDIDO
GO
/* **** */
CREATE TABLE PRODUTO
(
    COD_PROD      INT IDENTITY      NOT NULL,
    COD_TIPOPROD  INT             NOT NULL,
    NOME_PROD     VARCHAR(100)     NOT NULL,
    QTD_ESTQ      INT             NOT NULL ,

```

```

VAL_UNIT           VALOR NOT NULL,
CONSTRAINT PK_PRODUTO PRIMARY KEY(COD_PROD),
CONSTRAINT FK_PRODUTO FOREIGN KEY(COD_TIPOPROD) REFERENCES
TIPOPROD(COD_TIPOPROD),
CONSTRAINT UQ_PRODUTO UNIQUE(NOME_PROD)
)
GO

INSERT PRODUTO VALUES (1,'CADERNO',10,1.25)
INSERT PRODUTO VALUES (1,'LAPIS',1,0.25)
INSERT PRODUTO VALUES (1,'BORRACHA',7,2.00)
INSERT PRODUTO VALUES (1,'CANETA',5,0.50)
INSERT PRODUTO VALUES (1,'CAMA',6,100.00)
INSERT PRODUTO VALUES (1,'CADEIRA',3,30.00)
INSERT PRODUTO VALUES (2,'GUARDA-ROUPA',2,1200.00)
INSERT PRODUTO VALUES (2,'GELADEIRA',2,3000.00)
INSERT PRODUTO VALUES (2,'TV',6,300.00)
INSERT PRODUTO VALUES (3,'COMIDA',7,15.00)
INSERT PRODUTO VALUES (3,'BEBIDA',2,1.00)
GO
/* **** */
CREATE TABLE ITENS
(
    NUM_PED  CODIGO,
    COD_PROD CODIGO,
    QTD_PROD INT,
    VAL_VEND VALOR,
    CONSTRAINT PK_ITENS PRIMARY KEY(NUM_PED,COD_PROD),
    CONSTRAINT FK_PROD_1 FOREIGN KEY (NUM_PED) REFERENCES PEDIDO(NUM_PED),
    CONSTRAINT FK_PROD_2 FOREIGN KEY (COD_PROD) REFERENCES PRODUTO(COD_PROD)
)
GO
/* **** */
INSERT ITENS
SELECT PEDIDO.NUM_PED,
       PRODUTO.COD_PROD,
       CASE
           WHEN COD_PROD - NUM_PED / 100 < 0 THEN 1
           WHEN COD_PROD - NUM_PED / 100 = 0 THEN 2
           ELSE COD_PROD - NUM_PED / 100
       END,
       CASE
           WHEN VAL_PED - NUM_PED < 0 THEN (VAL_PED - NUM_PED) * -1
           WHEN VAL_PED - NUM_PED > 100 THEN NUM_PED
           ELSE VAL_PED - NUM_PED
       END
FROM PEDIDO CROSS JOIN PRODUTO
WHERE PRODUTO.COD_PROD <= 10
GO
/* **** */
SELECT * FROM TIPOFONE
SELECT * FROM TIPOPROD
SELECT * FROM CLIENTE
SELECT * FROM CONJUGE
SELECT * FROM FONE
SELECT * FROM FUNCIONARIO
SELECT * FROM DEPENDENTE
SELECT * FROM PREMIO
SELECT * FROM PEDIDO
SELECT * FROM PRODUTO
SELECT * FROM ITENS
GO
/* **** */

```

```

-- OBTENDO INFORMAÇÕES:

USE SISCOM

SELECT * FROM SYSOBJECTS
WHERE ID > 100

EXEC SP_HELPCONSTRAINT TIPOFONE
EXEC SP_HELPCONSTRAINT TIPOPROD
EXEC SP_HELPCONSTRAINT CLIENTE
EXEC SP_HELPCONSTRAINT CONJUGE
EXEC SP_HELPCONSTRAINT FONE
EXEC SP_HELPCONSTRAINT FUNCIONARIO
EXEC SP_HELPCONSTRAINT DEPENDENTE
EXEC SP_HELPCONSTRAINT PREMIO
EXEC SP_HELPCONSTRAINT PEDIDO
EXEC SP_HELPCONSTRAINT PRODUTO
EXEC SP_HELPCONSTRAINT ITENS
/* **** */
PRINT '*.....SISTEMA SISCOM CRIADO.....*'
/* **** */

```

1. Criar uma procedure que receba o código do funcionário e um percentual de aumento salarial como parâmetro de entrada. Verificar o salário deste funcionário e o seu sexo. Se sexo = 'F' e salário < 1000.00, aplicar o aumento salarial.

Resposta:

2. Criar uma procedure que receba o código do funcionário como parâmetro e um percentual de aumento de salarial. Obter o salário e o sexo deste funcionário. Se sexo='F' e o salário < 1000, aplicar o aumento para este funcionário. Se sexo='m', diminuir o salário deste funcionário em 100.00

Resposta:

3. Criar uma procedure que receba o código do funcionário como parâmetro. Verificar se este funcionário tem dependente. Se ele possuir dependentes, aplicar um desconto de 5.00 no seu salário. Se ele não tiver dependente, aplicar um aumento salarial de 500.00 para este funcionário.

Resposta:

Capítulo 15

Transações

Transações são unidades de programação capazes de manter a consistência e a integridade dos dados. Devemos considerar uma transação como uma coleção de operações que executam uma função lógica única em uma aplicação de banco de dados.

Todas as alterações de dados realizadas durante a transação são submetidas e tornam-se parte permanente do banco de dados caso a transação seja realizada com êxito. No entanto, caso a transação não seja realizada por conta de erros, são excluídas quaisquer alterações feitas sobre os dados.

O Sistema Gerenciador de Banco de Dados tem como um de seus itens principais um esquema de recuperação capaz de encontrar falhas e de restaurar um banco de dados para seu estado consistente, ou seja, para o estado em que ele encontrava-se antes da ocorrência da falha.

É comum que uma unidade lógica de trabalho seja formada por diversas operações do banco de dados. Um exemplo que ocorre com bastante freqüência é a transferência de fundos, em que temos o débito de uma conta para o crédito em outra. Para que este tipo de operação seja realizado da forma adequada, a consistência do banco de dados é essencial, visto que ou as duas operações são realizadas, ou nenhuma é. Isso significa que, se algum erro impedir a operação de parte do débito, o crédito não será realizado. Este requerimento, em que é tudo ou nada, é denominado **atomicidade**. Cada transação é uma unidade de atomicidade.

A preservação da atomicidade é uma das questões mais importantes para o processamento de transações, mesmo com a possibilidade de ocorrência de falhas no sistema do computador, visto que isso pode acontecer com qualquer dispositivo elétrico ou mecânico. Dentro dessas falhas, podemos mencionar problemas com o fornecimento de energia elétrica, com o disco ou com o software, entre outros.

O tratamento das falhas deve ser realizado de acordo com seu tipo. As falhas mais simples são aquelas que não resultam em perda de informações. Já as falhas mais complexas e, portanto, mais difíceis de serem tratadas são aquelas que resultam na perda de informação. Este tipo de falha pode fazer com que o banco de dados perca sua consistência. Novamente, devemos ressaltar que a consistência do banco de dados depende da atomicidade das transações.

Exemplo de Transação

Os comando **INSERT**, **UPDATE** e **DELETE** são tratados individualmente como sendo uma transação. O comando **SELECT** não é tratado como uma transação porque esta é definida com sendo o comando que altera o estado inicial da tabela, coisa que o comando **SELECT** não pode fazer.

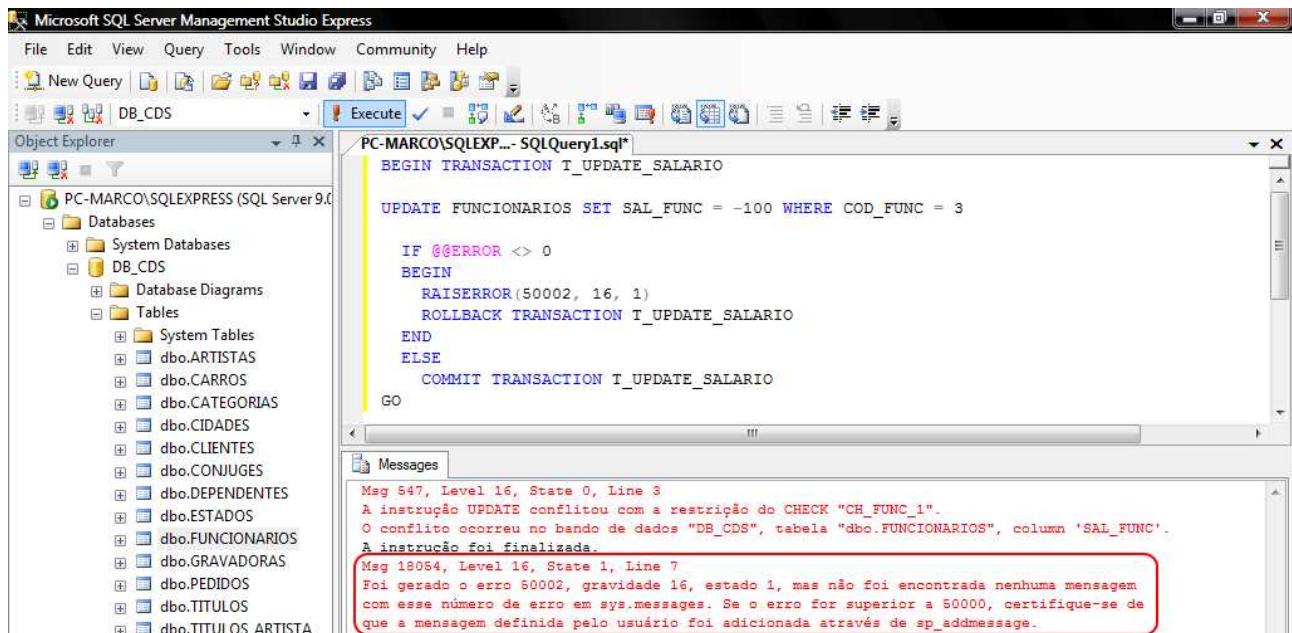
Veja o exemplo a seguir:

