

**SQL 2014 - Módulo I**

Guilherme FPM Caires  
392.280.768-28



# SQL 2014 – Módulo I



**IMPACTA**  
EDITORIA

## Créditos

Copyright © TechnoEdition Editora Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da TechnoEdition Editora Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

*"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."*

# SQL 2014 – Módulo I

### Coordenação Geral

Marcia M. Rosa

### Coordenação Editorial

Henrique Thomaz Bruscagin

### Atualização

Daniel Paulo Tamarosi Salvador

### Revisão Ortográfica e Gramatical

Marcos Cesar dos Santos Silva

### Diagramação

Shelida Letícia Lopes

**Edição nº 1 | Cód.: 1685/1**

Agosto/2014



Este material constitui uma nova obra e é uma derivação da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Out/2013: **SQL - Módulo I**

Autoria:

Carlos Magno Pratico de Souza

# Sumário

## Informações sobre o treinamento ..... 9

### Capítulo 1 - Introdução ao SQL Server 2014 ..... 11

1.1.	Banco de dados relacional .....	12
1.2.	Design do banco de dados .....	12
1.2.1.	Modelo descritivo .....	12
1.2.2.	Modelo conceitual .....	15
1.2.3.	Modelo lógico.....	16
1.2.4.	Modelo físico.....	17
1.3.	Arquitetura cliente / servidor .....	18
1.4.	As linguagens SQL e T-SQL .....	20
1.5.	SQL Server .....	21
1.5.1.	Componentes .....	21
1.5.2.	Objetos de banco de dados .....	22
1.5.2.1.	Tabelas .....	22
1.5.2.2.	Índices .....	22
1.5.2.3.	CONSTRAINT .....	22
1.5.2.4.	VIEW (Visão) .....	23
1.5.2.5.	PROCEDURE (Procedimento Armazenado) .....	23
1.5.2.6.	FUNCTION (Função) .....	23
1.5.2.7.	TRIGGER (Gatilho) .....	23
1.6.	Ferramentas de gerenciamento.....	24
1.7.	SQL Server Management Studio (SSMS).....	26
1.7.1.	Inicializando o SSMS .....	26
1.7.2.	Interface.....	28
1.7.3.	Executando um comando .....	32
1.7.4.	Salvando scripts .....	34
	Pontos principais .....	36
	<b>Teste seus conhecimentos</b> ..... 37	

### Capítulo 2 - Criando um banco de dados ..... 41

2.1.	Introdução.....	42
2.2.	CREATE DATABASE .....	42
2.3.	CREATE TABLE .....	44
2.4.	Tipos de dados .....	46
2.4.1.	Numéricos exatos.....	46
2.4.2.	Numéricos aproximados.....	48
2.4.3.	Data e hora .....	49
2.4.4.	Strings de caracteres ANSI .....	50
2.4.5.	Strings de caracteres Unicode.....	51
2.4.6.	Strings binárias .....	52
2.4.7.	Outros tipos de dados .....	53
2.5.	Campo de autonumeração (IDENTITY).....	54
2.6.	Constraints.....	54
2.6.1.	Nulabilidade .....	55

# SQL 2014 – Módulo I

2.6.2.	Tipos de constraints .....	55
2.6.2.1.	PRIMARY KEY (chave primária) .....	55
2.6.2.2.	UNIQUE .....	56
2.6.2.3.	CHECK.....	58
2.6.2.4.	DEFAULT .....	59
2.6.2.5.	FOREIGN KEY (chave estrangeira) .....	60
2.6.3.	Criando constraints .....	62
2.6.3.1.	Criando constraints com CREATE TABLE .....	62
2.6.3.2.	Criando constraints com ALTER TABLE.....	65
2.6.3.3.	Criando constraints graficamente .....	66
2.7.	Normalização de dados .....	76
2.7.1.	Regras de normalização .....	77
2.8.	Índices .....	84
2.8.1.	Criando índices .....	86
2.8.1.1.	Excluindo índices .....	87
	Pontos principais .....	88
	Teste seus conhecimentos.....	91
	Mãos à obra!.....	99

## Capítulo 3 - Inserção de dados..... 103

3.1.	Constantes .....	104
3.2.	Inserindo dados.....	107
3.2.1.	INSERT posicional.....	109
3.2.2.	INSERT declarativo.....	109
3.3.	Utilizando TOP em uma instrução INSERT .....	110
3.4.	OUTPUT.....	111
3.4.1.	OUTPUT em uma instrução INSERT .....	112
	Pontos principais .....	115
	Teste seus conhecimentos.....	117
	Mãos à obra!.....	123

## Capítulo 4 - Consultando dados..... 127

4.1.	Introdução.....	128
4.2.	SELECT .....	132
4.2.1.	Consultando todas as colunas .....	135
4.2.2.	Consultando colunas específicas .....	135
4.2.3.	Redefinindo os identificadores de coluna com uso de alias .....	138
4.3.	Ordenando dados.....	140
4.3.1.	Retornando linhas na ordem ascendente .....	140
4.3.2.	Retornando linhas na ordem descendente .....	140
4.3.3.	Ordenando por nome, alias ou posição.....	141
4.3.4.	ORDER BY com TOP .....	145
4.3.5.	ORDER BY com TOP WITH TIES .....	146
4.4.	Filtrando consultas .....	149
4.5.	Operadores relacionais .....	150
4.6.	Operadores lógicos .....	153
4.7.	Consultando intervalos com BETWEEN .....	155
4.8.	Consulta com base em caracteres.....	156
4.9.	Consultando valores pertencentes ou não a uma lista de elementos ...	159

# Sumário

4.10.	Lidando com valores nulos .....	160
4.11.	Substituindo valores nulos.....	162
4.11.1.	ISNULL.....	162
4.11.2.	COALESCE .....	163
4.12.	Manipulando campos do tipo datetime .....	164
4.13.	Alterando a configuração de idioma a partir do SSMS .....	176
	Pontos principais .....	180
	<b>Teste seus conhecimentos.....</b>	<b>181</b>
	<b>Mãos à obra!</b> .....	<b>189</b>

## **Capítulo 5 - Atualizando e excluindo dados ..... 195**

5.1.	Introdução.....	196
5.2.	UPDATE.....	196
5.2.1.	Alterando dados de uma coluna .....	199
5.2.2.	Alterando dados de diversas colunas.....	199
5.2.3.	Utilizando TOP em uma instrução UPDATE.....	200
5.3.	DELETE .....	201
5.3.1.	Excluindo todas as linhas de uma tabela.....	202
5.3.2.	Utilizando TOP em uma instrução DELETE.....	203
5.4.	OUTPUT para DELETE e UPDATE.....	204
5.5.	Transações .....	205
5.5.1.	Transações explícitas .....	206
	Pontos principais .....	208
	<b>Teste seus conhecimentos.....</b>	<b>209</b>
	<b>Mãos à obra!</b> .....	<b>213</b>

## **Capítulo 6 - Associando tabelas ..... 217**

6.1.	Introdução.....	218
6.2.	INNER JOIN .....	218
6.3.	OUTER JOIN .....	226
6.4.	CROSS JOIN .....	230
	Pontos principais .....	231
	<b>Teste seus conhecimentos.....</b>	<b>233</b>
	<b>Mãos à obra!</b> .....	<b>237</b>

## **Capítulo 7 - Consultas com subqueries ..... 241**

7.1.	Introdução.....	242
7.2.	Principais características das subqueries .....	242
7.3.	Subqueries introduzidas com IN e NOT IN .....	246
7.4.	Subqueries introduzidas com sinal de igualdade (=) .....	247
7.5.	Subqueries correlacionadas .....	247
7.5.1.	Subqueries correlacionadas com EXISTS.....	248
7.6.	Diferenças entre subqueries e associações .....	249
7.7.	Diferenças entre subqueries e tabelas temporárias .....	250
	Pontos principais .....	252
	<b>Teste seus conhecimentos.....</b>	<b>255</b>
	<b>Mãos à obra!</b> .....	<b>261</b>

<b>Capítulo 8 - Atualizando e excluindo dados em associações e subqueries.....</b>	<b>265</b>
8.1.        UPDATE com subqueries.....	266
8.2.        DELETE com subqueries.....	266
8.3.        UPDATE com JOIN.....	267
8.4.        DELETE com JOIN.....	267
Pontos principais .....	268
<b>Teste seus conhecimentos.....</b>	<b>269</b>
<b>Mãos à obra!.....</b>	<b>275</b>
<b>Capítulo 9 - Agrupando dados .....</b>	<b>279</b>
9.1.        Introdução.....	280
9.2.        Funções de agregação .....	280
9.2.1.    Tipos de função de agregação .....	281
9.3.        GROUP BY.....	284
9.3.1.    Utilizando ALL .....	288
9.3.2.    Utilizando HAVING .....	288
9.3.3.    Utilizando WITH ROLLUP .....	290
9.3.4.    Utilizando WITH CUBE .....	294
Pontos principais .....	296
<b>Teste seus conhecimentos.....</b>	<b>297</b>
<b>Mãos à obra!.....</b>	<b>301</b>
<b>Capítulo 10 - Comandos Adicionais .....</b>	<b>305</b>
10.1.       Funções de cadeia de caracteres.....	306
10.2.       Função CASE .....	311
10.3.       UNION .....	312
10.3.1.  Utilizando UNION ALL .....	312
10.4.       EXCEPT e INTERSECT .....	314
Pontos principais .....	318
<b>Teste seus conhecimentos.....</b>	<b>319</b>
<b>Mãos à obra!.....</b>	<b>323</b>

## Informações sobre o treinamento

Para que os alunos possam obter um bom aproveitamento do **curso SQL 2014 - Módulo I**, é imprescindível que eles tenham participado dos nossos cursos de Ambiente Windows e Introdução à Lógica de Programação, ou possuam conhecimentos equivalentes.



# Introdução ao SQL Server 2014

1

- ✓ Banco de dados relacional;
- ✓ Design do banco de dados;
- ✓ Arquitetura cliente / servidor;
- ✓ As linguagens SQL e T-SQL;
- ✓ SQL Server;
- ✓ Ferramentas de gerenciamento;
- ✓ SQL Server Management Studio (SSMS).



**IMPACTA**  
EDITORA

## 1.1. Banco de dados relacional

Um banco de dados é uma forma organizada de armazenar informações de modo a facilitar sua inserção, alteração, exclusão e recuperação.

Um banco de dados relacional é uma arquitetura na qual os dados são armazenados em tabelas retangulares, semelhantes a uma planilha. Na maioria das vezes, essas tabelas possuem uma informação chave que as relaciona.

## 1.2. Design do banco de dados

O design do banco de dados é fundamental para que o banco de dados possua um bom desempenho. Para isso, é importante entender os princípios que norteiam a boa construção de um banco de dados.

A construção do banco de dados passa por quatro etapas até a criação dos objetos dentro do banco de dados, a saber: modelos descritivo, conceitual, lógico e físico.

### 1.2.1. Modelo descritivo

O modelo descritivo de dados é um documento que indica a necessidade de construção de um banco de dados. Nesse modelo, o cenário é colocado de forma escrita, informando a necessidade de armazenamento de dados. Não existe uma forma padrão para escrever esse modelo, pois cada cenário é traduzido em um modelo diferente. A seguir, mostraremos um exemplo de modelo descritivo.