```
BEGIN TRANSACTION T_UPDATE_SALARIO

UPDATE FUNCIONARIOS SET SAL_FUNC = -100 WHERE COD_FUNC = 3

IF @@ERROR <> 0
BEGIN
    RAISERROR(50002, 16, 1)
    ROLLBACK TRANSACTION T_UPDATE_SALARIO
END
ELSE
    COMMIT TRANSACTION T_UPDATE_SALARIO
GO
```

Executando este script, teremos como retorno um erro, pois uma constraint **CHECK** definida na coluna **SAL_FUNC** de **FUNCIONARIOS** nos impede de alterar o salário de um funcionário para um valor inferior à 0 (zero). Vejamos o erro:



Transações Explícitas

As transações explícitas, também chamadas em versões anteriores do SQL Server de transações específicas ou definidas pelo usuário, são aquelas em que temos seu início e seu término determinados de forma explícita. Para definir este tipo de transação, os scripts Transact-SQL e as aplicações DB-Library utilizam os seguintes comandos:

- **BEGIN TRANSACTION [nome_transacao]**: Dá início a uma transação explícita.
- **COMMIT TRANSACTION [nome_transacao]**: Finaliza a transação e efetua as modificações necessárias no banco de dados.
- **ROLLBACK TRANSACTION [nome_transacao]**: Finaliza a transação no banco de dados, porém não efetua qualquer modificação, ou invés retorna o estado original do banco de dados.

O modo explícito dura apenas até o encerramento da transação, quando a conexão retorna ao modo de transação no qual se encontrava antes de a transação explícita ter início. Este modo pode ser implícito ou submetido de forma automática.

Transações Implícitas

O comando **SET** permite criar transações implícitas para aplicação que foram desenvolvidas em sistemas diferentes do SQL Server. Vejamos sua aplicação:

SET IMPLICIT_TRANSACTION { ON | OFF }

Caso já exista uma transação implícita na conexão, o comando **SET** não iniciará outra transação, visto que não são permitidas as transações aninhadas. Nas situações em que essa transação estiver configurada para **ON**, seu encerramento deve ser feito por meio da execução explícita de um dos seguintes comandos: **COMMIT** ou **ROLLBACK TRANSACTION**. Se a transação não for desfeita dessa maneira, as alterações e dados modificadas serão desfeitos no momento em que o usuário se desconectar. Por padrão, a transação será configurada para **OFF**. Quando a transação anterior é encerrada, inicia-se uma nova transação de forma implícita.

Nota: É importante salientar que caso as transações implícitas estejam no estado **ON**, qualquer tipo de transação será monitorada. Isso pode acarretar uma carga de processo muito alto por parte do servidor, fazendo com que este perca em grande parte sua performance.

Considerações Sobre Transações

Visto que uma transação muito extensa aumenta a probabilidade de os usuários não serem capazes de acessar os dados bloqueados, ela deve ser o mais curta possível. Comando da linguagem **DDL** (Data Definition Language) e comandos como o **WHILE** devem ser utilizados com cautela a fim de reduzir o tempo de processamento de uma transação.

Antes que a transação seja iniciada, devemos executar a maior quantidade possível de comandos. Além disso, com os comandos **DELETE** e **UPDATE**, devemos utilizar a cláusula **WHERE**.

Manipulando Erros

Os erros que abortam um comando ou um batch podem ocorrer quando os batches e as stored procedures remotas são executadas para o cliente a partir de uma instância local do SQL Server. Nas situações em que temos um erro que aborta um comando, ocorre o encerramento do comando que gerou o erro, porém, não ocorre o encerramento da execução do batch ou da stored procedure remota. Já nas situações em que temos um erro que aborta um batch, a execução do batch ou da stored procedure é encerrada.

Além disso, devemos ter em mente que a manipulação de erros que abortam batches pode ser realizada por uma construção do tipo **TRY / CATCH** nas situações em que a execução dos batches e das stored procedures remotas é realizada dentro do escopo de um bloco **TRY**. A configuração de **SET XACT_ABORT** do servidor local é utilizada como base para determinar o comportamento dos batches e das stored procedures remotas nas situações em que ocorrem erros que abortam comandos e batches.

SET XACT_ABORT

Conforme vimos acima, a configuração de **SET XACT_ABORT** do servidor local determina o comportamento de batches e de stored procedures remotas quando ocorrem erros que abortam comandos e batches. Quando **SET XACT_ABORT** está configurado como **ON** no servidor local, esta configuração também é aplicada ao servidor que está ligado ao mesmo. Com isso, ocorre uma conversão dos erros que abortam batches e comandos na stored procedures remotas: todos eles são convertidos apenas para erros que abortam batches. Diante de tal conversão, o encerramento da execução ocorre de forma simultânea ao encerramento desta última.

Nas situações em que a execução da stored procedure remota que gerou o erro ocorre no escopo do bloco **TRY** no servidor local, as informações a respeito do último erro ocorrido no servidor remoto são passadas pelo controle para o bloco **CATCH**. Já nas situações em que a

stored procedure remota não é executada no escopo de um bloco **TRY**, o valor de **@@ERROR** não pode ser verificado.

Nota: O valor de **@@ERROR** permite verificar se a não-execução do comando posterior ao **EXECUTE** foi a responsável por causar o erro de batch.

A execução mais adequada da stored procedure remota deve ocorrer a partir do bloco **TRY** referente a uma construção **TRY / CATCH**. Com isso, caso haja problemas para que tal procedure seja finalizada, a execução é direcionada ao bloco **CATCH** associado, o qual se encontra no servidor local e possui informações a respeito do último erro gerado no servidor remoto. Já nas situações em que a finalização dessa procedure ocorrem sem problemas, a execução segue seu curso normal dentro do bloco **TRY** no servidor local. O resultado obtido com esta execução pode ser utilizado normalmente.

RAISERROR

Este comando é responsável por gerar uma mensagem de erro, a qual pode ser construída de forma dinâmica. Além disso, **RAISERROR** pode fazer referência a uma mensagem definida pelo usuário que se encontra armazenada em **SYSMESSAGES**, uma **VIEW** de catálogo. Essa mensagem é retornada para a aplicação que fez a chamada ou para o bloco **CATCH** associado. Esse retorno ocorre como se a mensagem fosse um erro do servidor.

RAISERROR gera erros que funcionam de forma semelhante aos erros gerados pelo código Database Engine. Os valores referentes a esses erros são relatados pelas seguintes funções de sistema: **@@ERROR**, **ERROR_LINE**, **ERROR_MESSAGE**, **ERROR_NUMBER**, **ERROR_PROCEDURE**, **ERROR_SEVERITY** e **ERROR_STATE**.

Vale destacar, também, que **RAISERROR** pode ser utilizada por blocos **CATCH** com a finalidade de lançar novamente um erro que invocou este tipo de bloco utilizando funções de sistema para recuperar dados a respeito do erro original.

Nota: A inclusão de uma mensagem definida pelo usuário é feita por meio do **SP_ADDMESSAGE**. Já sua exclusão é feita por meio do **SP_DROPMESSAGE**.

Visto que o **RAISERROR**, ao contrário do comando **PRINT**, é capaz de suportar a substituição de caracteres, ele pode ser utilizado como alternativa a este comando com a finalidade de retornar mensagens às aplicações que as chamaram. Para que o **RAISERROR**

possa retornar uma mensagem do bloco **TRY** sem ter que invocar o bloco **CATCH**, é preciso especificar um valor de severidade 10 ou inferior. O bloco **TRY** não afeta o comando **PRINT**. Vale destacar que os argumentos substituem as especificações de conversão de forma sucessiva. Veja o exemplo:

```
RAISERROR ( N'Essa é uma mensagem %s %d.', -- Texto da mensagem.
             10, -- Severidade,
             1, -- Estado,
             N'número', -- Primeiro argumento.
             5); -- Segundo argumento.

GO
```

Execute esse script e observe o resultado.

A Variável @@ERROR

Esta variável tem a finalidade de retornar o número referente ao erro após a execução de um comando Transact-SQL. Caso este número seja referente a um dos erros definidos na **VIEW** de catálogo **SYSMESSAGES**, o valor de **@@ERROR** é referente à coluna **SYSMESSAGES.MESSAGE_ID** relativa ao erro ocorrido. A partir de **SYSMESSAGES**, é possível verificar o texto correspondente ao erro.

O valor de **@@ERROR** deve ser verificado de forma imediata ou deve ser salvo em uma variável local a fim de que se possa ser verificado posteriormente. Isso se faz necessário porque esse valor é excluído e **@@ERROR** é reiniciado a cada comando executado. Observe o exemplo a seguir:

```
DECLARE @ERRO INT

UPDATE FUNCIONARIOS SET SAL_FUNC = -100 WHERE COD_FUNC = 3

SET @ERRO = @@ERROR

IF @ERRO <> 0
    PRINT N'O número do erro gerado foi ' + CAST(@ERRO AS VARCHAR(10))

GO
```

Parecido com um exemplo anterior, tentamos alterar o valor do salário de um funcionário para um valor negativo, a constraint **CHECK** é acionada retornando um erro que por sua vez é atribuído a uma variável chamada **@ERRO**. Essa variável é por fim impressa na tela com a utilização do comando **PRINT**. Execute o comando e veja o resultado.

Criando Uma Mensagem de Erro

Para criarmos uma mensagem de erro, é necessário fazer uso da stored procedure **SP_ADDMESSAGE**, que tem a seguinte sintaxe:

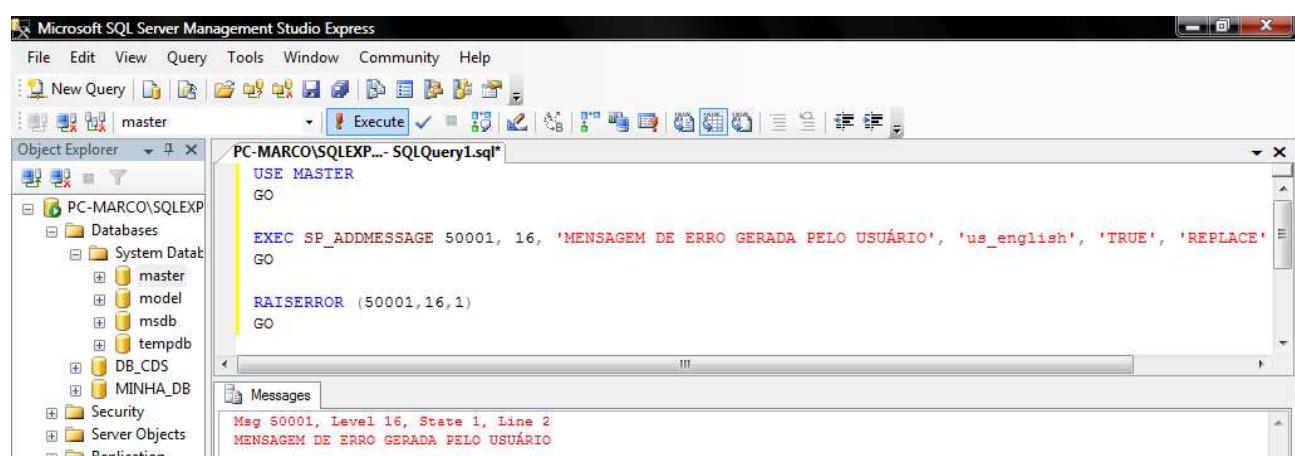
```
EXEC SP_ADDMESSAGE [numero_msg], [severidade_msg], '[texto_msg]', '[lingua_msg]',  
'[log_msg]', '[alterar_msg]'
```

Em que:

- **[numero_msg]**: Número da mensagem. Para mensagens definidas pelo usuário os valores devem ser maiores que 50000. É necessário que a combinação entre **[numero_msg]** e **[lingua_msg]** seja única. Datatype: **INT**.
- **[severidade_msg]**: Valor da severidade do erro. Os administradores são os únicos a emitir severidades maiores que 18 (entre 19 e 25), severidades essas que irão desconectar o usuário. Datatype: **SMLALLINT**.
- **[texto_msg]**: Texto referente a mensagem. Datatype: **NVARCHAR(255)**.
- **[lingua_msg]**: Refere-se a língua utilizada para se escrever a mensagem. Se omitida, receberá o valor da sessão em uso. O código das línguas pode ser encontrada pela **VIEW SYS.SYSLANGUAGES**. Datatype: **NVARCHAR(128)**.
- **[log_msg]**: Determina se a mensagem de erro deve ser escrita no log de eventos do Windows Server e no **ERROR LOG** do SQL Server. Datatype: **BIT**.
- **[alterar_msg]**: Determina se a mensagem poderá ser alterada. Datatype: **VARCHAR(7)**. O valor que deve ser indicado para alteração é **REPLACE**.

Vejamos um exemplo:

```
USE MASTER  
GO  
  
EXEC SP_ADDMESSAGE 50001, 16, 'MENSAGEM DE ERRO GERADA PELO USUÁRIO',  
'us_english', 'TRUE', 'REPLACE'  
GO  
  
RAISERROR (50001,16,1)  
GO
```



Nesse caso, criamos essa mensagem no database **MASTER**, mas ela pode ser criada na database do usuário. A vantagem de se criar na **MASTER** é que a mensagem fica acessível a qualquer usuário.

Como essa mensagem foi definida para escrever no log do Windows sempre que acontecer, vejamos o Windows Logs:

The screenshot shows the Windows Event Viewer interface. On the left, the navigation pane lists various system tools and the Event Viewer section, with 'Windows Logs' and 'Application' selected. The main pane displays a table of events. One event is highlighted in blue, showing the following details:

Level	Date and Time	Source	Event ID	Task Category
Error	10/07/2007 13:45:51	MSSQL\$SQLEXPRESS	17063	(2)
Error	10/07/2007 13:45:00	MSSQL\$SQLEXPRESS	17063	(2)
Error	10/07/2007 13:45:00	MSSQL\$SQLEXPRESS	17063	(2)
Information	10/07/2007 13:42:57	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:42:57	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:42:57	MSSQL\$SQLEXPRESS	2803	(2)
Error	10/07/2007 13:12:42	MSSQL\$SQLEXPRESS	18054	(2)
Error	10/07/2007 13:12:22	MSSQL\$SQLEXPRESS	18054	(2)
Information	10/07/2007 13:12:20	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:12:20	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:12:20	MSSQL\$SQLEXPRESS	17137	(2)
Information	10/07/2007 13:02:52	MSSQL\$SQLEXPRESS	17137	(2)
Information	10/07/2007 13:02:51	MSSQL\$SQLEXPRESS	17137	(2)
Information	10/07/2007 13:02:50	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:02:50	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:02:50	MSSQL\$SQLEXPRESS	2803	(2)
Information	10/07/2007 13:02:50	MSSQL\$SQLEXPRESS	17137	(2)

A modal window titled 'Event 17063, MSSQL\$SQLEXPRESS' is open, showing the 'General' tab. It contains the message: 'Erro: 50001, Gravidade: 16, Estado: 1 MENSAGEM DE ERRO GERADA PELO USUÁRIO'. This message is highlighted with a red box. Below it, the 'Details' tab shows the following event properties:

Log Name:	Application
Source:	MSSQL\$SQLEXPRESS
Event ID:	17063
Level:	Error
User:	pc-marco\Administrador
OpCode:	
Logged:	10/07/2007 13:45:51
Task Category:	(2)
Keywords:	Classic
Computer:	pc-marco

Para acessar o Windows Logs, vá em **INICIAR / PROGRAMAS / FERRAMENTAS ADMINISTRATIVAS / EVENT VIEWER**.

Eliminando Uma Mensagem de Erro

Para eliminarmos uma mensagem de erro, usamos a stored procedure **SP_DROPMESSAGE**. A sintaxe básica é:

```
EXEC SP_DROPMESSAGE [numero_msg], @lang = '[lingua_msg]'
```

```
EXEC SP_DROPMESSAGE 50001, @lang = 'us_english'
GO
```

Batch

Definimos batches como sendo um grupo composto por um ou mais comandos SQL que uma aplicação envia, de uma única vez, para serem executados pelo SQL Server. Caso haja algum erro de compilação, o Plano de Execução deixa de ser executado, o que significa que os comandos que compõem o batch também não são executados. Já um erro ocorrido em tempo de execução poderá apresentar um dos seguintes efeitos:

- Grande parte dos erros ocorridos em tempo de execução paralisam o comando atual e o seguinte do batch. Apenas alguns erros em tempo de execução paralisam somente o comando atual, sem afetar os outros comandos batch.
- O comando executado antes da ocorrência do erro em tempo de execução não é afetado. Apenas há uma exceção, que ocorre nas situações em que o batch está em uma transação e o erro causa um **ROLLBACK**. Quando isso ocorre, qualquer comando que não tenha sido submetido será desfeito.

Exercícios

1. Que são transações? Como elas funcionam?

Resposta:

2. Quando ocorrem falhas, o que pode acontecer com as informações?

Resposta:

3. Como ocorre a manipulação de erros nas transações?

Resposta:

4. Que é um batch?

Resposta:

5. Quais são os efeitos que podem ser apresentados por um erro ocorrido em tempo de execução?

Resposta:

6. O que acontece quando o **SET XACT_ABORT** está configurado como **ON** no servidor local?
 - a. É gerada uma mensagem de erro, a qual pode ser construída de forma dinâmica.
 - b. Ocorre uma conversão dos erros que abortam batches e comandos na stored procedure remota: todos eles são convertidos apenas para erros que abortam batches.
 - c. Ocorre a exclusão da mensagem definida pelo usuário, feita por meio do **SP_DROPMESSAGE**.
 - d. É gerada uma mensagem de erro, a qual só pode ser construída de forma não-dinâmica.
 - e. Ocorre uma conversão dos erros que abortam somente os comandos na stored procedure remota.
7. Que comando pode fazer referência a uma mensagem definida pelo usuário, que se encontra armazenada em **SYSMESSAGES**, uma **VIEW** de catálogo?
 - a. ROLLBACK
 - b. COMMIT
 - c. RAISERROR
 - d. RETURN
 - e. INSERT
8. Qual a função da variável **@@ERROR**?
 - a. Armazenar uma mensagem de erro utilizando uma stored procedure de sistema **SP_ADDMESSAGE**
 - b. Permite alterar o texto de uma mensagem de erro.
 - c. Determinar se a mensagem deve ser escrita no log de aplicações do Windows NT e no Error Log do SQL Server.
 - d. Esta variável tem a função de retornar o número referente ao erro após a execução de um comando Transact_SQL
 - e. Armazenar o nível de severidade do erro.

9. Que é plano de Execução?
- a. São comandos responsáveis por iniciar a transação nas situações em que não ocorrem erros.
 - b. É uma conversão dos erros que abortam batches e comandos na stored procedure remota.
 - c. Conjunto de transações específicas ou definidas pelo usuário em versões anteriores do SQL Server.
 - d. São comandos responsáveis por encerrar a transação com êxito nas situações em que não ocorrem erros.
 - e. São os comandos que compõem um batch e são compilados pelo SQL Server em uma única unidade executável.
10. Que comandos não podem ser combinados em um Batch?
- a. DROP DATABASE, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, EXECUTE
 - b. CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, CREATE VIEW
 - c. EXECUTE, CREATE PROCEDURE, DROP RULE, CREATE TRIGGER, CREATE VIEW
 - d. CREATE DEFAULT, DROP PROCEDURE, CREATE RULE, DROP TRIGGER, CREATE VIEW
 - e. CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, CREATE VIEW

Laboratório

Crie uma procedure (utilizando transações) que receba o código do funcionário como parâmetro e verifique o total do último pedido que este funcionário atendeu. Se este total for maior do que 50,00 aplicar um prêmio de 10% do valor do pedido para este funcionário (gravando uma linha correspondente na tabela PREMIO) e atribuir um desconto de 10% no valor deste último pedido. Para que possamos realizar este laboratório, usaremos o database SISCOM:

Resposta:

Capítulo 16

Funções

Trata-se de um tipo de programa criado em um determinado banco de dados para retornar um valor específico ou uma série de valores. Esses valores podem ser retornados para outra função, para uma aplicação, para uma **STORED PROCEDURE** ou diretamente para o usuário.

Regras Para Utilização de Funções

Antes de trabalharmos com funções no SQL Server, devemos conhecer uma série de regras necessárias para sua utilização, como veremos a seguir.

- Os datatypes **CURSOR**, **TABLE** ou **TIMESTAMP** não podem ser utilizados nos parâmetros de entrada de uma função.
- O retorno obtido a partir de uma função pode ser tanto os dados de uma tabela como um valor escalar.
- Uma função que foi definida pelo usuário não aceita parâmetros de saída (também conhecidos como parâmetros **OUTPUT**), mas é capaz de receber parâmetros de entrada. Neste tipo de função, não é possível a execução de **UPDATE**, **INSERT** e **DELETE**, uma vez que esses comandos podem comprometer os dados das tabelas armazenadas no disco. Porém, podemos utilizá-los para a manipulação de dados de variáveis de memória do datatype **TABLE**.
- Uma permissão de **SELECT** é exigida para a execução de uma **USER DEFINED FUNCTION** por parte do usuário.
- Cada vez que uma questão é executada com o mesmo conjunto de parâmetros de entrada, um valor diferente é retornado pelas funções não-determinísticas. As funções não-determinísticas internas do SQL Server (também conhecidas como **built-in**) não podem ser executadas pelas funções definidas pelo usuário.
- A execução de uma função escalar exige a utilização do comando **SELECT** e do nome qualificado por duas partes. Isso significa que, antes no nome da função, é preciso utilizar o nome do seu schema para que essa função seja realizada com sucesso.

Funções Built-In

As funções built-in podem ser utilizadas em comandos Transact-SQL para a obtenção de dados das tabelas do SQL Server sem a necessidade de acessá-las diretamente. Essas funções permitem, ainda, a execução de tarefas comuns ao SQL Server, como **SUM()**, **GETDATE()** ou **IDENTITY()**. Quando utilizadas, as funções built-in retornam valores com datatypes escalares ou tabulares. É importante lembrar que não é possível modificá-las.

Tipos de USER DEFINED FUNCTIONS

Por meio das UDFs, podemos executar uma determinada ação para obter um valor, que pode ser tanto um único valor escalar como um conjunto de valores. Essas rotinas ainda são capazes de aceitar diversos parâmetros, da mesma forma que a maioria das funções em outras linguagens de programação.

Podem ser criados três tipos diferentes te UDFs:

- Funções em que os dados de uma determinada tabela são retornados por meio de um único comando **SELECT** (In-Line).
- Funções que contêm diversos comandos e são capazes de retornar os dados de uma determinada tabela.
- Funções conhecidas como escalares.

Funções Escalares

Similares às funções built-in, retornam um único valor. Esse valor retornado possui um datatype definido na cláusula **RETURNS**. Os parâmetros de entrada podem ser aceitos ou recusados por meio dessas funções, que podem ser utilizadas sempre que uma expressão é válida.

O valor escalar de uma função in-line é obtido a partir de um só comando, sendo que não existe corpo de função neste caso. Já uma função que possui vários comandos Transact-SQL é capaz de retornar um valor de qualquer datatype. O corpo desta função é definido no bloco **BEGIN / END**.

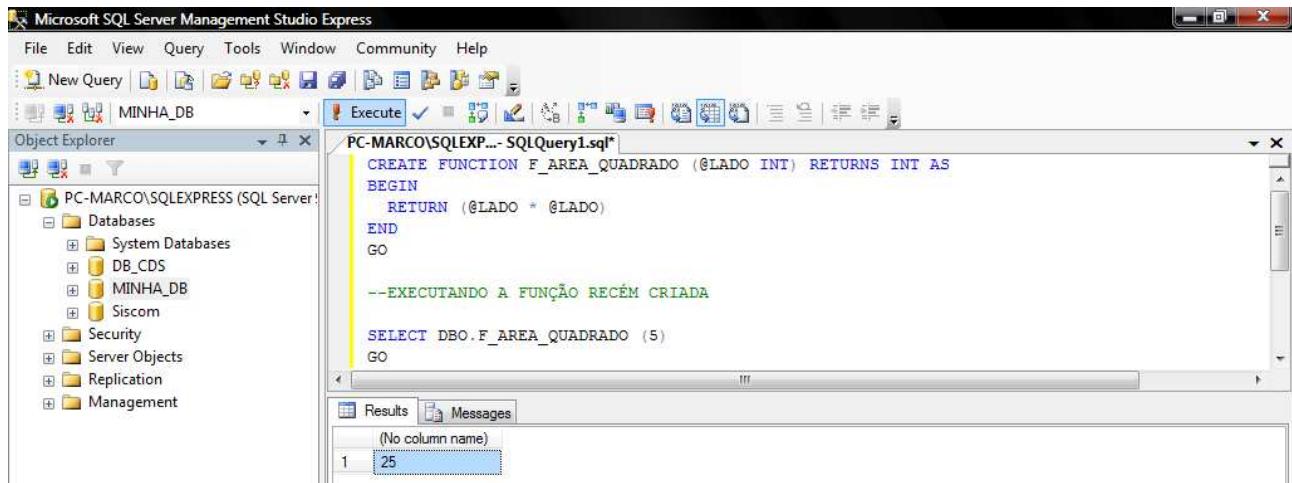
Vejamos um exemplo de uma função escalar:

```
CREATE FUNCTION F_AREA_QUADRADO (@LADO INT) RETURNS INT AS
BEGIN
    RETURN (@LADO * @LADO)
END
GO

--EXECUTANDO A FUNÇÃO RECÉM CRIADA

SELECT DBO.F_AREA_QUADRADO (5)
GO
```

Como resposta temos que a área do quadrado é 25. Observe a imagem:



Funções Table-Valued

As funções table-valued utilizam um único comando **SELECT** para retornar um conjunto de resultados em forma de tabela, já que o datatype desse valor retornado é **TABLE**. Caso a função table-valued seja in-line, não existirá corpo de função. Vejamos um exemplo:

```
CREATE FUNCTION F_DATA_CADASTRO (@DATA SMALLDATETIME) RETURNS TABLE AS
    RETURN (SELECT * FROM DBO.FUNCIONARIO WHERE CONVERT(CHAR(08),
DATA_CADFUNC) = CONVERT(CHAR(08), @DATA))
GO