A fábrica de alimentos ImpactaNatural deseja elaborar um sistema de informação para controlar suas atividades. O objetivo é modelar uma solução que atenda às áreas mais importantes da empresa. Os requisitos estão listados a seguir:

- A empresa atua com clientes previamente cadastrados e só atende pessoas jurídicas. Todos os clientes da ImpactaNatural são varejistas de diversas regiões do país. Sobre os clientes, é fundamental armazenarmos um código, que será o identificador único, o CNPJ, a razão social, o endereço de cobrança, o endereço de correspondência e o endereço para entrega das mercadorias compradas, além dos telefones de contato, o contato principal, o ramo de atividade e a data do cadastramento da instituição;
- A empresa possui um elenco bastante variado de produtos, como pães, bolos, doces, entre outros. Sobre os produtos, devemos guardar o código, o nome, a cor, as dimensões, o peso, o preço, a validade, o tempo médio de fabricação e o desenho do produto;
- Para fabricar os alimentos, diversos componentes são essenciais: matéria-prima (farinha, sal, açúcar, fermento etc.), materiais diversos (embalagens, rótulos etc.) e máquinas (centrífuga, forno etc.);
- Cada componente pode ser utilizado em vários produtos e um produto pode utilizar diversos componentes. Sobre cada um dos componentes, devemos registrar: código, nome, quantidade em estoque, preço unitário e unidade de estoque. Para as máquinas, precisamos registrar o tempo médio de vida, a data da compra e a data de fim da garantia;
- No que diz respeito às matérias-primas e aos materiais diversos necessários na elaboração de um produto, precisamos controlar a quantidade necessária e a unidade de medida;
- Devemos controlar o tempo necessário de uso das máquinas e as ferramentas necessárias na elaboração de um produto;
- Precisamos controlar a quantidade de horas necessárias da mão de obra destinada à elaboração do produto;

- Sobre a mão de obra, devemos registrar matrícula, nome, endereço, telefones, cargo, salário, data de admissão e uma descrição com as qualificações profissionais. Toda mão de obra é empregada da Madeira de Lei. Precisamos também registrar a hierarquia de subordinação entre os empregados, pois um empregado pode estar subordinado a somente um empregado e este, por sua vez, pode gerenciar vários outros empregados;
- A Madeira de Lei trabalha com o esquema de encomenda. Ela não mantém o estoque de produtos elaborados, ou seja, cada vez que um cliente solicita um produto, ela o elabora. Uma encomenda pode envolver vários produtos e pertencer a um único cliente. Sobre as encomendas, devemos registrar um número, a data da inclusão, o valor total da encomenda, o valor do desconto (caso exista), o valor líquido, um identificador para a forma de pagamento (se cheque, dinheiro ou cartão de crédito) e a quantidade de parcelas. Sobre os produtos solicitados em uma encomenda, precisamos registrar a quantidade e a data de necessidade do produto;
- Matéria-prima, materiais diversos, máquinas e ferramentas utilizadas para fabricar os alimentos possuem diversos fornecedores que devem ser controlados para que o item não falte e comprometa a fabricação e, consequentemente, a entrega de uma encomenda. Sobre os fornecedores, deve-se registrar CNPJ, razão social, endereço, telefones, pessoa de contato. Um determinado fornecedor pode fornecer diversos itens de cada um dos grupos citados;
- É necessário também realizar manutenções nas máquinas para que elas mantenham sua vida útil e trabalhem com eficiência. A manutenção é feita por empresas especializadas em máquinas industriais. Sobre essas empresas, registramos CNPJ, razão social, endereço, telefones e pessoa de contato. Sempre que uma máquina sofrer manutenção por uma empresa, deve-se registrar a data da manutenção e uma descrição das ações realizadas nela.

Notemos que esse modelo apenas apresenta o cenário de forma ampla. A próxima etapa é a construção do modelo conceitual.

## 1.2.2. Modelo conceitual

Nesse modelo, extraímos informações do modelo descritivo, usando a seguinte técnica:

- Substantivos (pessoas, coisas, papéis, objetos) são denominados entidades, as quais são elementos que possuem informações a serem tratadas;
- Propriedades ou características ligadas aos substantivos são chamadas de atributos, os quais são elementos que caracterizam uma entidade.

Nesse modelo, apenas qualificamos as entidades encontradas. Vejamos a seguir um exemplo do dicionário de dados de um modelo conceitual:

- CLIENTES = Código + CNPJ + razão social + ramo de atividade + data do cadastramento + {telefones} + {endereços} + pessoa de contato;
- EMPREGADOS = Matrícula + nome + {telefones} + cargo + salário + data de admissão + qualificações + endereço;
- EMPRESAS = CNPJ + razão social + {telefones} + pessoa de contato + endereço;
- FORNECEDORES = CNPJ + razão social + endereço + {telefones} + pessoa de contato;
- TIPO DE ENDEREÇO = Código + nome;
- ENDEREÇOS = Número + logradouro + complemento + CEP + bairro + cidade + Estado;
- ENCOMENDAS = Número + data da inclusão + valor total + valor do desconto + valor líquido + ID forma de pagamento + quantidade de parcelas;
- PRODUTOS = Código + nome + cor + dimensões + peso + preço + tempo de fabricação + desenho do produto + horas de mão de obra;

- TIPOS DE COMPONENTE = Código + nome;
- COMPONENTES = Código + nome + quantidade em estoque + preço unitário + unidade;
- MÁQUINAS = Tempo de vida + data da compra + data fim da garantia;
- RE = Quantidade necessária + unidade + tempo de uso + horas da mão de obra;
- RM = Data + descrição;
- RS = Quantidade + data de necessidade.

A partir desse modelo, iniciamos a modelagem de dados no formato de diagrama, ou seja, utilizando representações gráficas para os elementos encontrados nos modelos descritivo e conceitual.

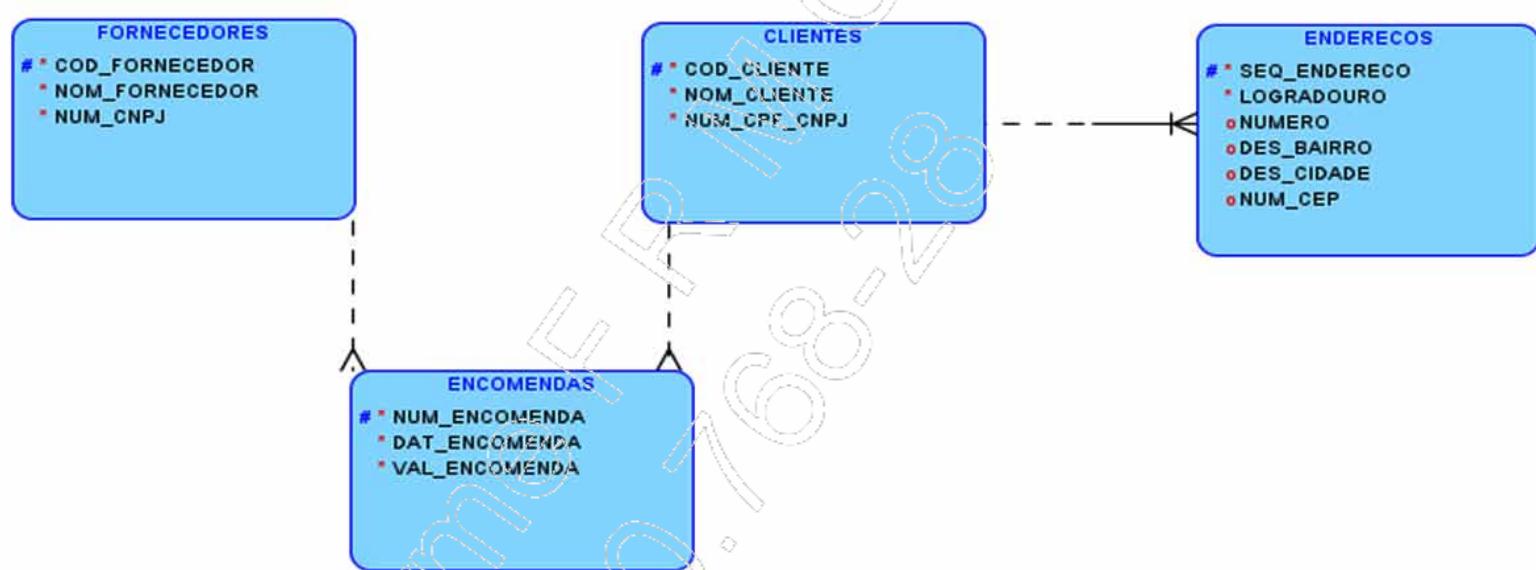
## 1.2.3. Modelo lógico

Esse modelo apresenta, em um formato de diagrama, as entidades e os atributos encontrados nos modelos anteriores. Nesse modelo, também conhecido como diagrama lógico de dados, ainda não é possível determinar qual banco de dados será utilizado.

As ferramentas de modelagem geralmente iniciam os modelos a partir do modelo lógico de dados. A partir dele é que será gerado o modelo para implementação no banco de dados. Veremos adiante um exemplo de um diagrama lógico de dados, mas antes vamos entender algumas regras importantes.

Toda entidade deve ter um identificador único, que representa um valor que não se repete para cada ocorrência da mesma entidade. Por exemplo, a entidade Fornecedor não tem nenhum atributo de valor único, dessa forma, criamos um atributo fictício chamado ID\_FORNECEDOR. Essa regra é importante, pois esse identificador único permite o relacionamento entre as entidades.

Devemos definir quais atributos são obrigatórios e quais são opcionais, além de determinar os atributos que deverão ter valores específicos (por exemplo, sexo: M para masculino e F para feminino).