--EXECUTANDO A FUNÇÃO RECÉM CRIADA F_DATA_CADASTRO
SELECT * FROM F_DATA_CADASTRO ('2001-05-05')
GO
```

Crie e execute a função a cima (será necessário o banco de dados SISCOM, visto no laboratório anterior).

Funções Que Contêm Vários Comandos e Que Retornam Dados de Uma Tabela

Para executarmos o exemplo adiante, inicialmente, devemos criar a tabela **USUARIOS** e, também, inserir uma nova coluna na tabela **FUNCIONARIO**. Esta coluna deve ser nomeada como **NUM_REGIAO**, do tipo **TINYINT**. (Use o database **SISCOM**):

```
CREATE TABLE USUARIOS
(
    NUM_REGIAO TINYINT,
    NOME_USUARIO VARCHAR(20)
)
GO

INSERT INTO USUARIOS VALUES (10, 'MARCO')
GO

ALTER TABLE FUNCIONARIO ADD NUM_REGIAO TINYINT
GO

CREATE FUNCTION F_REGIAO() RETURNS @FUNC TABLE (
    NOME_FUNC VARCHAR(100) NOT NULL,
    SAL_FUNC DECIMAL(10,2) NOT NULL
) AS
BEGIN
    DECLARE @NUMREG TINYINT

    SELECT @NUMREG = NUM_REGIAO FROM USUARIOS WHERE NOME_USUARIO = 'MARCO'

    IF @NUMREG IS NOT NULL AND @NUMREG <> 10
        INSERT INTO @FUNC
            SELECT NOME_FUNC, SAL_FUNC FROM FUNCIONARIO
            WHERE NUM_REGIAO = @NUMREG
    ELSE IF @NUMREG = 10
        INSERT INTO @FUNC
            SELECT NOME_FUNC, SAL_FUNC FROM FUNCIONARIO

    RETURN
END
GO

--EXECUTANDO A FUNÇÃO F_REGIAO()
SELECT * FROM DBO.F_REGIAO()
GO
```

Exercícios

1. Que são funções?

Resposta:

2. Onde podemos utilizar a função built-in?

Resposta:

3. Que são UDFs e quais são os tipos que podem ser criados?

Resposta:

4. Que são funções escalares? Como elas funcionam?

Resposta:

5. Como funcionam as funções table-valued?

Resposta:

6. Quais das alternativas a seguir apresentam regras que devem ser consideradas quando trabalhamos com funções?
 - a. Os valores escalares representam o único retorno que pode ser obtido a partir de uma função.
 - b. Uma função que foi definida pelo usuário não aceita parâmetros de output.
 - c. Uma permissão de SELECT é exigida para a execução de uma User Defined Function por parte do usuário.
 - d. A execução de uma função escalar exige a utilização do comando SELECT e do nome qualificado por duas partes.
 - e. Os datatypes cursor, table ou timestamp podem ser utilizados nos parâmetros de entrada de uma função no SQL Server.
7. Que função pode ser utilizada em comandos Transact-SQL para a obtenção de informações das tabelas do SQL Server sem a necessidade de acessar diretamente essas tabelas?
 - a. User Defined Function
 - b. Escalares
 - c. Table-Valued
 - d. Built-In
 - e. Não-escalares
8. Qual das alternativas apresenta as palavras que devem ser inseridas nos espaços em branco do parágrafo a seguir? Similares às funções _____, as funções _____ retornam um único valor, pois operam com um só valor. Este valor retornado possui um datatype definido na cláusula _____.
 - a. User Defined Function / built-in / where
 - b. Escalares / User Defined Function / if
 - c. built-in / escalares / RETURNS
 - d. Escalares / built-in / where
 - e. built-in / escalares / else

Observe as afirmativas a seguir:

- I. As funções table-valued utilizam um único comando SELECT para retornar um conjunto de resultados em forma de tabela, já que o datatype desse valor retornado é table. Caso a função table-valued seja inline, existirá corpo de função.
 - II. O valor escalar de uma função inline é obtido a partir de um só comando, sendo que não existe corpo de função nesse caso. Já uma função que possui vários comandos Transact-SQL é capaz de retornar um valor de qualquer datatype.
 - III. As funções built-in podem ser utilizadas em comandos Transact-SQL para a obtenção de informações das tabelas do SQL Server sem a necessidade de acessar diretamente essas tabelas. Essas funções não permitem a execução de tarefas comuns ao SQL Server, como o uso de IDENTITY.
9. Qual das alternativas a seguir está correta?
- a. As afirmativas I e III são falsas.
 - b. Somente a alternativa III é falsa.
 - c. Todas as afirmativas são verdadeiras.
 - d. Todas as afirmativas são falsas.
 - e. Somente a afirmativa II é falsa.
10. Que funções são também conhecidas como funções não-determinísticas internas do SQL Server?
- a. Table-Valued
 - b. Built-in
 - c. Escalares
 - d. user defined functions
 - e. Inline

Capítulo 17

TRIGGERS

Definimos um **TRIGGER** como sendo um **STORED PROCEDURE** específica cuja execução ocorre no momento em que é realizada uma alteração sobre os dados de uma tabela. São dois os principais tipos de **TRIGGERS** disponibilizados pelo SQL Server: **DDL** e **DML**.

Os **TRIGGERS DDL** podem ser utilizados com a finalidade de realizar tarefas administrativas. Suas funções disparam **STORED PROCEDURES** para responder aos comandos **DDL**, os quais são, principalmente, iniciados com **ALTER**, **DROP** e **CREATE**. Já **TRIGGERS DML** são utilizados nas situações em que os comandos **INSERT**, **DELETE** e **UPDATE** realizam a alteração na tabela ou **VIEWS**.

O acionamento dos **TRIGGERS** ocorre de forma automática, e eles não recebem e, tampouco, devolvem qualquer tipo de valor. Apesar disso, eles podem gerar erros com o comando **RAISERROR**. Os **TRIGGERS** podem ser criados a fim de que seu acionamento ocorra quando os comandos **DML** ou **DDL** são utilizados.

TRIGGER x Transação

Tanto o **TRIGGER** como o comando que o acionou são tratados como transações. Esta transação pode ser desfeita em qualquer lugar de dentro do **TRIGGER** com o comando **ROLLBACK TRANSACTION**. Caso seja utilizado, o **ROLLBACK** irá desfazer tudo o que o **TRIGGER** tenha feito até o momento e também o que o comando fez antes do **TRIGGER**.

Para que possamos ver um exemplo funcional, vamos criar duas tabelas (**FUNCIONARIOS** e **HISTORICOS**). Utilize o database **MINHA_DB** para esse exemplo, caso o database já possua uma tabela chamada **FUNCIONARIOS**, "drope-a". Elimine também as **VIEWS** associadas, caso seja necessário.

Tabela: **FUNCIONARIOS**

COD_FUNC (INT IDENTITY)	NOME_FUNC (VARCHAR(50))	IDADE_FUNC (TINYINT)	SAL_FUNC (DECIMAL(9,2))
1	CARLOS	35	2500.00
2	RENATO	36	2000.00
3	JOAO	34	3000.00
4	JOANA	18	1000.00

5	CARMEN	27	700.00
6	RAQUEL	25	3500.00
7	MARIA	28	2000.00

Tabela: **HISTORICOS**

COD_FUNC (INT)	SAL_FUNC (DECIMAL(9,2))	DATA_SAL (SMALLDATETIME)
1	2500.00	25/02/06
2	2000.00	30/03/06
3	3000.00	24/04/06
4	1000.00	25/06/06
5	700.00	27/07/06
6	3500.00	28/07/06

Com o **TRIGGER** demonstrado a seguir, torna-se possível gravar os dados alterados da tabela **FUNCIONARIOS** na tabela **HISTORICOS**:

```
CREATE TRIGGER T_INC_HISTORICO ON FUNCIONARIOS FOR INSERT AS
    INSERT INTO HISTORICOS
        SELECT COD_FUNC, SAL_FUNC, GETDATE() FROM INSERTED
GO
```

Agora, para que possamos ver o funcionamento do **TRIGGER**, inclua um novo registro na tabela funcionários. Depois, use o comando **SELECT** na tabela **HISTORICOS** e observe os registros.

```
INSERT INTO FUNCIONARIOS VALUES ('MARCO', 25, 2000.00)
GO

SELECT * FROM HISTORICOS
GO
```

Observe a imagem abaixo:

The screenshot shows the Microsoft SQL Server Management Studio Express interface. In the Object Explorer, the database 'MINHA_DB' is selected. In the center pane, a query window titled 'PC-MARCO\SQLEXPRESS - SQLQuery1.sql*' contains the following T-SQL code:

```
CREATE TRIGGER T_INC_HISTORICO ON FUNCIONARIOS FOR INSERT AS
    INSERT INTO HISTORICOS
        SELECT COD_FUNC, SAL_FUNC, GETDATE() FROM INSERTED
    GO

    INSERT INTO FUNCIONARIOS VALUES ('MARCO', 25, 2000.00)
    GO

    SELECT * FROM HISTORICOS
    GO
```

The 'Results' tab of the query window displays the output of the 'SELECT * FROM HISTORICOS' command, showing the following data:

	COD_FUNC	SAL_FUNC	DATA_SAL
1	1	2500.00	2006-02-25 00:00:00
2	2	2000.00	2006-03-30 00:00:00
3	3	3000.00	2006-04-24 00:00:00
4	4	1000.00	2006-06-25 00:00:00
5	5	700.00	2006-07-27 00:00:00
6	6	3500.00	2006-07-28 00:00:00
7	8	2000.00	2007-07-11 13:53:00

O comando **INSERT**, portanto, aciona o **TRIGGER** que executa o código interno. Este tipo de **TRIGGER**, cujo acionamento ocorre no momento em que dados são inclusos na tabela, é capaz de executar internamente qualquer um dos seguintes comandos: **SELECT**, **UPDATE**, **INSERT** e **DELETE**. Os únicos comandos que não podem ser executados pelos **TRIGGERs** são:

- **CREATE (todos)**
- **DROP (todos)**
- **DISK (todos)**
- **GRANT**
- **LOAD**
- **REVOKE**
- **ALTER TABLE**
- **ALTAR DATABASE**
- **TRUNCATE TABLE**
- **UPDATE STATISTICS**
- **RECONFIGURE**
- **RESTORE DATABASE**
- **RESTORE LOG**
- **SELECT INTO**

O comando **TRUNCATE TABLE** não é capaz de realizar a execução de um **TRIGGER** devido ao fato de não estar logado. Já o comando **WRITETEXT** não é capaz de ativar um **TRIGGER**, independente do fato de estar ou não logado.

TRIGGERS São Reativas

As constraints que já estão prontas podem ser adicionadas na tabela a fim de que as regras de negócios comuns sejam mantidas. Já para construir regras de negócios mais complexas, é preciso utilizar os **TRIGGERS**, os quais, diferentemente das constraints, são reativos. Isso significa que, utilizando os **TRIGGERS**, o SQL Server primeiramente executa o comando que o acionou e, depois, verifica a ocorrência de erros; caso haja um erro, o comando é desfeito.

Já as constraints são pré-ativas, ou seja, elas são verificadas pelo SQL Server a fim de assegurar que não estejam violadas antes que o comando seja executado. Caso as constraints estejam com problemas, uma mensagem de erro é enviada e a operação é abortada. O comando apenas é executado se a constraint não apresentar problemas.

Tabelas Criadas Em Tempo de Execução

O SQL Server cria na memória uma ou duas tabelas no decorrer da execução de um **TRIGGER**. Essas tabelas têm a função de armazenar os dados com os quais trabalhamos. O fato de criar uma ou duas tabelas depende do tipo de **TRIGGER** que está sendo utilizado. Vejamos a seguir quais são esses tipos de **TRIGGERS** e as tabelas criadas pelo SQL Server.

- Os **TRIGGERS** de **INSERT** realizam a criação de uma tabela chamada **INSERTED**.
- Os **TRIGGERS** de **DELETE** realizam a criação de uma tabela chamada **DELETED**.
- Os **TRIGGERS** de **UPDATE** realizam a criação de duas tabelas: **INSERTED** e **DELETED**. Isso ocorre porque o comando **UPDATE** não "atualiza" o registro de uma tabela. O comando exclui o registro antigo e inclui o registro com os novos valores.

TRIGGER DDL

Para demonstrar o **TRIGGER DDL**, iremos criar um database chamado **TESTE** e uma tabela com as colunas **CODIGO (INT)** e **TEXTO (VARCHAR(20))** chamada **MINHA_TABELA**.

```
CREATE DATABASE TESTE
GO

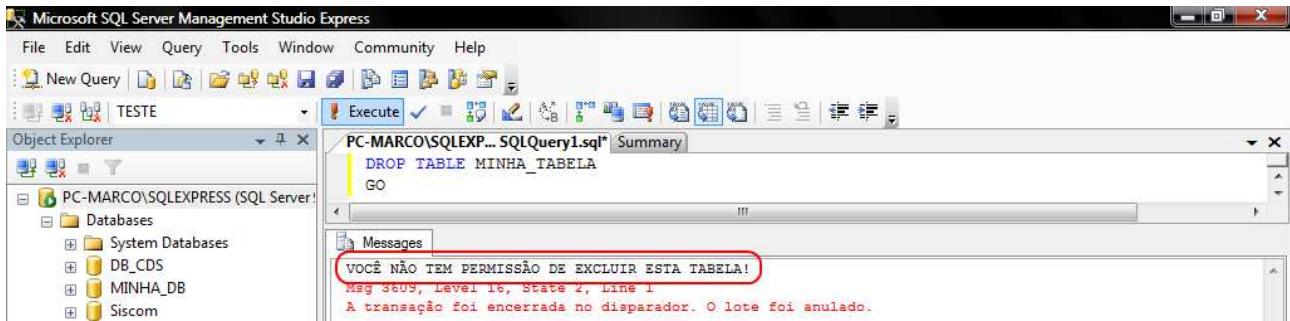
USE TESTE
GO

CREATE TABLE MINHA_TABELA
(
    CODIGO INT,
    TEXTO VARCHAR(20)
)
GO
```

Agora criaremos um **TRIGGER** que será acionado assim que o usuário tentar excluir a nossa tabela. Vejamos:

```
CREATE TRIGGER T_NAO_EXC_TABELA ON DATABASE FOR DROP_TABLE, ALTER_TABLE AS
    PRINT 'VOCÊ NÃO TEM PERMISSÃO DE EXCLUIR ESTA TABELA!'
    ROLLBACK TRANSACTION
GO
```

Se tentarmos excluir a tabela, receberemos uma mensagem de erro:



Alterando ou Eliminando Um TRIGGER

Para alterarmos um **TRIGGER** já criado, basta que utilizemos o comando **ALTER TRIGGER**, da seguinte forma:

```
ALTER TRIGGER [nome_trigger] ON [{ DATABASE | nome_tabela }] FOR [comando] AS
[corpo_trigger]
```

Para eliminar um **TRIGGER**, usamos o comando **DROP TRIGGER [nome_trigger]**.

TRIGGERS Aninhados

Um **TRIGGER** pode conter os comandos **UPDATE**, **DELETE** ou **INSERT** capazes de afetar outras tabelas. Quando o aninhamento está habilitado, um **TRIGGER** que realizar alterações sobre uma tabela pode ativar um segundo **TRIGGER**, o qual, por sua vez, pode ativar um terceiro **TRIGGER** e assim sucessivamente.

Como podemos perceber, se o aninhamento de **TRIGGERS** não for bem formulado, esse poderia gerar um loop infinito no servidor causando uma possível queda de sistema. O que não seria nada agradável. Por este motivo, o SQL Server permite no máximo um aninhamento de 32 níveis, ou seja, caso um **TRIGGER** chame outro, o limite de execuções são 32.

O aninhamento de **TRIGGERS** é configurado no momento da instalação do SQL Server, mas também é possível habilitado posteriormente. Para habilitar o aninhamento de **TRIGGERS**, devemos usar a **STORED PROCEDURE SP_CONFIGURE**. Observe:

```
-- DESABILITA TRIGGERS ANINHADOS
EXEC SP_CONFIGURE 'NESTED TRIGGERS', 0
GO

-- HABILITA TRIGGERS ANINHADOS
EXEC SP_CONFIGURE 'NESTED TRIGGERS', 1
GO
```

INSTEAD OF

Este comando é responsável por determinar que o **TRIGGER DML** seja executado ao invés do comando SQL ser disparado. Com isso, as ações realizadas por comandos disparadores são sobreescritas. Apenas um **TRIGGER INSTEAD OF** pode ser definido por tabela ou em uma **VIEW** por cada comando **INSERT**, **UPDATE** ou **DELETE**. Apesar deste aspecto, é possível definir **VIEWS** em outras **VIEWS**, cada uma delas possuindo seu próprio **TRIGGER INSTEAD OF**.

Os **TRIGGERS INSTEAD OF** não podem ser especificados para **TRIGGERS DDL** e utilizados com **VIEWS** aptas a serem utilizadas e que utilizam o **WITH CHECK OPTION**, pois, caso isso aconteça, o SQL Server gera um erro. A fim de evitar problemas com os **TRIGGERS INSTEAD OF**, o usuário deve utilizar a opção **ALTER VIEW** antes de defini-lo.

Quando trabalhamos com este tipo de **TRIGGER**, não podemos utilizar os comandos **DELETE** e **UPDATE** em tabelas que possuem um relacionamento referencial que determina ações em cascata **ON DELETE** e **ON UPDATE**, respectivamente. Vale destacar que, no mínimo, uma das seguintes opções é determinar quais comandos de alteração de dados terão a finalidade de ativar um **TRIGGER DML** no momento em que ele é utilizado com uma tabela ou com uma **VIEW**. Essas opções podem não apenas serem utilizadas separadamente como em conjunto.

O exemplo descrito a seguir não permite realizar a exclusão de um registro da tabela:

```
CREATE TRIGGER T_NAO_EXCLUIR ON MINHA_TABELA INSTEAD OF DELETE AS
    RAISERROR ('NA NA NI NA NÃO...NADA FEITO!', 16, 1)
GO
```

Exercícios

1. Que são **TRIGGERS** e onde podem ser utilizados?

Resposta:

2. Quais são as diferenças entre **CONSTRAINTS** e **TRIGGERS**?

Resposta:

3. Que são tabelas criadas em tempo de execução? Qual a sua função?

Resposta:

4. Que são **TRIGGERS** aninhados?

Resposta:

5. Qual a função do comando **INSTEAD OF**?

Resposta:

6. Que comandos não podem ser executados com **TRIGGERS**?
 - a. CREATE(todos), DROP(todos), LOAD, INSERT
 - b. REVOKE, ALTER TABLE, TRUNCATE TABLE, DELETE
 - c. RECONFIGURE,SELECT INTO, UPDATE STATISTICS, INSERT
 - d. GRANT, ALTER TABLE, INSERT, SELECT INTO
 - e. RESTORE LOG, ALTER DATABASE, LOAD, GRANT
7. Sobre os **TRIGGERS** aninhados, é correto afirmar que:
 - a. Não permitem controlar se é possível utilizar um **TRIGGER** de **AFTER** em cascata.
 - b. Podemos aninhá-los em infinitos níveis.
 - c. Quando o aninhamento está habilitado, um **TRIGGER** que realiza alterações sobre uma tabela pode ativar um segundo **TRIGGER**, o qual, por sua vez, pode ativar um terceiro e assim sucessivamente.
 - d. Visto que os **TRIGGERS** fazem parte de uma transação, caso haja uma falha em qualquer um dos níveis de aninhamento dos **TRIGGERS**, a transação inteira será parcialmente desfeita mas nenhuma das alterações realizadas sobre os dados será cancelada.
 - e. Seja qual for o valor de configuração, os **TRIGGERS INSTEAD OF** não podem ser aninhados.
8. Quais alternativas são características dos **TRIGGERS**?
 - a. A criação, alteração e exclusão de um **TRIGGER** são tarefas que podem ser realizadas apenas pelo proprietário da tabela e essa permissão não pode ser transferida.
 - b. Os **TRIGGERS** podem ser criados em **VIEWS** ou em tabelas temporárias.
 - c. Os **TRIGGERS** retornam valores.
 - d. **TRIGGERS** podem ser criados apenas no banco de dados atual, mas podem fazer referências aos objetos que se encontram fora desse banco de dados.
 - e. Quando trabalhamos com tabelas que possuem chaves estrangeiras com uma ação **DELETE/UPDATE** em cascata, podemos definir os **TRIGGERS** dos comandos **INSTEAD OF DELETE/UPDATE**.

9. Escolha a alternativa que apresenta a palavra que deve ser inserida no espaço em branco da frase a seguir: Os **TRIGGERS** _____ não podem ser utilizados com **VIEWS** que podem ser atualizadas e que utilizam **WITH CHECK OPTION**, pois, caso isso aconteça, o SQL Server gerará um erro.
- a. Aninhados
 - b. de alteração
 - c. de exclusão
 - d. de inclusão
 - e. INSTEAD OF
10. Que comando não é capaz de realizar a execução de um **TRIGGER** devido ao fato de não estar logado?
- a. WRITETEXT
 - b. TRUNCATE TABLE
 - c. SET NOCOUNT
 - d. CREATE TRIGGER
 - e. INSTEAD OF

Laboratório

1. Crie um **TRIGGER** de inclusão na tabela **FUNCIONARIO**, que insira uma linha na tabela **PREMIO** cada vez que um funcionário do sexo feminino for inserido na tabela. O valor do prêmio deverá ser de 200.00.

Resposta:

2. Crie um **TRIGGER** sobre a tabela **ITENS**. A cada item inserido, o **TRIGGER** deverá calcular o valor que deve ser pago neste item e atualizar o total do pedido na tabela **PEDIDO**.

Resposta:

Capítulo 18

Concorrência

Em um sistema tipo cliente/servidor, é possível a execução simultânea e concorrente de diversas transações em um banco de dados. Nesse sistema, o processador pode ser compartilhado pelas diversas transações. Tudo isso reflete o que é chamado de multiprogramação, um dos conceitos mais importantes do sistema cliente/servidor.

A multiprogramação permite que as transações possam compartilhar um mesmo processador e oferecer os seguintes benefícios:

- Maior quantidade de trabalho em um intervalo de tempo específico, ou seja, maior quantidade de transações throughput.
- Menor tempo de resposta das transações interativas nas quais o usuário aguarda a resposta.
- O processador é utilizado de maneira otimizada durante as transações.

Em virtude da multiprogramação, as transações podem ser executadas de forma concorrente em um banco de dados. Por causa dessa concorrência, é necessário haver um controle por parte do sistema para com as interações existentes entre as transações concorrentes. Para tal, o sistema possui um mecanismo chamado **LOCK**, ou esquema de controle de concorrência.

LOCKS

O **LOCK** é um mecanismo desenvolvido para evitar a ocorrência de problemas de manipulação concorrentes das informações. Funciona como um bloqueio no acesso aos recursos dentro de um sistema cliente/servidor.

Por conta dos **LOCKS**, um mesmo dado não pode ser utilizado simultaneamente pelos usuários. Assim, um usuário não terá a capacidade de ler e alterar dados que estão sendo processados por um outro usuário, ou seja, não haverá conflitos de **UPDATE**.

Tipos de LOCKs

SHARED (S) LOCKs

São aplicados em operações que são apenas para leitura (alteração e mudança de dados não são permitidas), como a utilização do comando **SELECT**, e permitem que duas transações possam utilizar o mesmo recurso.

Duas transações podem ter um **SHARED LOCK** em um mesmo recurso, mesmo que uma das transações ainda não tenha sido finalizada.

Enquanto todas as linhas que irão satisfazer a query não forem enviadas para o cliente, os **SHARED LOCKs** serão mantidos.

UPDATE (U) LOCKs

Compatíveis com **SHARED LOCKs**, porém diferem deles, os **UPDATE LOCKs** são utilizados em recursos que podem ser modificados.

O **UPDATE LOCK** é adotado pelo SQL Server na alteração de páginas de dados. Quando isso é feito, esse modo de **LOCK** é promovido para o modo **EXCLUSIVE LOCK**, na página alterada.

Obtidos no início de uma atualização, durante a leitura das páginas de dados, os **UPDATE LOCKs** têm a propriedade de evitar um tipo comum de **DEADLOCK**, cujo processo é descrito a seguir:

1. Duas transações fazem a leitura simultânea de um registro.
2. No recurso, que pode ser uma página ou linha, as transações adquirem um **SHARED LOCK** e tentam alterar a linha ao mesmo tempo.
3. Uma das transações tenta converter seu **SHARED LOCK** para **EXCLUSIVE LOCK**.
4. Pelo fato de **EXCLUSIVE LOCK** de uma transação não ser compatível com o **SHARED LOCK** de outra transação, ocorre um **LOCK WAIT**, ou espera de conversão.
5. Da mesma forma, a outra transação tenta converter seu **SHARED LOCK** para **EXCLUSIVE LOCK**.
6. Ocorre um **DEADLOCK**, ou seja, ambas as transações estão aguardando que cada uma libere seus **SHARED LOCKs**.

A fim de evitar o **DEADLOCK** descrito, é necessário que uma única transação obtenha o **UPDATE LOCK** em um recurso, de maneira que, caso um recurso seja alterado por uma transação, tenhamos a conversão do **UPDATE LOCK** para **EXCLUSIVE LOCK**. Caso não haja alteração, o **LOCK** é convertido para **SHARED LOCK**.

EXCLUSIVE (X) LOCKs

Este modo de **LOCK** evita que duas transações possam alterar simultaneamente um mesmo recurso. Os **EXCLUSIVE LOCKs** são utilizados em operações para alterar dados, tais como **INSERT**, **UPDATE** ou **DELETE**.

INTENT (I) LOCKs

Os **INTENT LOCKs** têm a finalidade de aplicar uma hierarquia de **LOCK**, ou **LOCK** hierárquico, e são usados internamente pelo SQL Server para diminuir a ocorrência de conflitos de **LOCK**.

Os **INTENT LOCKs** têm a finalidade de indicar a intenção do SQL Server em obter um tipo de **LOCK INTENT** ou **SHARED** em um determinado recurso.

Quando aplicamos um **INTENT LOCK** no nível de uma tabela que já possui um **EXCLUSIVE LOCK** para uma linha ou página, evitamos que uma outra transação obtenha um **EXCLUSIVE LOCK** nessa tabela.

No caso de aplicarmos um **SHARED INTENT LOCK** em uma tabela, estamos informando que a intenção da transação é aplicar **SHARED LOCKs** em linhas ou páginas dessa tabela.

Uma das vantagens dos **INTENT LOCK** é poupar o SQL Server de analisar todos os **LOCKs** nas linhas ou páginas de uma tabela com o objetivo de definir se transação poderá alocar a tabela inteira. Isso traz um ganho de performance, já que o SQL Server precisa examinar apenas os **INTENT LOCKs** na tabela.

EXTENT LOCKs

Contidos em grupos de páginas do banco de dados no momento em que essas páginas estão sendo liberadas ou alocadas, os **EXTENT LOCKs** são adquiridos nas seguintes situações:

- Durante a execução de um comando **DROP** ou **CREATE**.
- Quando novas páginas de dados ou de índices são requeridas por um comando **INSERT**.

O SQL Server possuir **LOCKs** que bloqueiam outros processos que requerem um **LOCK**. Esses **LOCKs** bloqueadores são denominados **BLOCKING LOCKS**.

Devido ao **BLOCKING LOCK**, um processo poderá ter continuidade logo após o término do processo que está provocando o bloqueio.

SCHEMA (SCH) LOCKs

Estes **LOCKs** são utilizados com a finalidade de evitar que a tabela ou índice assim como seu esquema, seja destruído ao ser referenciado por uma outra sessão.

Os **SCHEMA LOCKs** podem ser de dois tipos: **SCHEMA STABILITY (SCH-S)** e **SCHEMA MODIFICATION (SCH-M)**. O primeiro tipo evita a destruição de um recurso, enquanto o segundo evita que outra seção possa fazer referência a um recurso em alteração.

Granularidade

No SQL Server podemos fazer com que uma transação possa realizar um **LOCK** de vários tipos de recursos. Isso é possível por meio da alocação multigranular do Database Engine, que aloca os recursos de maneira automática em um nível que seja apropriado para a tarefa, o que acaba diminuindo o custo de **LOCK**.

Fazer o **LOCK** em uma granularidade maior ou menor oferece vantagens e desvantagens, como explicado a seguir:

Alocar em menor granularidade: Maximiza a concorrência. As linhas são um exemplo de baixa granularidade. Porém, o **LOCK** apresenta uma maior overhead, já que, quanto mais linhas são alocadas, maior é a quantidade de **LOCKs** a serem mantidos.

Alocar em maior granularidade: Tabelas são exemplos de alta granularidade. Alocar uma tabela inteira acaba fazendo com que outras transações tenham acesso restrito a qualquer parte da tabela, fato que aumenta os custos no que se refere à concorrência. Por outro lado, o overhead é baixo, devido ao pequeno número de **LOCKs** mantidos.

O SQL Server pode alocar os seguintes recursos:

- **RID**: Trata-se de um identificador de linhas. Este recurso tem a finalidade de fazer o **LOCK** individual de uma única linha em uma tabela.
- **PAGE**: Este recurso é uma página de 8Kbytes de dados ou de índice.
- **EXTENT**: Este recurso compreende um grupo contíguo de páginas de dados ou de índices.
- **TABLE**: Este recurso apresenta uma tabela inteira, junto de seus dados e índices.
- **KEY-RANGE**: Este recurso tem a finalidade de alocar ranges entre registros em uma tabela.
- **DB**: Este recurso representa um banco de dados inteiro. DB impede que um banco de dados seja removido e deve ser utilizado sempre que houver uma conexão no banco de dados.

Problemas de Concorrência Impedidos Pelos LOCKs

LOST UPDATE

Vamos supor que as mesmas informações fossem modificadas por dois usuário diferentes. Não fossem os **LOCKs**, apenas a última alteração seria registrada no banco de dados, ou seja, as alterações de uma transação que fossem sobreescritas por alterações de uma outra transação seriam perdidas.

UNCOMMITED DEPENDENCY

O problema de leitura suja seria provocado pelo fato de uma transação ler os dados de uma outra transação que ainda não foi confirmada. Diante dessa situação, ao invés de **COMMIT TRANSACTION**, a transação não confirmada poderia executar um **ROLLBACK TRANSACTION**.

INCONSISTENT ANALYSIS (NONREPEATABLE READ)

A análise de inconsistência ocorre da seguinte maneira:

1. Uma segunda transação acessa a mesma linha diversas vezes, sendo que, em cada acesso, faz a leitura de dados diferentes. Trata-se de um processo semelhante ao **DIRTY READ**, em que uma transação modifica os dados que estão sendo lidos por uma segunda transação. É importante considerar, no entanto, que na análise de inconsistência, a transação que fez alteração é a responsável por liberar os dados que foram lidos pela segunda transação.
2. Entre duas ou mais leituras, uma outra transação altera as informações da linha. Daí o termo **NONREPEATABLE READ**, pois a leitura original nunca será repetida. Há, ainda, uma inconsistência porque cada leitura produzirá valores diferentes.

PHANTOMS

Este problema poderia ocorrer caso as transações não fossem separadas umas das outras.

Vamos supor a seguinte situação: iremos alterar de uma única vez todos os registros de tabelas existentes em uma determinada região. Ao mesmo tempo em que essa alteração é

feita, um novo registro é acrescentado à região por outra transação. Assim, da próxima vez que a primeira transação fizer a leitura dos registros, haverá um registro a mais na leitura.

Customizando o LOCK

É possível customizar o **LOCK** por meio de várias opções, vejamos:

- **ROWLOCK**: Obriga a utilização de um **LOCK** de linha, que é mantido pelo SQL Server até o fim do comando. Porém, será mantido até o final da transação caso especifiquemos **HOLDLOCK**.
- **PAGLOCK**: Obriga a utilização de um **SHARED LOCK** no nível de página. Também é mantido até o fim do comando a não ser que seja especificado **HOLDLOCK**.
- **TABLOCK**: O SQL Server obriga a utilização de um **SHARED LOCK** na tabela. Os usuários podem ver os dados mas não podem alterá-los.
- **UPDLOCK**: Utiliza um **SHARED LOCK**, mas sim do **UPDATE LOCK** no nível de página durante a leitura dos dados da tabela. Quando utilizamos um **UPDATE LOCK**, outras transações tem acesso a leitura mas não a escrita.
- **TABLOCKX**: Utiliza um **EXCLUSIVE LOCK** na tabela até o fim do comando. Impede leitura e escrita. Veja o modo de utilização:

```
BEGIN TRANSACTION  
GO  
  
UPDATE TELEFONES WITH (TABLOCKX)  
    SET NOME = 'JOAOZINHO' WHERE CODIGO = 2  
GO  
  
COMMIT TRANSACTION  
GO
```

- **NOLOCK**: Quando aplicada, permite a existência, na leitura, de transações não confirmadas ou um conjunto de páginas desfeitas e **DIRTY READS**.
- **READ PAST**: Uma tabela com dados bloqueados pode ser lida, sendo que teremos a exibição apenas dos dados que não estão bloqueados.

Customizando LOCKs na Seção

Em uma seção, utilizamos opções de nível de isolamento específicas para fazer a configuração do **LOCK**. Para configurar um nível de isolamento a ser aplicado na seção, utilizamos **TRANSACTION ISOLATION LEVEL (TIL)**.

A configuração do **TIL** possibilita determinar qual será o comportamento padrão dos **LOCKs** dos comandos existentes em uma seção.

Por meio das especificações de **LOCK**, podemos sobrepor um **LOCK** de seção de um determinado comando. Já para especificar um **TIL** de um comando, podemos utilizar o comando **DBCC USEROPTIONS**, como mostra o exemplo:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  
DBCC USEROPTIONS  
GO
```

Os níveis de isolamento são:

- **READ COMMITTED**: Este argumento, que é o padrão, faz com que o SQL Server utilize **SHARED LOCKs** em operações de leitura. Não aceita **DIRTY READS**.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

- **READ UNCOMMITTED**: Exerce a mesma função do **NOLOCK**, ou seja, o SQL Server é obrigado a não gerar **SHARED LOCKs**. Aceita **DIRTY READS**.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

- **REPEATABLE READ**: Esta opção evita que os dados acessados em uma query sejam alterados por outros usuários, por meio da utilização de **LOCKs** nos dados. **NONREPEATABLE READs** e **DIRTY READs** não são permitidas. Pode-se acrescentar novas linhas ao conjunto, pois serão consideradas na próxima leitura. Deve ser usado somente em extrema necessidade. Pois o nível de concorrência é menor do que o nível de isolamento padrão.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

- **SERIALIZABLE**: Tido como o mais restritivo dos **LOCKs**, o **SERIALIZABLE** proíbe a alteração ou inserção de novas linhas que apresentam o mesmo critério da cláusula **WHERE** da transação. Desta forma, impede a concorrência de fantasmas. É aconselhável utilizar o nível de isolamento **SERIALIZABLE** apenas quando necessário.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

LOCK Dinâmico

O SQL Server dispõe de um meio de definir automaticamente qual o tipo de **LOCK** é o mais adequado para as diferentes queries executadas. Isso é chamado **LOCK** dinâmico.

A escolha automática dos tipos de **LOCK**, além de poupar o trabalho do **DBA** em ajustá-los manualmente, concede aos desenvolvedores a chance de voltar seus esforços unicamente para a aplicação. Além disso, pelo fato de o SQL Server escolher automaticamente o tipo de **LOCK** para cada tarefa, há uma diminuição do overhead do sistema.

TIMEOUT

Como visto anteriormente, uma transação que solicita um lock, mas não o obtém pelo fato de uma outra transação já possuir um **LOCK** conflitante no recurso, irá aguardar um determinado tempo até que consiga realizar o **LOCK** no recurso desejado e, desta forma, possa completar o processo. A esse tempo de espera damos o nome de **TIMEOUT**. É possível configurá-lo por meio do seguinte comando: **SET LOCK_TIMEOUT [tempo_milisegundos]**

A opção **SET LOCK_TIMEOUT** permite configurar o tempo máximo que um comando irá aguardar até que o recurso seja desbloqueado para a realização do **LOCK**. Caso especifiquemos o valor de -1, o SQL Server entenderá que não há **TIMEOUT**.