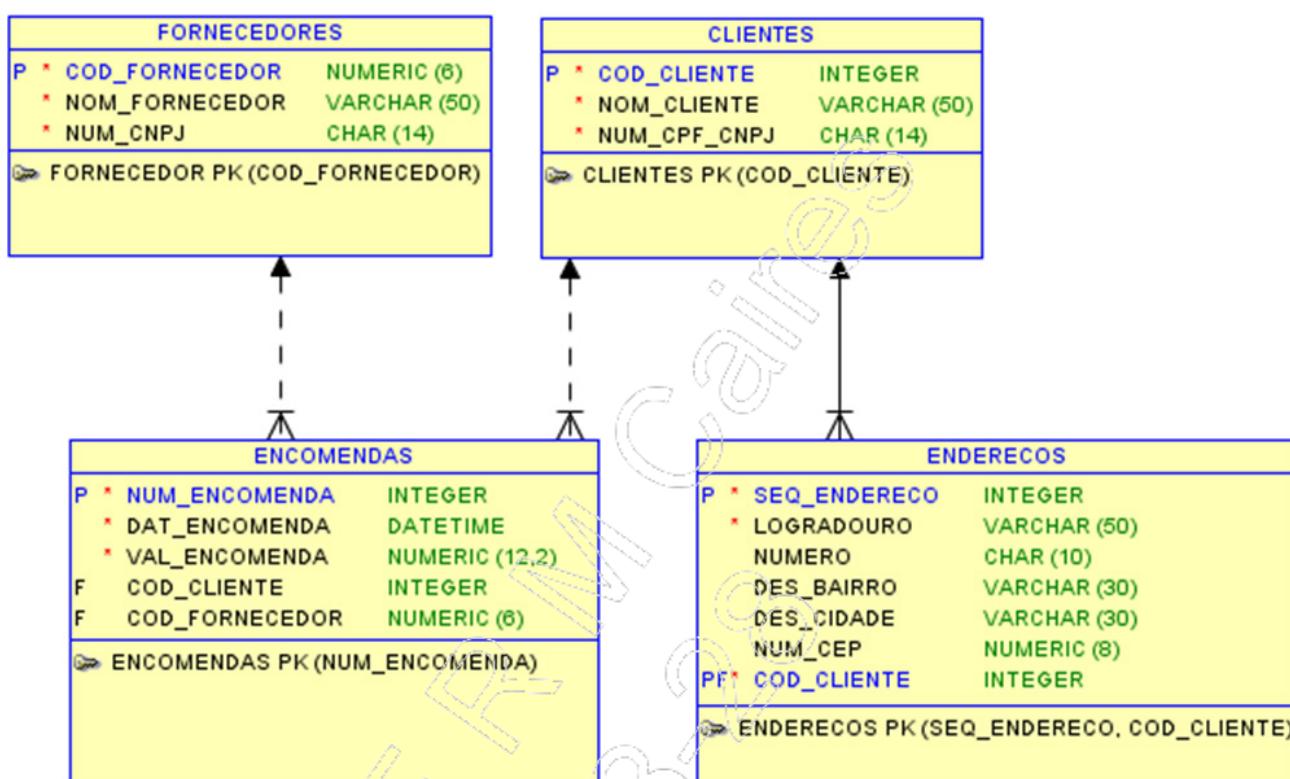


Nesse esquema, as caixas representam as entidades, as linhas representam os relacionamentos e os valores nas caixas representam os atributos.

## 1.2.4. Modelo físico

O modelo físico de dados pode ser obtido por meio do diagrama lógico de dados e está associado ao software de gerenciamento de banco de dados, neste caso, o SQL Server 2014.

Vejamos, a seguir, um exemplo de modelo físico de dados:

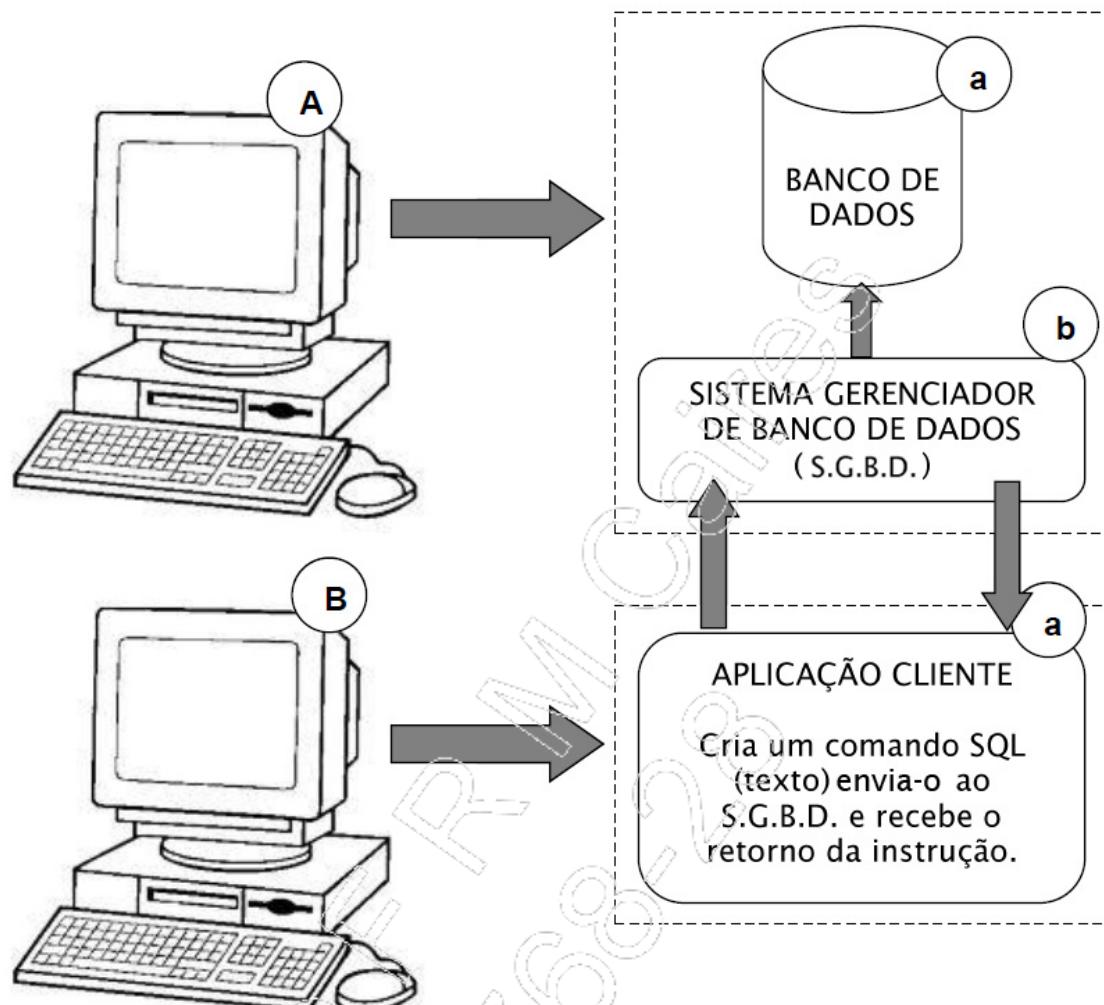


## 1.3. Arquitetura cliente / servidor

Essa arquitetura funciona da seguinte forma:

- Usuários acessam o servidor por meio de um aplicativo instalado no próprio servidor ou de outro computador;
- O computador cliente executa as tarefas do aplicativo, ou seja, fornece a interface do usuário (tela, processamento de entrada e saída) e manda uma solicitação ao servidor;
- O servidor de banco de dados processa a solicitação, que pode ser uma consulta, alteração, exclusão, inclusão etc.

A imagem a seguir ilustra a arquitetura cliente / servidor:



- **A - Servidor**
  - a - Software servidor de banco de dados. É ele que gerencia todo o acesso ao banco de dados. Ele recebe os comandos SQL, verifica sua sintaxe e os executa, enviando o retorno para a aplicação que enviou o comando;
  - b - Banco de dados, incluindo as tabelas que serão manipuladas.
- **B - Cliente**
  - a - Aplicação contendo a interface visual que envia os comandos SQL ao servidor.

### 1.4. As linguagens SQL e T-SQL

Toda a manipulação de um banco de dados é feita por meio de uma linguagem específica, com uma exigência sintática rígida, chamada SQL (Structure Query Language). Os fundamentos dessa linguagem estão baseados no conceito de banco de dados relacional.

Essa linguagem foi desenvolvida pela IBM no início da década de 1970 e, posteriormente, foi adotada como linguagem padrão pela ANSI (American National Standard Institute) e pela ISO (International Organization for Standardization), em 1986 e 1987, respectivamente.

A T-SQL (Transact-SQL) é uma implementação da Microsoft para a SQL padrão ANSI. Ela cria opções adicionais para os comandos e também cria novos comandos que permitem o recurso de programação, como os de controle de fluxo, variáveis de memória etc.



Além da T-SQL, há outras implementações da SQL, como Oracle PL/SQL (Procedural Language/SQL) e IBM's SQL Procedural Language.

Veja um exemplo de comando SQL:

The screenshot shows the SSMS interface with a query window titled "SQLQuery1.sql - NO...OTEDELL\magno (52)\*". The query is a SELECT statement:

```
-- comando de consulta SQL
SELECT * FROM TABELADEP
```

The results pane displays a table with columns "COD\_DEPTO" and "DEPTO". The data is as follows:

	COD_DEPTO	DEPTO
1	1	PESSOAL
2	2	C.P.D.
3	3	CONTROLE DE ESTOQUE
4	4	COMPRAS
5	5	PRODUCAO
6	6	DIRETORIA
7	7	TELEMARKETING
8	8	FINANCEIRO

A red arrow points from the text "Resultado da execução do comando SELECT." to the results table.

Resultado da execução do comando SELECT.

## 1.5. SQL Server

O SQL Server é uma plataforma de banco de dados utilizada para armazenar dados e processá-los, tanto em um formato relacional quanto em documentos XML. Também é utilizada em aplicações de comércio eletrônico e atua como uma plataforma inteligente de negócios para integração e análise de dados, bem como de soluções.

Para essas tarefas, o SQL Server faz uso da linguagem T-SQL para gerenciar bancos de dados relacionais, que contém, além da SQL, comandos de linguagem procedural.

### 1.5.1. Componentes

O SQL Server oferece diversos componentes opcionais e ferramentas relacionadas que auxiliam e facilitam a manipulação de seus sistemas. Por padrão, nenhum dos componentes será instalado.

A seguir, descreveremos as funcionalidades oferecidas pelos principais componentes do SQL Server:

- **SQL Server Database Engine:** É o principal componente do MS-SQL. Recebe as instruções SQL, executa-as e devolve o resultado para a aplicação solicitante. Funciona como um serviço no Windows e, normalmente, é iniciado juntamente com a inicialização do sistema operacional;
- **Analysis Services:** Usado para consultas avançadas, que envolvem muitas tabelas simultaneamente, e para geração de estruturas OLAP (On-Line Analytical Processing);
- **Reporting Services:** Ferramenta para geração de relatórios;
- **Integration Services:** Facilita o processo de transferência de dados entre bancos de dados.

## 1.5.2. Objetos de banco de dados

Os objetos que fazem parte de um sistema de banco de dados do SQL Server são criados dentro do objeto **DATABASE**, que é uma estrutura lógica formada por dois tipos de arquivo: um arquivo responsável por armazenar os dados e outro, por armazenar as transações realizadas. Veja a seguir alguns objetos de banco de dados do SQL Server.

### 1.5.2.1. Tabelas

Os dados são armazenados em objetos de duas dimensões denominados tabelas (**tables**), formadas por linhas e colunas. As tabelas contêm todos os dados de um banco de dados e são a principal forma para coleção de dados.

### 1.5.2.2. Índices

Quando realizamos uma consulta de dados, o SQL Server 2014 faz uso dos índices (**index**) para buscar, de forma fácil e rápida, informações específicas em uma tabela ou VIEW indexada.

### 1.5.2.3. CONSTRAINT

São objetos cuja finalidade é estabelecer regras de integridade e consistência nas colunas das tabelas de um banco de dados. São cinco os tipos de **CONSTRAINT** oferecidos pelo SQL Server: **PRIMARY KEY**, **FOREIGN KEY**, **UNIQUE**, **CHECK** e **DEFAULT**.

### 1.5.2.4. VIEW (Visão)

Definimos uma VIEW (visualização) como uma tabela virtual composta por linhas e colunas de dados, os quais são provenientes de tabelas referenciadas em uma consulta que define essa tabela.

Esse objeto oferece uma visualização lógica dos dados de uma tabela, de modo que diversas aplicações possam compartilhá-la.