Podemos conferir o tempo de **TIMEOUT** da seção pelo comando:

```
SELECT @@LOCK_TIMEOUT
```

Exercícios

1. Quais são as vantagens da multiprogramação?

Resposta:

2. Que é LOCK e quais os tipos existentes?

Resposta:

3. Quais as vantagens e desvantagens de fazer o LOCK em menor granularidade?

Resposta:

4. Cite alguns procedimentos que, se praticados, podem ajudar a diminuir a incidência de DEADLOCKS.

Resposta:

5. Que é TIMEOUT?

Resposta:

6. Que LOCK tem a finalidade de aplicar uma hierarquia de LOCK?
 - a. EXCLUSIVE (X) LOCKS
 - b. UPDATE (U) LOCKS
 - c. INTENT (I) LOCKS
 - d. SCHEMA (SCH) LOCKS
 - e. DEADLOCK
7. Em relação à NONREPEATABLE READ é correto afirmar:
 - a. Entre duas ou mais leituras, uma outra transação altera as informações da linha
 - b. Um transação faz a leitura de uma única linha por vez.
 - c. A primeira transação nunca repete uma leitura, já que a linha não é modificada pela outra transação
 - d. Em uma única leitura, uma outra transação altera as informações da linha.
 - e. A primeira transação sempre repete uma leitura, já que a linha é modificada pela outra transação as cada leitura.
8. Qual das alternativas a seguir devemos utilizar para configurar um nível de isolamento a ser aplicado na seção.
 - a. LOCK ISOLATION LEVEL
 - b. TRANSACT ISOLATION LEVEL
 - c. TRANSACTION ISOLATION LEVEL
 - d. ROLE ISOLATION LEVEL
 - e. NOLOCK

Capítulo 19

Distribuição de Dados

Há situações em que se faz necessária a distribuição de dados entre regiões geograficamente diferentes. Para tanto, é preciso implementar uma solução que torne o ambiente mais gerenciável, visto que alguns ambientes, como o OLTP, requerem total consistência de dados a todo o momento, ao passo que outros ambientes, como o de suporte a decisões, não exigem a freqüente atualização dos dados.

Acessando Dados de um SQL Server Remoto

Para acessar os dados de um servidor remoto, utilizando a função **OPENROWSET()** sem a necessidade de ligar os servidores, conforme demonstra o exemplo a seguir:

```
SELECT * FROM OPENROWSET (
    'SQLOLEDB',
    'SERVER',
    'SA',
    '',
    'SELECT * FROM DBO.TABELA_REMOTA')
GO
```

Daqui tiramos que a sintaxe básica:

```
[comando] [parametros_comando] OPENROWSET ( '[driver_conexao]', 
'[nome_servidor]', '[usuario]', '[senha]', '[comando]' )
```

Conectando-se a um SQL Server Remoto

A fim de que possamos acessar os dados presentes no servidor remoto com freqüência, basta realizar um link entre o servidor remoto e o local e, então, fazer as queries utilizando o nome totalmente qualificado do objeto, o qual é formado pelos seguintes itens:

servidor.banco_dados.schema.objeto

Observemos:

```
EXEC SP_ADDLINKEDSERVER @SERVER = 'SERVER', @SRVPRODUCT = 'SQL SERVER'
GO
```

A Stored Procedure de Sistema, **SP_ADDLINKEDSERVER**, permite determinar o nome do servidor conectado, que diz respeito a um servidor virtual definido para o SQL Server com todos os dados que se fazem necessários para realizar o acesso à origem de dados **OLEDB**.

Nota: A **STORE PROCEDURE SP_ADDLINKEDSRVLOGIN** permite a realizar o mapeamento dos logins referentes ao servidor local para os logins presentes no servidor remoto.

O nome do servidor conectado pode ser utilizado com a finalidade de fazer uma referência às tabelas remotas de várias maneiras. Uma das formas possíveis é utilizar esse nome como um parâmetro de entrada para a função **OPENQUERY()**, sua finalidade é enviar um comando para executar no provedor **OLEDB**. Com isso, o rowset obtido como resultado pode ser utilizado no comando Transact-SQL como referência a uma **VIEW** ou a uma tabela.

Conectando-se a Uma Origem OLEDB

Para que possamos fazer conexão com uma origem de dados **OLEDB**, devemos utilizar a sintaxe descrita a seguir:

```
EXEC SP_ADDLINKEDSERVER  
    @SERVER = 'SERVIDOR',  
    @SRVPRODUCT = 'ORACLE',  
    @PROVIDER = 'MSAPKE',  
    @DATASRC = 'ORACLEDDB'  
GO
```

Em que:

- **SP_ADDLINKEDSERVER:** Stored Procedure de Sistema capaz de definir o provedor OLEDB e especificar, no computador local, um SQL Server remoto.
- **@SERVER:** Nome do servidor ao qual se deseja conectar.
- **@SRVPRODUCT:** Produto da origem de dados **OLEDB**.
- **@PROVIDER:** Nome do provedor **OLEDB** correspondente à origem de dados.
- **@DATASRC:** Refere-se ao nome da origem de dados da forma como ele é interpretado pelo provedor **OLEDB**.

Segurança

Para estabelecer a segurança entre os servidores local e remoto da forma adequada, devemos considerar alguns fatores importantes, a destacar:

- No momento em que um usuário efetua seu logon no servidor local e executa uma query distribuída, o servidor realiza um logon no servidor remoto em nome do usuário.
- Caso as informações de logon do usuário (usuário e senha) existam nos dois servidores, o SQL Server pode conectar-se ao servidor remoto utilizando as credenciais do usuário atual.

Conforme mencionado anteriormente, a Stored Procedure de Sistema **SP_ADDLINKEDSRVLOGIN** permite realizar o mapeamento dos IDs de login e das senhas entre o servidor local e remoto. Portanto, esta **STORED PROCEDURE** dispensa a criação de um ID de login e uma senha para cada um dos usuários nos dois servidores.

Observe a sintaxe descrita abaixo. Ela permite estabelecer uma conexão segura entre os dois servidores:

```
EXEC SP_ADDLINKEDSRVLOGIN
    @RMTSRVNAME = 'SERVIDOR',
    @USESELF = 'FALSE',
    @LOCALLOGIN = 'CLASSROOM\ALUNO',
    @RMTUSER = 'XXXX',
    @RMTPASSWORD = 'XXXX'
GO
```

Em que:

- **@RMTSRVNAME:** Nome do servidor conectado para o qual está sendo realizado o mapeamento do login.
- **@USESELF:** Determina se o login do SQL Server será realizado com suas próprias credenciais.
- **@LOCALLOGIN:** Trata-se de um ID de login do servidor local a ser utilizado em caráter opcional. Vale destacar que, a fim de que possa ser utilizado, o valor definido no parâmetro deve existir no servidor local.
- **@RMTUSER:** Nome do usuário.
- **@RMTPASSWORD:** Senha do usuário.

Acessando Dados do Servidor Conectado Com o Nome Totalmente Qualificado

O acesso aos dados presentes em um servidor conectado pode ser feito a partir de um nome totalmente qualificado. Para tanto, basta utilizar como base os exemplos descritos:

```
SELECT * INTO #MINHA_TABELA_TEMPORARIA FROM  
[nome_servidor].[database].dbo.[nome_tabela]  
GO  
  
SELECT * FROM [nome_servidor].MASTER.DBO.SYSSERVERS  
GO
```

Acessando Dados do Servidor Conectado Com a Função OPENQUERY()

Outra maneira de acessar os dados presentes em um servidor conectado é utilizar a função **OPENQUERY()**, conforme descrito abaixo:

```
SELECT * FROM  
OPENQUERY([nome_servidor], 'SELECT * FROM [database].dbo.[nome_tabela]')  
GO
```

Podemos referenciar a função **OPENQUERY()** na cláusula **FROM** de uma query de duas formas distintas: como o nome de uma tabela ou como sendo a tabela de destino dos comandos **INSERT**, **DELETE** ou **UPDATE**. Isso depende dos recursos apresentados pelo provedor **OLEDB**.

Comando e Ações Proibidos no Servidor Conectado

Nas situações em que o acesso a um servidor conectado é feita a partir do servidor local por meio de queries distribuídas, não podemos realizar algumas ações ou executar alguns comandos, os quais serão descritos a seguir:

As seguintes cláusulas não podem ser utilizadas: **ORDER BY**, **READTEXT**, **WRITETEXT** e **UPDATETEXT**.

Os seguintes comandos não podem ser executados: **SELECT INTO**, **CREATE**, **ALTER** e **DROP**.

O comando **SELECT INTO**, embora não possa ser utilizado no servidor conectado devido ao fato de criar tabelas, pode ser utilizado no servidor local tendo como origem dos dados o servidor conectado.

Opções Para Configurar o Servidor Conectado

A Stored Procedure de Sistema **SP_SERVEROPTION** permite determinar algumas opções para configurar o servidor local. Observe:

```
EXEC SP_SERVEROPTION  
    @SERVER = 'SERVIDOR',  
    @OPTNAME = 'OPCAO',  
    @OPTVALUE = 'VALOR'  
GO
```

Um exemplo de opção configurável é este:

```
EXEC SP_SERVEROPTION  
    @SERVER = 'SERVER',  
    @OPTNAME = 'COLLATION COMPATIBLE',  
    @OPTVALUE = 'TRUE'  
GO
```

COLLATION COMPATIBLE, se configurado para **TRUE**, faz com que o SQL Server assuma que todas as colunas e **CHARACTERS SETS** presentes no servidor remoto são compatíveis com o servidor local. (Idioma)

Modificando Dados em Um Servidor Remoto

O comando **BEGIN DISTRIBUTED TRANSACTION** permite executar uma transação distribuída a fim de que possamos alterar os dados presentes em um servidor remoto. A instância do SQL Server Database Engine que executa este comando é responsável por dar origem à transação e controlar o encerramento da mesma.

Essa instância solicita que o controle do encerramento de transações distribuídas seja feito pelo MS DTC por todas as instâncias que participam do processo nas situações em que um comando **COMMIT TRANSACTION** ou **ROLLBACK TRANSACTION** é emitido de forma subsequente. Devemos ter em mente que as transações distribuídas não são suportadas pelo isolamento de **SNAPSHOT** no nível de transação.

A principal forma de registrar instâncias remotas do database engine em uma transação distribuída ocorre nas situações em que uma query distribuída que faz referência a um servidor conectado é executada por uma sessão que já foi registrada em uma transação distribuída.

Um servidor remoto apenas pode ser registrado em uma transação caso ele seja uma chamada da Stored Procedure Remota ou um destino de uma query distribuída. As sessões que estão envolvidas nas transações distribuídas Transact-SQL não obtêm um objeto de transação a ser passado para outra sessão a fim de que ela possa ser registrada na transação distribuída de forma explícita.

Caso a origem de dados **OLEDB** de destino seja capaz de suportar **ITransactionLocal**, a transação passa a ser distribuída de forma automática no momento em que uma query distribuída é executada na transação local. Caso não haja esse suporte ao **ITransactionLocal**,

podem ser executadas na query distribuída apenas as operações somente-leitura. Com isso, uma chamada à stored procedure remota fazendo referência a um servidor remoto é realizada pela sessão que já se encontra registrada na transação distribuída.

Para controlar se as chamadas a uma stored procedure remota realizadas em uma transação local fazem com que esta se torne uma transação distribuída cujo gerenciamento é realizado pelo MS DTC, contamos com a opção **SP_CONFIGURE REMOTE PROC TRANS**, a qual estabelece uma instância padrão que pode ser sobreescrita por meio de **REMOTE_PROC_TRANSACTIONS** da opção SET no nível da conexão.

Caso a opção **SP_CONFIGURE REMOTE PROC TRANS** esteja configurada com o valor **ON**, além de as transações locais serem promovidas a transações distribuídas por conta da chamada a uma stored procedure remota, tais chamadas feitas locais são protegidas de forma automática como sendo parte das transações distribuídas. Para tanto, torna-se desnecessário reescrever aplicações a fim de emitir o comando **BEGIN DISTRIBUTED TRANSACTION**.

Como as Transações São Distribuídas

São realizadas da seguinte forma:

1. A aplicação chama o gerenciador de transações com o comando **BEGIN DISTRIBUTED TRANSACTION** a fim de iniciar uma transação.
2. Em seguida, o comando em questão cria um objeto capaz de representar a transação.
3. Então, a aplicação envia ao gerenciador de recursos o objeto da transação em conjunto aos seus requerimentos.
4. O gerenciador de recursos, por sua vez, executa o requerimento. Além disso, este gerenciador coloca em uma listagem os endereços dos servidores que participam do processo.
5. A transação chama o MS DTC **COMMIT TRANSACTION** assim que a aplicação é encerrada com êxito.
6. A partir do protocolo **TWO FASE COMMIT** e da listagem contendo os endereços dos servidores que participam do processo, o MS DTC assegura que todos os SQL Servers realizem o **COMMIT** ou o **ROLLBACK TRANSACTION**.

O protocolo **TWO FASE COMMIT** assegura que todos os gerenciadores de recursos confirmem a transação ou abortem a mesma. O processo é composto basicamente por duas fases: na primeira, o MS DTC define se cada um dos gerenciadores de recursos está pronto para realizar o **COMMIT**. Caso todos eles estejam, começa a segunda fase, na qual o MS DTC realiza a transação da mensagem de **COMMIT**.

Vale destacar que, devido a falhas de sistema ou de comunicação, é possível que haja um atraso na entrega das mensagens de **COMMIT** ou de **ABORT**. No decorrer do período de atraso, todos os **LOCKS** de objetos que tenham sido omitidos são mantidos pelo gerenciador de recursos.

A transação será abortada caso haja falha em qualquer uma de suas partes. Quando isso acontece, o gerenciador não responde ao requerimento de preparação para o **COMMIT** ou responde simplesmente **NO**.

Participando de Transações Distribuídas

O comando **BEGIN DISTRIBUTED TRANSACTION** é utilizado para iniciar a participação do MS DTC e do SQL Server em uma transação distribuída. Como isso, as Remote Stored Procedures são utilizadas pelo SQL Server com a finalidade de determinar o trabalho em diferentes servidores. É essencial ter em mente que o SQL Server precisa ser avisado para que seja possível participar de uma transação distribuída.

Considerações

As transações distribuídas não são capazes de suportar o **SAVE POINT**. No entanto, podemos utilizar dentro delas as transações comuns. Além disso, o comando **BEGIN DISTRIBUTED TRANSACTION** não pode ser utilizado de forma aninhada, e quando utilizamos um **ROLLBACK TRANSACTION** toda a transação é desfeita.

MS DTC (Microsoft Distributed Transaction Coordinator)

Este serviço oferecido pelo SQL Server assegura a integridade das transações a partir do gerenciamento de alterações feito através de dois ou mais servidores. Dessa forma, o MS DTC oferece alta consistência entre os servidores na medida em que garante que as alterações realizadas sobre os dados serão feitas em todos os servidores de forma simultânea.

Quando trabalhamos em um ambiente homogêneo contendo apenas o SQL Server, o MS DTC é invocado de uma das seguintes formas: utilizando o comando **BEGIN DISTRIBUTED TRANSACTION** ou utilizando as interfaces de programação **ODBC** ou **OLEDB** em um cliente SQL Server.

Alguns objetos **COM** são expostos pelo MS DTC. Eles têm a finalidade de permitir que os clientes participem de transações coordenadas através de diversas conexões para uma variedade de armazenamento de dados.

A interface **ITransactionDispenser** do MS DTC é utilizada pelo usuário do provedor SQL Native Client OLEDB a fim de dar início a um transação. Com isso, o membro **BeginTransaction** (tudo junto mesmo) dessa interface retorna uma referência sobre o objeto de uma transação distribuída, o qual requer a utilização de **JoinTransaction** a fim de que essa referência seja passada ao provedor SQL Native Client **OLEDB**. Os usuário desse provedor podem utilizar o

método `ITransactionJoin::JoinTransaction` com a finalidade de participar de uma transação distribuída cuja coordenação é feita pelo MS DTC.

A fim de que o usuário possa receber notificações a respeito do status de transações assíncronas, ele deve realizar duas tarefas: implementar a interface `ITransactionOutcomeEvents` e conectá-la ao objeto de transação do MS DTC. Isso é importante porque o MS DTC suporta processos de **COMMIT** e de **ABORT** realizados de forma assíncrona sobre as transações distribuídas.

Exercícios

1. Como são fornecidas as informações necessárias para acessar os dados presentes em uma origem de dados **OLEDB**? (o nome do provedor **OLEDB** e as informações necessárias ao provedor para que ele seja capaz de encontrar a origem de dados)

Resposta:

2. Que fatores devem ser levados em consideração para estabelecer a segurança entre os servidores local e remoto de forma adequada?

Resposta:

3. Quais são as ações ou comandos que não podemos executar nas situações em que o acesso a um servidor conectado é feito a partir do servidor local por meio de queries distribuídas?

Resposta:

4. O que faz o comando **BEGIN DISTRIBUTED TRANSACTION**?

Resposta:

5. Como as transações são distribuídas?

Resposta:

6. Para que é utilizada a **STORED PROCEDURE SP_ADDLINKEDSERVER**?
 - a. Ela permite realizar o mapeamento dos logins referentes ao servidor local para os logins presentes no servidor conectado.
 - b. Permite realizar o mapeamento dos IDs de login e das senhas entre os servidores local e remoto.
 - c. Permite executar uma transação distribuída a fim de que possamos alterar os dados presentes em um servidor remoto.
 - d. Estabelece uma instância padrão que pode ser sobreescrita por meio de **REMOTE_PROC_TRANSACTIONS** da opção **SET** no nível da conexão.
 - e. Assegura que todos os gerenciadores de recursos confirmem a transação ou abortem a mesma.
7. Qual das alternativas apresenta a opção descrita no parágrafo a seguir? Trata-se de uma opção que afeta a execução das queries no servidor conectado. Caso o valor atribuído a esta opção seja true, o SQL Server assume que todas as colunas e **CHARACTER SETS** presentes no servidor remoto são compatíveis com o servidor local.
 - a. DATA ACCESS
 - b. SERIALIZABLE
 - c. COLLATION COMPATIBLE
 - d. READ UNCOMMITED
 - e. READ COMMITED
8. Como funciona o protocolo **TWO FASE COMMIT**?
 - a. O processo é composto basicamente por duas fases: Na primeira, o MS DTC define se cada transação está pronta para realizar o **COMMIT**. Caso todas elas estejam, começa a segunda fase, na qual o MS DTC realiza a transmissão da mensagem de **COMMIT**.
 - b. O processo é composto basicamente por duas fases: Na primeira, o Activity Monitor define se cada um dos gerenciadores de recursos está pronto para realizar o **COMMIT**. Caso todos eles estejam, começa a segunda fase, na qual o MS DTC realiza a transmissão da mensagem de **COMMIT**.
 - c. O processo é composto basicamente por duas fases: Na primeira, o SQL Server Agent define se cada um dos gerenciadores de recursos está pronto para realizar o **COMMIT**. Caso todos eles estejam, começa a segunda fase, na qual o MS DTC realiza a transmissão da mensagem de **COMMIT**.
 - d. O processo é composto basicamente por duas fases: Na primeira, o MS DTC define se cada um dos gerenciadores de recursos está pronto para realizar o **COMMIT**. Caso todos eles estejam, começa a segunda fase, na qual o MS DTC realiza a transmissão da mensagem de **COMMIT**.

- e. O processo é composto basicamente por duas fases: Na primeira, o MS DTC define se cada transação está pronta para realizar o **COMMIT**. Caso todos eles estejam, começa a segunda fase, na qual o MS DTC realiza **ROLLBACK** em caso de falha.
9. Sobre as transações distribuídas, é incorreto dizer que:
- a. As transações distribuídas não são capazes de suportar o **SAVE POINT**
 - b. Quando utilizamos um **ROLLBACK TRANSACTION**, toda a transação é desfeita
 - c. O comando **BEGIN DISTRIBUTED TRANSACTION** não pode ser utilizado de forma aninhada
 - d. Não podemos utilizar dentro delas as transações comuns
 - e. O comando **BEGIN DISTRIBUTED TRANSACTION** pode ser utilizado de forma aninhada
10. Qual das alternativas apresenta as palavras que complementam corretamente o parágrafo a seguir? Quando trabalhamos em um ambiente _____ contendo apenas o SQL Server, o _____ é invocado de um das seguintes formas: utilizando o comando _____ ou utilizando as interfaces de programação **ODBC** ou **OLEDB** em um cliente do SQL Server.
- a. heterogêneo / SQL Server Agent / **BEGIN DISTRIBUTED TRANSACTION**
 - b. homogêneo / MS DTC / **BEGIN TRANSACTION**
 - c. heterogêneo / Linked Server / **BEGIN DISTRIBUTED TRANSACTION**
 - d. homogêneo / Activity Monitor / **BEGIN TRANSACTION**
 - e. heterogêneo / MS DTC / **BEGIN DISTRIBUTED TRANSACTION**

Apêndice

Banco de Dados de Sistema

RESOURCE

Este banco de dados possui todos os objetos de sistema do SQL Server, os quais são visualizados de forma lógica no esquema **SYS** de todos os bancos de dados. O **RESOURCE**, que é um banco de dados somente leitura, não possui metadados ou dados dos usuários. Ele permite a rápida atualização de versões e o fácil rollback dos pacotes de serviços.

MASTER

O banco de dados **MASTER** é o responsável por efetuar os registros de todas as informações do nível do sistema. Sendo assim, ele registra informações para iniciar o SQL Server, contas de login, configurações referentes ao sistema e a existência de outros bancos de dados e de seus arquivos. Com isso, ele possui informações que permitem localizar os bancos de dados dos usuários. As stored procedures estendida e de sistema são criadas pelo setup dentro do **MASTER**.

TEMPDB

Este banco de dados é responsável por manter todos os tipos de armazenamentos temporários, inclusive o de tabelas de caráter temporário. Ele é considerado um recurso global, visto que armazena tabelas e stored procedures temporárias que podem ser acessadas por todos os usuários que se encontram conectados a instância do SQL Server.

MODEL

É utilizado como um modelo para criar todos os outros bancos de dados. Dessa forma, quando solicitamos a criação de um novo banco de dados, o SQL Server copia todos os objetos de **MODEL** para o novo banco de dados. Caso sejam feitas alterações em cima do banco de dados **MODEL**, todos os novos bancos de dados irão herdar essas alterações.

MSDB

Utilizado pelo SQL Server Agent com a finalidade de registrar operadores e programar a execução de **JOBs** e as configurações de replicação. O **MSDB** é utilizado para armazenar os dados não apenas do SQL Server Agent, mas também do SQL Server e SSMS. Um histórico completo de backups e restaurações on-line é mantido pelo SQL Server no banco de dados **MSDB**.

BACKUP

Talvez uma das tarefas mais importantes do DBA seja a criação de backups. Apenas um pequeno deslize no servidor de dados e todas as informações podem ir por água a baixo, a não ser que o DBA, previamente, tenha feito um backup de sua base de dados.

A realização do backup no SQL Server pode ser feita de três maneiras principais: por linha de comando através do SSMS, pelo passo-a-passo do SSMS ou pode meio de **JOBs** agendados. Veremos como fazer um backup a partir da linha de comando do SSMS. A sintaxe básica é a seguinte:

```
BACKUP DATABASE [nome_database] TO DISK 'X:\caminho_logico\nome_database.bak'  
GO
```

Isso irá criar um arquivo de backup no local especificado no disco. Para restaurar um backup, procedemos da forma a restaurar um database pelo passo-a-passo do SSMS. Só que a localidade dos arquivos será a localidade do arquivo de backup.

Stored Procedures Úteis

Aqui está listado algumas procedures para manutenção e configuração do SQL Server. Vejamos:

- SP_HELPDB [nome_database]: Trás as informações do database.
- SP_DETACH_DB [nome_database]: Desconecta um database do SQL Server.
- SP_ATTACH_DB '[nome_database]', '[primary_data_file]', '[log_data_file]' : Conecta um database ao SQL Server.
- SP_HELPFILEGROUP: Dentro do database mostra informações sobre o FILEGROUP do database.
- SP_HELPFILE: Dentro do database mostra informações dos arquivos do database.
- DBCC SHRINKFILE('[nome_database]', 0) : Executa o Shrink no database.
- SP_SPACEUSED: Dentro do database mostra o espaço usado pelos arquivos.
- SP_HELP [nome_tabela]: Retorna informações da tabela. Pode ser usando para obter informações de PROCEDUREs, VIEWs, FUNCTIONs e TRIGGERS.
- SP_SPACEUSED [nome_tabela]: Mostra o espaço ocupado pela tabela em disco.

- SP_RENAME '[nome_tabela].[nome_coluna]', '[novo_nome_coluna]': Renomeia uma coluna da tabela.
- SP_HELPINDEX [nome_tabela]: Trás informações sobre o índice da tabela.