Essas linhas e colunas são geradas de forma dinâmica no momento em que é feita uma referência a uma VIEW.

### 1.5.2.5. PROCEDURE (Procedimento Armazenado)

Nesse objeto encontramos um bloco de comandos T-SQL, responsável por uma determinada tarefa. Sua lógica pode ser compartilhada por diversas aplicações. A execução de uma procedure é realizada no servidor de dados. Por isso, seu processamento ocorre de forma rápida, visto que seu código tende a ficar compilado na memória.

### 1.5.2.6. FUNCTION (Função)

Nesse objeto, encontramos um bloco de comandos T-SQL responsável por uma determinada tarefa, isto é, a função (FUNCTION) executa um procedimento e retorna um valor. Sua lógica pode ser compartilhada por diversas aplicações.

### 1.5.2.7. TRIGGER (Gatilho)

Esse objeto também possui um bloco de comandos T-SQL. O TRIGGER é criado sobre uma tabela e ativado automaticamente no momento da execução dos comandos UPDATE, INSERT ou DELETE.

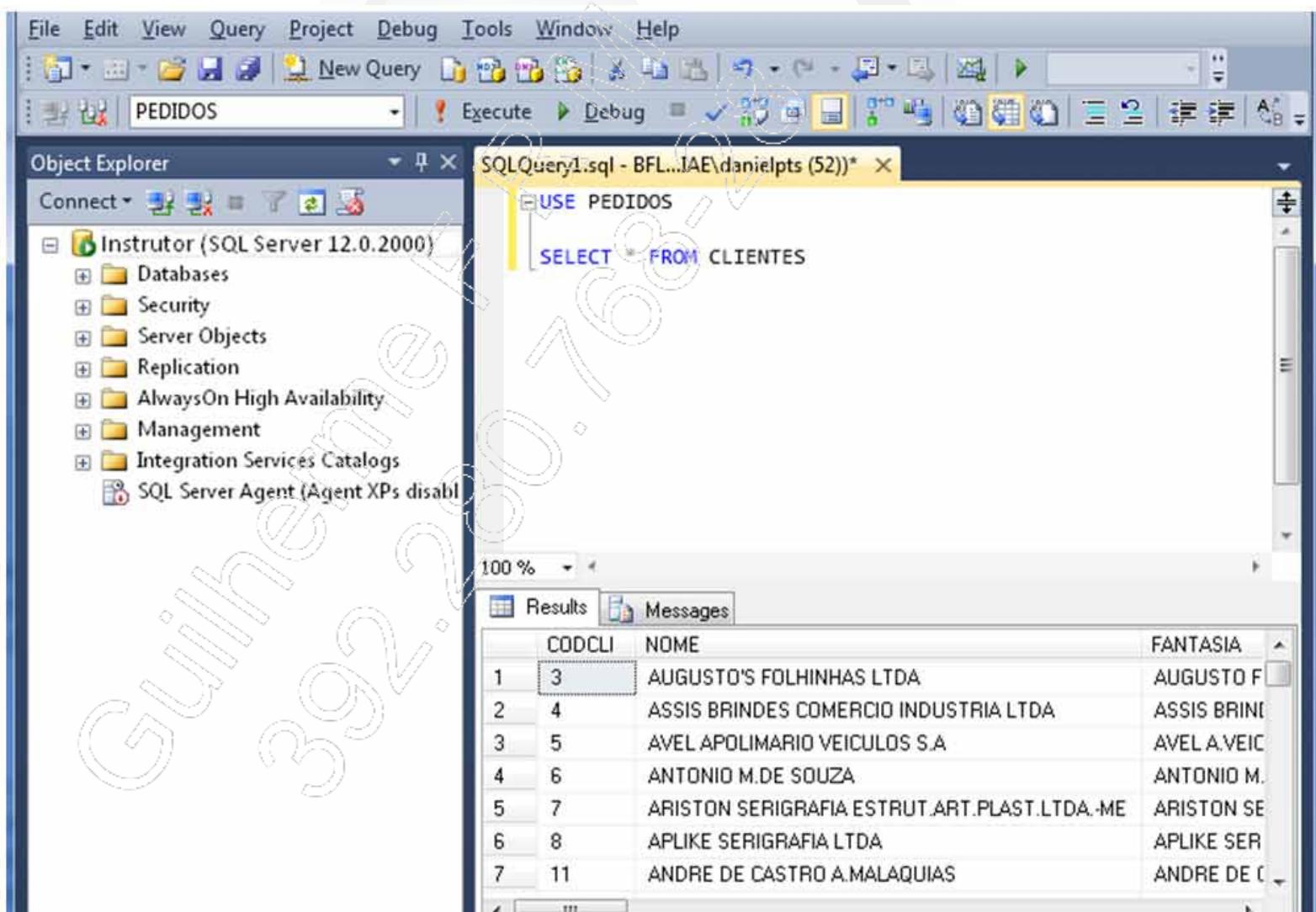
Quando atualizamos, inserimos ou excluímos dados em uma tabela, o TRIGGER automaticamente grava em uma tabela temporária os dados do registro atualizado, inserido ou excluído.

## 1.6. Ferramentas de gerenciamento

A seguir, descreveremos as funcionalidades oferecidas pelas ferramentas de gerenciamento disponíveis no SQL Server e que trabalham associadas aos componentes descritos anteriormente:

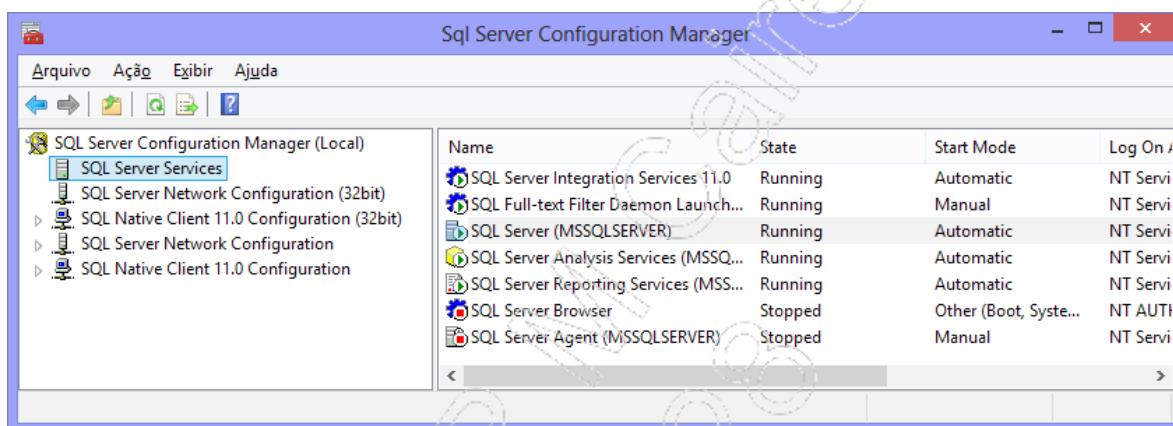
- **SQL Server Management Studio (SSMS)**

É um aplicativo usado para gerenciar bancos de dados e que permite criar, alterar e excluir objetos no banco de dados:



- **SQL Server Configuration Manager**

Permite visualizar, alterar e configurar os serviços dos componentes do SQL Server:



- **Microsoft SQL Server Profiler**

Essa ferramenta permite capturar e salvar dados de cada evento em um arquivo ou tabela para análise posterior.

- **Database Engine Tuning Advisor**

Analisa o desempenho das operações e sugere opções para sua melhora.

- **SQL Server Data Tools**

Possui uma interface que integra os componentes Business Intelligence, Analysis Services, Reporting Services e Integration Services.

## 1.7. SQL Server Management Studio (SSMS)

Essa é a principal ferramenta para gerenciamento de bancos de dados, por isso, é fundamental conhecer o seu funcionamento. Os próximos tópicos apresentam como inicializar o SSMS, sua interface, como executar comandos e como salvar scripts.

### 1.7.1. Inicializando o SSMS

Para abrir o SQL Server Management Studio, siga os passos adiante:

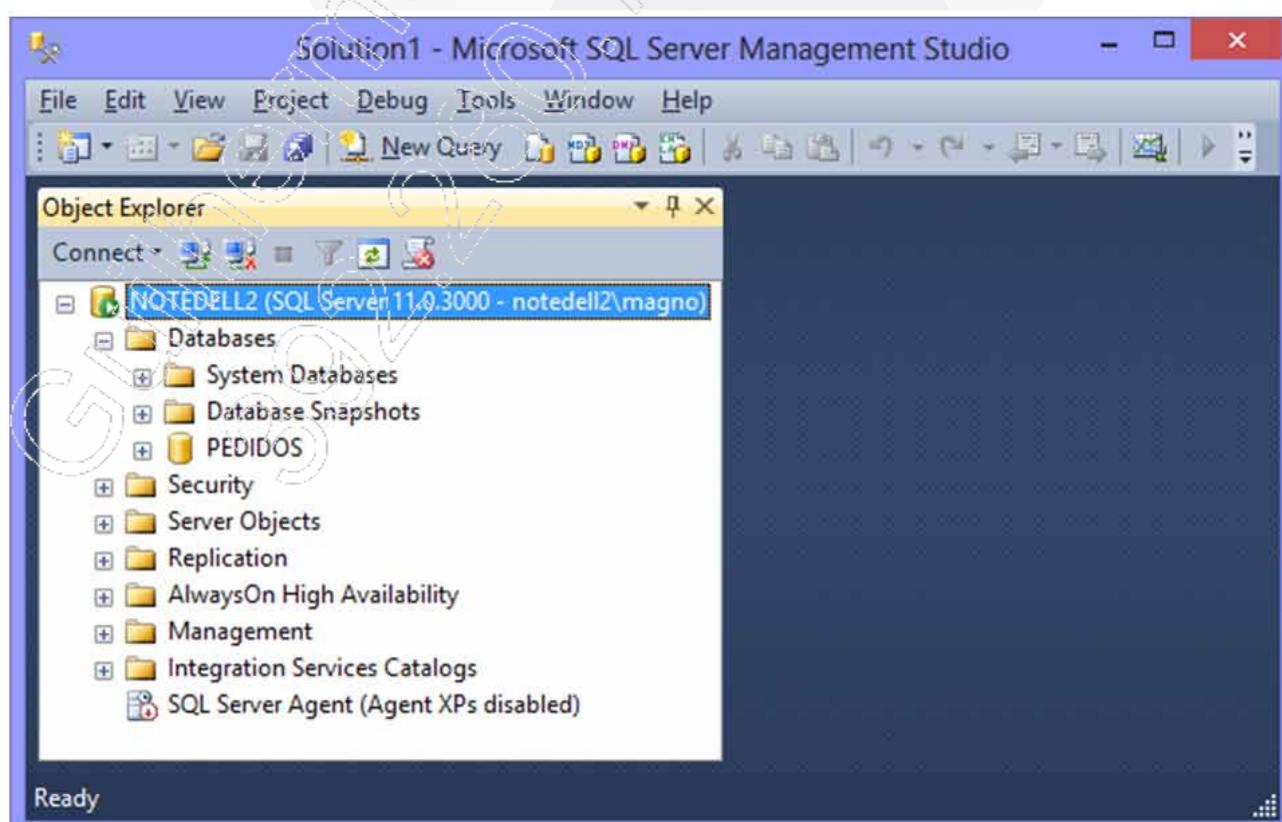
1. Clique no botão **Iniciar** e selecione a opção **Todos os Programas**;
2. Selecione **Microsoft SQL Server 2014** e, em seguida, **SQL Server 2014 Management Studio**:



3. Na tela **Connect to Database Engine**, escolha a opção **Windows Authentication** para o campo **Authentication** e, no campo **Server Name**, especifique o nome do servidor com o qual será feita a conexão:



4. Clique no botão **Connect**. A interface do **SQL Server Management Studio** será aberta, conforme mostra a imagem a seguir:

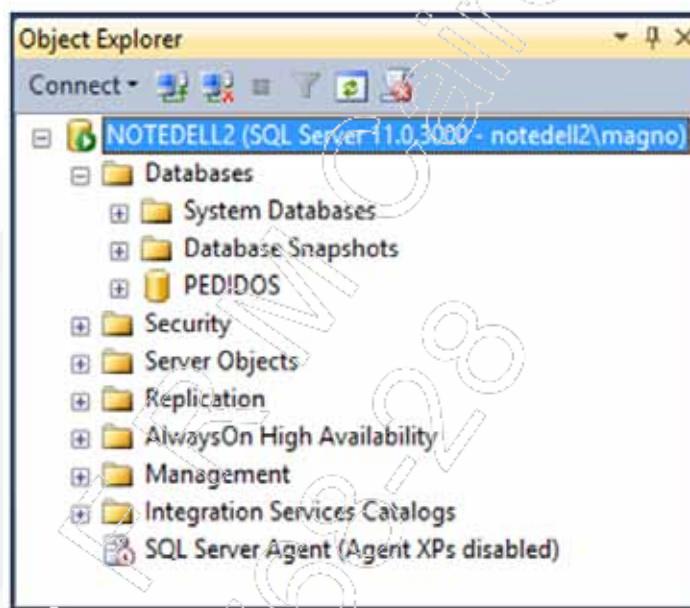


## 1.7.2. Interface

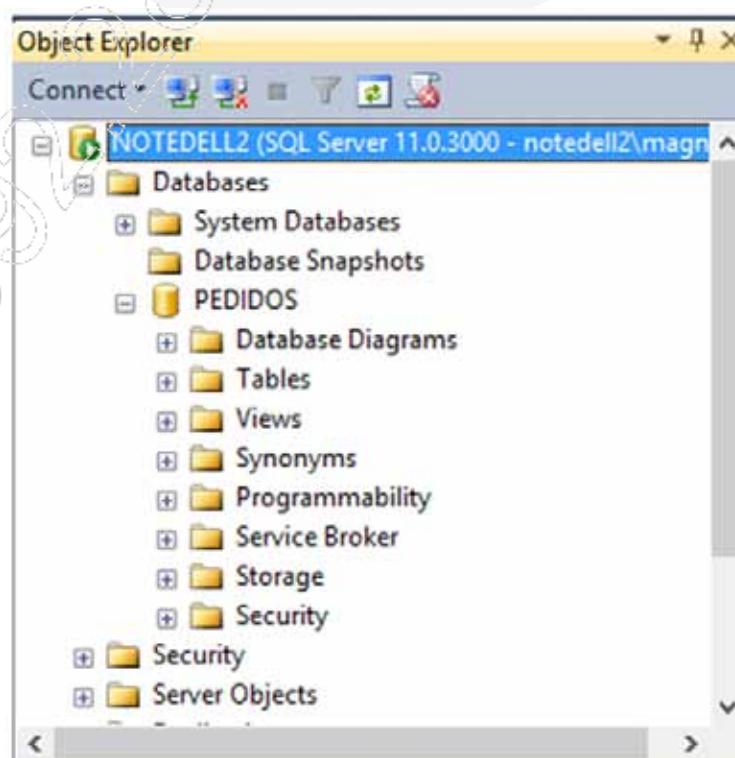
A interface do SSMS é composta pelo **Object Explorer** e pelo **Code Editor**, explicados a seguir:

- **Object Explorer**

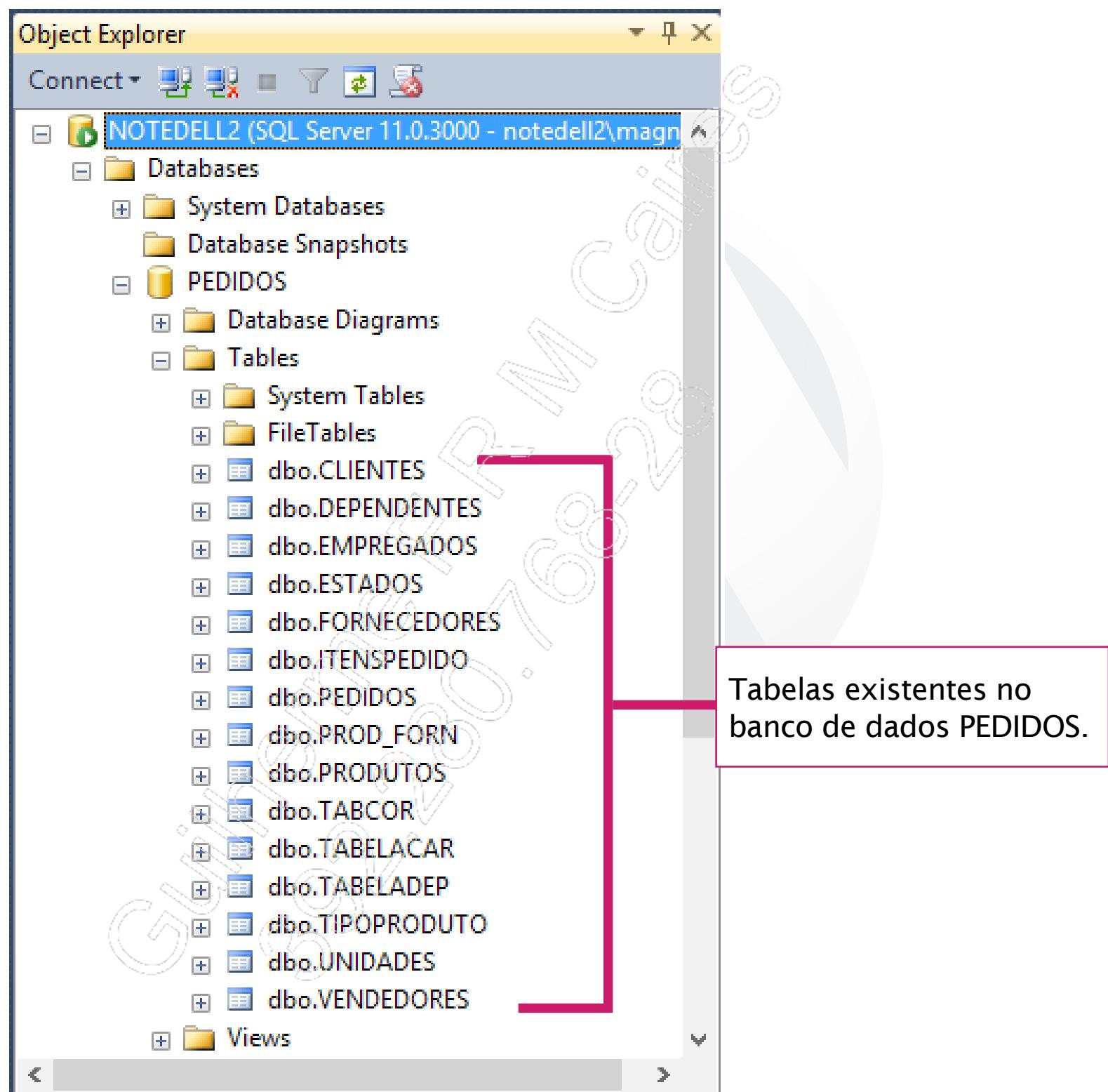
É uma janela que contém todos os elementos existentes dentro do seu servidor MS-SQL Server no formato de árvore:



Observe que a pasta **Databases** mostra todos os bancos de dados existentes no servidor. No caso da imagem a seguir, **PEDIDOS** é um banco de dados:



Expandindo o item **PEDIDOS** e depois o item **Tables**, veremos os nomes das tabelas existentes no banco de dados:



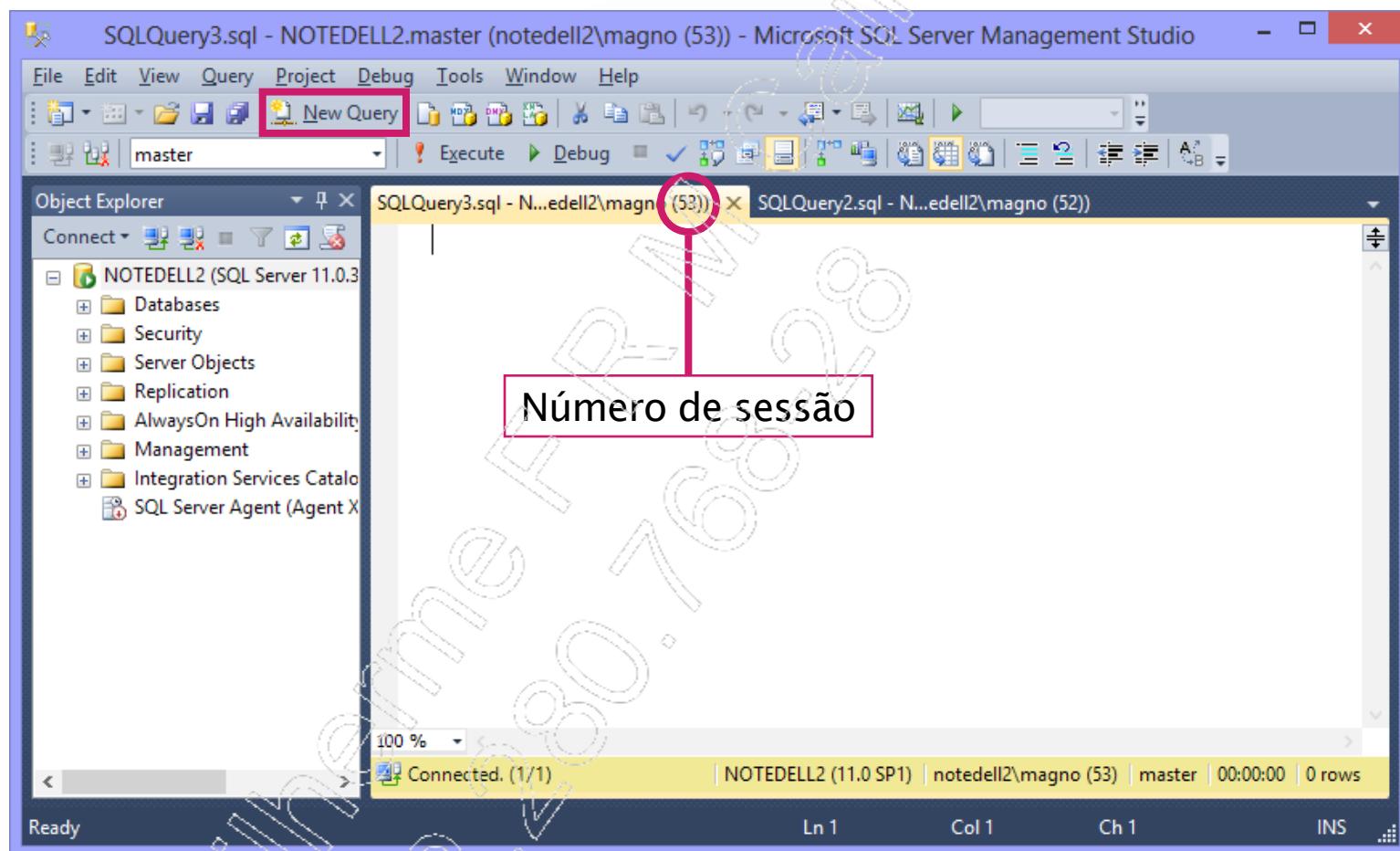
# SQL 2014 - Módulo I

Expandindo uma das tabelas, veremos características da sua estrutura, bem como as suas colunas:

The screenshot shows the SQL Server Object Explorer window. The left pane displays a tree structure of database objects. Under the 'PEDIDOS' database, there are 'Database Diagrams', 'Tables', 'System Tables', 'FileTables', and three specific tables: 'dbo.CLIENTES', 'dbo.DEPENDENTES', and 'dbo.EMPREGADOS'. The 'dbo.EMPREGADOS' table is expanded, revealing its columns: CODFUN (PK, int, not null), NOME (varchar(35), not null), NUM\_DEPEND (smallint, null), DATA\_NASCIMENTO (datetime, null), COD\_DEPTO (FK, int, null), COD\_CARGO (FK, int, null), DATA\_ADMISSAO (datetime, null), SALARIO (numeric(18,2), null), PREMIO\_MENSAL (numeric(10,2), null), SINDICALIZADO (varchar(1), null), OBS (text, null), FOTO (image, null), and COD\_SUPERVISOR (int, null). A red box highlights the 'Columns' node under 'EMPREGADOS', and a callout bubble points to it with the text 'Campos (colunas) da tabela'.

- **Code Editor**

O **Code Editor** (Editor de código) do SQL Server Management Studio permite escrever comandos T-SQL, MDX, DMX, XML/A e XML. Clicando no botão **New Query** (Nova Consulta), será aberta uma janela vazia para edição dos comandos. Cada vez que clicarmos em **New Query**, uma nova aba vazia será criada:

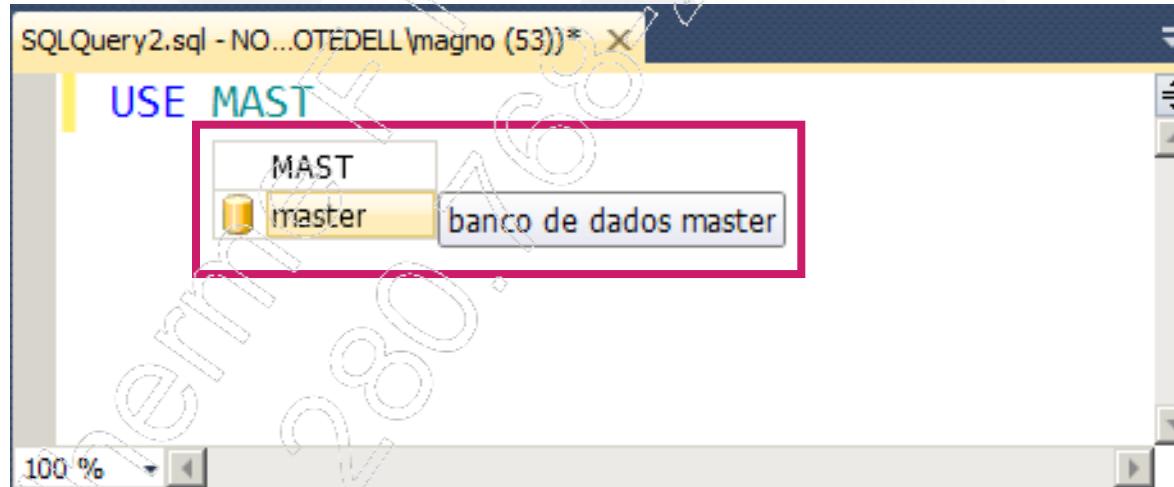


Cada uma dessas abas representa uma conexão com o banco de dados e recebe um **número de sessão**. Cada conexão tem um número de sessão único, mesmo que tenha sido aberta pelo mesmo usuário com o mesmo LOGIN e senha. Quando outra aplicação criada em outra linguagem, como Delphi, VB ou C#, abrir uma conexão, ela também receberá um número único de sessão.

## 1.7.3. Executando um comando

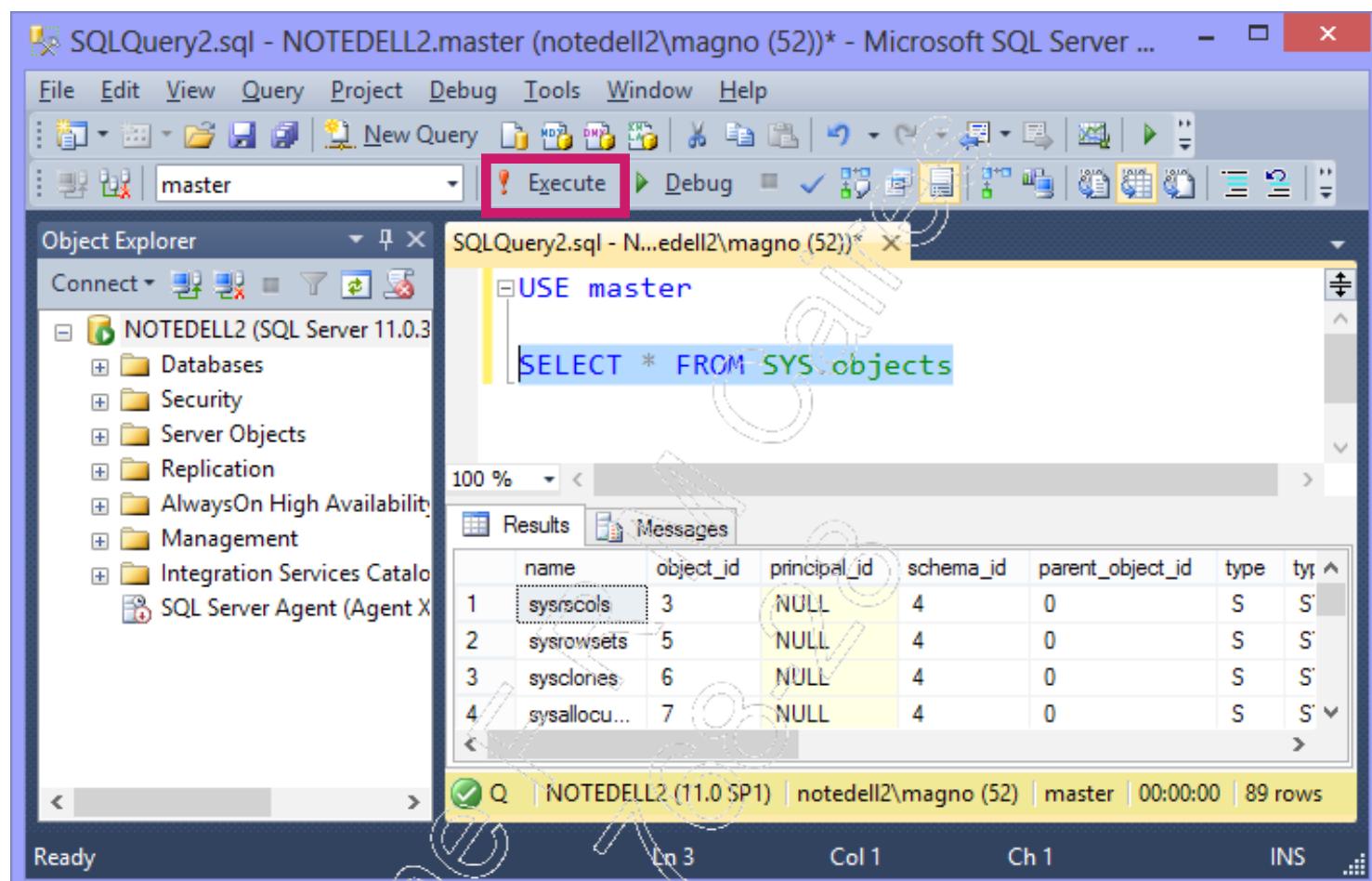
Para executar um comando a partir do SQL Server Management Studio, adote o seguinte procedimento:

1. Escreva o comando desejado no **Code Editor**. Enquanto um comando é digitado no **Code Editor**, o SQL Server oferece um recurso denominado **IntelliSense**, que destaca erros de sintaxe e digitação e fornece ajuda para a utilização de parâmetros no código. Ele está ativado por padrão, mas pode ser desativado. Para forçar a exibição do **IntelliSense**, utilize CTRL + Barra de espaço:



2. Selecione o comando escrito. A seleção é necessária apenas quando comandos específicos devem ser executados, dentre vários;

3. Na barra de ferramentas do **Code Editor**, clique sobre o botão **Execute** ou pressione a tecla **F5** (ou **CTRL + E**) para que o comando seja executado. O resultado do comando será exibido na parte inferior da interface, conforme a imagem a seguir:



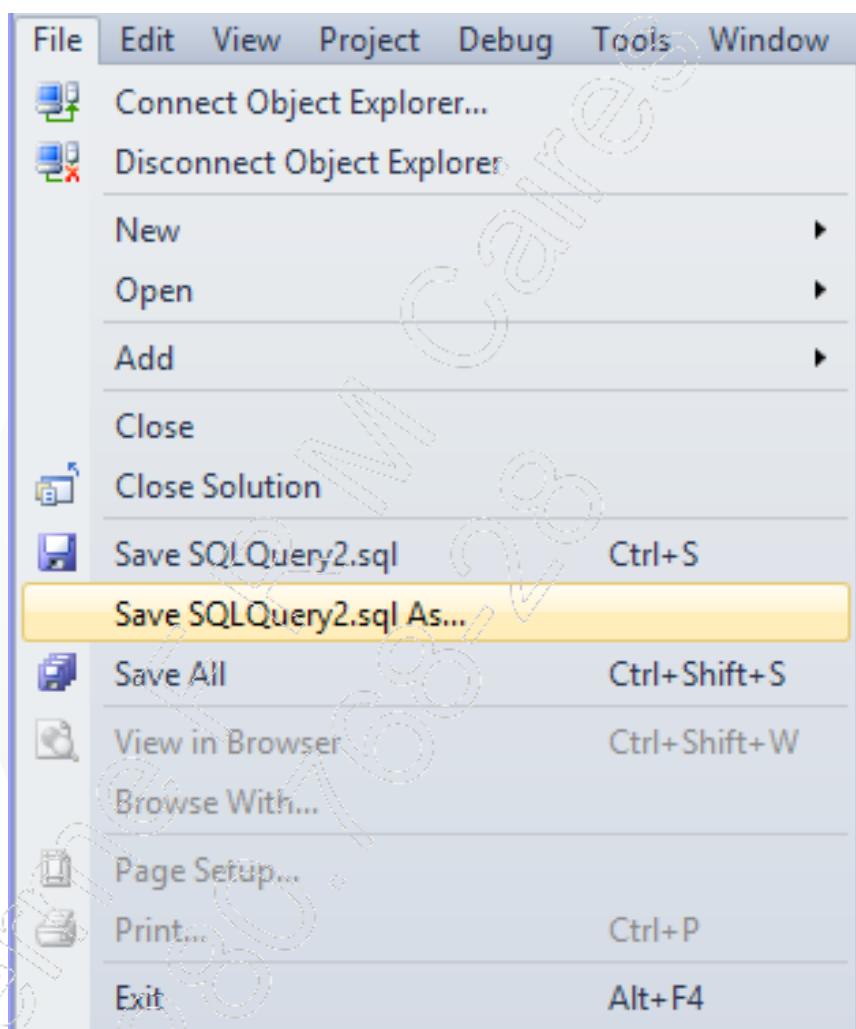
Com relação ao procedimento anterior, é importante considerar as seguintes informações:

- É possível ocultar o resultado do comando por meio do atalho **CTRL + R**;
- **MASTER** é um banco de dados de sistema e que já vem instalado no MS-SQL Server;
- Caso não selecione um comando específico, todos os comandos escritos no texto serão executados e, nesse caso, eles precisarão estar organizados em uma sequência lógica perfeita, senão ocorrerão erros;
- Quando salvamos o arquivo contido no editor, ele recebe a extensão **.sql**, por padrão. É um arquivo de texto também conhecido como **SCRIPT SQL**.

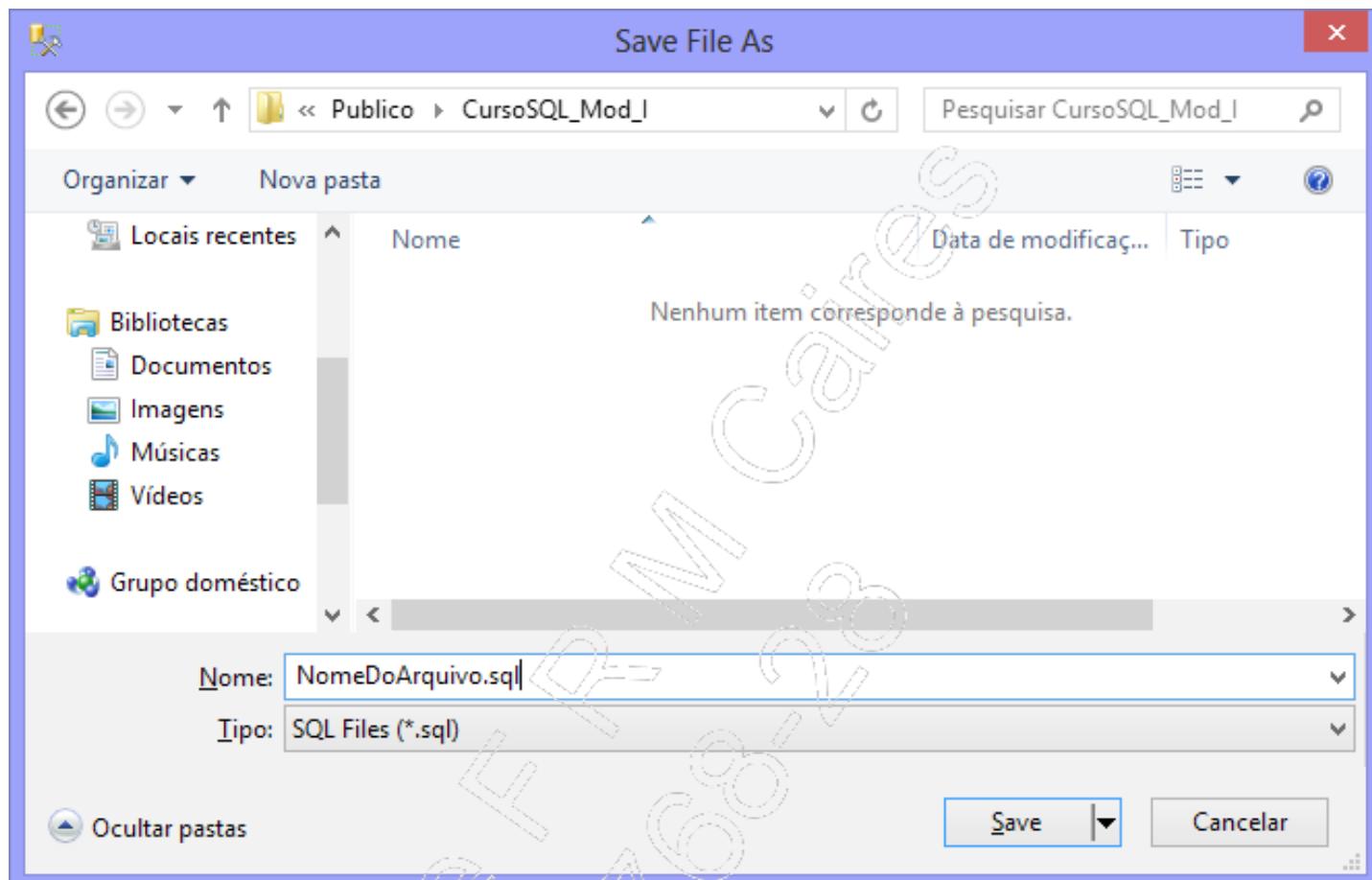
## 1.7.4. Salvando scripts

Para salvar os scripts utilizados, siga os passos adiante:

1. Acesse o menu **File** e clique em **Save As...**:



2. Digite o nome escolhido para o script:



3. Clique em Save.

## Pontos principais

**Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.**

- Um banco de dados armazena informações e seu principal objeto são as tabelas;
- É fundamental o design de um banco de dados para que possua um bom desempenho;
- Os modelos de design de um banco de dados são: modelo descritivo, modelo conceitual, modelo lógico e modelo físico;
- A linguagem T-SQL (Transact-SQL) é baseada na linguagem SQL ANSI, desenvolvida pela IBM na década de 1970;
- Os principais objetos de um banco de dados são: tabelas, índices, CONSTRAINT, VIEW, PROCEDURE e TRIGGER;
- Uma tabela precisa ter uma coluna que identifica de forma única cada uma de suas linhas. Essa coluna é chamada de CHAVE PRIMÁRIA;
- O SQL Server Management Studio (SSMS) é a principal ferramenta para gerenciamento de bancos de dados.

# Introdução ao SQL Server 2014

## Teste seus conhecimentos

1

Guilherme  
Caires  
392.280-100



**IMPACTA**  
EDITORA

## 1. Quais são os modelos recomendados para a criação de um banco de dados?

- a) Descritivo, conceitual e físico.
- b) Conceitual, lógico e físico.
- c) Descritivo, conceitual, lógico e físico.
- d) Descritivo, lógico e físico.
- e) Não precisamos de modelagem para criação de banco de dados.

## 2. Qual é a diferença entre os modelos lógico e físico?

- a) O modelo físico complementa o modelo lógico com a implementação do fabricante do banco de dados.
- b) O modelo lógico é mais completo que o físico.
- c) Não existe diferença nos modelos.
- d) Podemos afirmar que o modelo lógico possui todas as características para a implementação de um banco de dados.
- e) Não precisamos de modelagem para trabalhar com banco de dados.

### 3. Com relação à linguagem SQL, podemos afirmar que:

- a) A linguagem SQL foi desenvolvida pela IBM e não pode ser utilizada em outros bancos de dados.
- b) O SQL Server 2014 não utiliza a linguagem SQL.
- c) O SQL Server 2014 utiliza a linguagem T-SQL que é uma implementação da linguagem SQL.
- d) Nunca devemos utilizar a linguagem SQL, pois está ultrapassada.
- e) Todas as afirmações acima estão erradas.

### 4. Com relação ao SQL Server 2014, podemos afirmar que:

- a) É uma plataforma de banco de dados que armazena dados no formato relacional ou XML.
- b) Utiliza a linguagem T-SQL.
- c) Não pode trabalhar com PROCEDURES.
- d) Não utiliza VIEWS ou FUNCTIONS.
- e) As afirmações a e b estão corretas.

## 5. Qual objeto é responsável pelas regras de integridade?

- a) VIEW
- b) Tabelas
- c) CONSTRAINT
- d) PROCEDURE
- e) TRIGGER

# Criando um banco de dados

2

- ✓ CREATE DATABASE;
- ✓ CREATE TABLE;
- ✓ Tipos de dados;
- ✓ Campo de autonumeração (IDENTITY);
- ✓ Constraints;
- ✓ Normalização de dados;
- ✓ Índices.



**IMPACTA**  
EDITORA

## 2.1. Introdução

Neste capítulo, veremos os recursos iniciais para criação de banco de dados: os comandos **CREATE DATABASE** e **CREATE TABLE**, os tipos de dados, as constantes e como fazer a normalização dos dados.

Quando formos mostrar as opções de sintaxe de um comando SQL, usaremos a seguinte nomenclatura:

- []: Termos entre colchetes são opcionais;
- <>: Termos entre os sinais menor e maior são nomes ou valores definidos por nós;
- {a1|a2|a3...}: Lista de alternativas mutuamente exclusivas.

## 2.2. CREATE DATABASE

**DATABASE** é um conjunto de arquivos que armazena todos os objetos do banco de dados.

Para que um banco de dados seja criado no SQL Server, é necessário utilizar a instrução **CREATE DATABASE**, cuja sintaxe básica é a seguinte:

```
CREATE DATABASE <nome do banco de dados>
```

A seguir, veja um exemplo de criação de banco de dados:

```
CREATE DATABASE SALA_DE_AULA;
```

Essa instrução criará dois arquivos:

- **SALA\_DE\_AULA.MDF**: Armazena os dados;
- **SALA\_DE\_AULA\_LOG.LDF**: Armazena os logs de transações.

Normalmente, esses arquivos estão localizados na pasta **DATA** dentro do diretório de instalação do SQL-Server.

Assim que são criados, os bancos de dados possuem apenas os objetos de sistema, como tabelas, PROCEDURES, VIEWS, necessários para o gerenciamento das tabelas.

Para facilitar o acesso a um banco de dados, devemos colocá-lo em uso, mas isso não é obrigatório. Para colocar um banco de dados em uso, utilize o seguinte código:

```
USE <nome do banco de dados>
```

Veja um exemplo:

```
USE SALA_DE_AULA
```

Observe que na parte superior esquerda do SSMS existe um ComboBox que mostra o nome do banco de dados que está em uso.

## 2.3. CREATE TABLE

Os principais objetos de um banco de dados são suas tabelas, responsáveis pelo armazenamento dos dados.

A instrução **CREATE TABLE** deve ser utilizada para criar tabelas dentro de um banco de dados já existente. A sintaxe para uso dessa instrução é a seguinte:

```
CREATE TABLE <nome_tabela>
( <nome_campo1> <data_type> [IDENTITY [<inicio>,<incremento>]]
    [NOT NULL] [DEFAULT <exprDef>]
 [, <nome_campo2> <data_type> [NOT NULL] [DEFAULT <exprDef>]
```

Em que:

- **<nome\_tabela>**: Nome que vai identificar a tabela. A princípio, nomes devem começar por uma letra, seguida de letras, números e sublinhados. Porém, se o nome for escrito entre colchetes, poderá ter qualquer sequência de caracteres;
- **<nome\_campo>**: Nome que vai identificar cada coluna ou campo da tabela. É criado utilizando a regra para os nomes das tabelas;
- **<data\_type>**: Tipo de dado que será gravado na coluna (texto, número, data etc.);
- **[IDENTITY [<inicio>,<incremento>]]**: Define um campo como autonumeração;
- **[NOT NULL]**: Define um campo que precisa ser preenchido, isto é, não pode ficar vazio (**NULL**);
- **[DEFAULT <exprDef>]**: Valor que será gravado no campo, caso ele fique vazio (**NULL**).

Com relação à sintaxe de **CREATE TABLE**, é importante considerar, ainda, as seguintes informações:

- A sintaxe descrita foi simplificada, há outras cláusulas na instrução **CREATE TABLE**;
- Uma tabela não pode conter mais de um campo **IDENTITY**;
- Uma tabela não pode conter mais de uma chave primária, mas pode ter uma chave primária composta por vários campos.

A seguir, veja um exemplo de como criar uma tabela em um banco de dados:

```
CREATE TABLE ALUNOS
(
    NUM_ALUNO           INT,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO    DATETIME,
    IDADE               TINYINT,
    E_MAIL               VARCHAR(50),
    FONE_RES             CHAR(8),
    FONE_COM             CHAR(8),
    FAX                 CHAR(8),
    CELULAR              CHAR(9),
    PROFISSAO            VARCHAR(40),
    EMPRESA              VARCHAR(50) );
```

A estrutura dessa tabela não respeita as regras de normalização de dados. Adiante, corrigiremos esse problema.

## 2.4. Tipos de dados

Cada elemento, como uma coluna, variável ou expressão, possui um tipo de dado. O tipo de dado especifica o tipo de valor que o objeto pode armazenar, como números inteiros, texto, data e hora etc. O SQL Server organiza os tipos de dados dividindo-os em categorias.

A seguir, serão descritas as principais categorias de tipos de dados utilizados na linguagem Transact-SQL.

### 2.4.1. Numéricos exatos

A tabela a seguir descreve alguns dos tipos de dados que fazem parte dessa categoria:

- **Inteiros**

Nome	Descrição
<b>bigint 8 bytes</b>	Valor de número inteiro compreendido entre $-2^{63}$ (-9,223,372,036,854,775,808) e $2^{63}-1$ (9,223,372,036,854,775,807).
<b>int 4 bytes</b>	Valor de número inteiro compreendido entre $-2^{31}$ (-2,147,483,648) e $2^{31}-1$ (2,147,483,647).
<b>smallint 2 bytes</b>	Valor de número inteiro compreendido entre $-2^{15}$ (-32,768) e $2^{15}-1$ (32,767).
<b>tinyint 1 byte</b>	Valor de número inteiro de 0 a 255.

- Bit

Nome	Descrição
<b>bit 1 byte</b>	Valor de número inteiro com o valor 1 ou o valor 0.

- Numéricos exatos

Nome	Descrição
<b>decimal(&lt;T&gt;,&lt;D&gt;)</b>	Valor numérico de precisão e escala fixas de $-10^{38} + 1$ até $10^{38} - 1$ .
<b>numeric(&lt;T&gt;,&lt;D&gt;)</b>	Valor numérico de precisão e escala fixas de $-10^{38} + 1$ até $10^{38} - 1$ .

Nos numéricos exatos, é importante considerar as seguintes informações:

- <T>: Corresponde à quantidade máxima de algarismos que o número pode ter;
- <D>: Corresponde à quantidade máxima de casas decimais que o número pode ter;
- A quantidade de casas decimais <D> está contida na quantidade máxima de algarismos <T>;
- A quantidade de bytes ocupada varia dependendo de <T>.
- Valores monetários

Nome	Descrição
<b>money 8 bytes</b>	Compreende valores monetários ou de moeda corrente entre $-922.337.203.685.477,5808$ e $922.337.203.685.477,5807$ .
<b>smallmoney 4 bytes</b>	Compreende valores monetários ou de moeda corrente entre $-214,748.3648$ e $+214,748.3647$ .

## 2.4.2. Numéricos aproximados

A tabela a seguir descreve alguns dos tipos de dados que fazem parte dessa categoria:

Nome	Descrição
<b>float[(n)]</b>	Valor numérico de precisão flutuante entre -1.79E + 308 e -2.23E - 308, 0 e de 2.23E + 308 até 1.79E + 308.
<b>real</b> <b>o mesmo que</b> <b>float(24)</b>	Valor numérico de precisão flutuante entre -3.40E + 38 e -1.18E - 38, 0 e de 1.18E - 38 até 3.40E + 38.

Em que:

- O valor de **n** determina a precisão do número. O padrão (default) é 53;
- Se **n** está entre 1 e 24, a precisão é de 7 algarismos e ocupa 4 bytes de memória. Com **n** entre 25 e 53, a precisão é de 15 algarismos e ocupa 8 bytes.

Esse tipo de dado é chamado de “Numéricos aproximados” porque podem gerar imprecisão na parte decimal.

## 2.4.3. Data e hora

A tabela a seguir descreve alguns dos tipos de dados que fazem parte dessa categoria:

Nome	Descrição
<b>datetime</b> 8 bytes	Data e hora compreendidas entre 1º de janeiro de 1753 e 31 de dezembro de 9999, com a exatidão de 3.33 milissegundos.
<b>smalldatetime</b> 4 bytes	Data e hora compreendidas entre 1º de janeiro de 1900 e 6 de junho de 2079, com a exatidão de 1 minuto.
<b>datetime2[(p)]</b> 8 bytes	Data e hora compreendidas entre 01/01/0001 e 31/12/9999 com precisão de até 100 nanossegundos, dependendo do valor de <b>p</b> , que representa a quantidade de algarismos na fração de segundo. Omitindo <b>p</b> , o valor default será 7.
<b>date</b> 3 bytes	Data compreendida entre 01/01/0001 e 31/12/9999, com precisão de 1 dia.
<b>time[(p)]</b> 5 bytes	Hora no intervalo de 00:00:00.0000000 a 23.59.59.9999999. O parâmetro <b>p</b> indica a quantidade de dígitos na fração de segundo.
<b>Datetimeoffset[(p)]</b>	Data e hora compreendidas entre 01/01/0001 e 31/12/9999 com precisão de até 100 nanossegundos e com indicação do fuso horário, cujo intervalo pode variar de -14:00 a +14:00. O parâmetro <b>p</b> indica a quantidade de dígitos na fração de segundo.

## 2.4.4. Strings de caracteres ANSI

É chamada de **string** uma sequência de caracteres. No padrão ANSI, cada caractere é armazenado em 1 byte, o que permite a codificação de até 256 caracteres.

A tabela a seguir descreve alguns dos tipos de dados que fazem parte dessa categoria:

Nome	Descrição
<b>char(&lt;n&gt;)</b>	Comprimento fixo de no máximo 8.000 caracteres no padrão ANSI. Cada caractere é armazenado em 1 byte.
<b>varchar(&lt;n&gt;)</b>	Comprimento variável de no máximo 8.000 caracteres no padrão ANSI. Cada caractere é armazenado em 1 byte.
<b>text ou varchar(max)</b>	Comprimento variável de no máximo $2^{31} - 1$ (2,147,483,647) caracteres no padrão ANSI. Cada caractere é armazenado em 1 byte.

Em que:

- **<n>**: Representa a quantidade máxima de caracteres que poderemos armazenar. Cada caractere ocupa 1 byte.

 É recomendável a utilização do tipo **varchar(max)** em vez do tipo **text**. Esse último será removido em versões futuras do SQL Server. No caso de aplicações que já o utilizam, é indicado realizar a substituição pelo tipo recomendado. Ao utilizarmos **max** para **varchar**, estamos ampliando sua capacidade de armazenamento para 2 GB, aproximadamente.

## 2.4.5. Strings de caracteres Unicode

Em strings Unicode, cada caractere é armazenado em 2 bytes, o que amplia a quantidade de caracteres possíveis para mais de 65.000.

A tabela a seguir descreve alguns dos tipos de dados que fazem parte dessa categoria:

Nome	Descrição
<b>nchar(&lt;n&gt;)</b>	Comprimento fixo de no máximo 4.000 caracteres Unicode.
<b>nvarchar(&lt;n&gt;)</b>	Comprimento variável de no máximo 4.000 caracteres Unicode.
<b>ntext ou nvarchar(max)</b>	Comprimento variável de no máximo $2^{30} - 1$ (1.073.741.823) caracteres Unicode.

Em que:

- <n>: Representa a quantidade máxima de caracteres que poderemos armazenar. Cada caractere ocupa 2 bytes. Essa quantidade de 2 bytes é destinada a países cuja quantidade de caracteres utilizados é muito grande, como Japão e China.

Tanto no padrão ANSI quanto no UNICODE, existe uma tabela (ASCII) que codifica todos os caracteres. Essa tabela é usada para converter o caractere no seu código, quando gravamos, e para converter o código no caractere, quando lemos.

## 2.4.6. Strings binárias

No caso das strings binárias, não existe uma tabela para converter os caracteres, você interpreta os bits de cada byte de acordo com uma regra sua.

A tabela a seguir descreve alguns dos tipos de dados que fazem parte dessa categoria:

Nome	Descrição
<b>binary(&lt;n&gt;)</b>	Dado binário com comprimento fixo de, no máximo, 8.000 bytes.
<b>varbinary(&lt;n&gt;)</b>	Dado binário com comprimento variável de, no máximo, 8.000 bytes.
<b>image ou varbinary(max)</b>	Dado binário com comprimento variável de, no máximo, $2^{31} - 1$ (2,147,483,647) bytes.

 Os tipos **image** e **varbinary(max)** são muito usados para importar arquivos binários para dentro do banco de dados. Imagens, sons ou qualquer outro tipo de documento podem ser gravados em um campo desses tipos. É recomendável a utilização do tipo **varbinary(max)** em vez do tipo **image**. Esse último será removido em versões futuras do SQL Server. No caso de aplicações que já o utilizam, é indicado realizar a substituição pelo tipo recomendado.

## 2.4.7. Outros tipos de dados

Essa categoria inclui tipos de dados especiais, cuja utilização é específica e restrita a certas situações. A tabela adiante descreve alguns desses tipos:

Nome	Descrição
<b>table</b>	Serve para definir um dado tabular, composto de linhas e colunas, assim como uma tabela.
<b>cursor</b>	Serve para percorrer as linhas de um dado tabular.
<b>sql_variant</b>	Um tipo de dado que armazena valores de vários tipos suportados pelo SQL Server, exceto os seguintes: <b>text</b> , <b>ntext</b> , <b>timestamp</b> e <b>sql_variant</b> .
<b>Timestamp ou RowVersion</b>	Número hexadecimal sequencial gerado automaticamente.
<b>uniqueidentifier</b>	Globally Unique Identifier (GUID), também conhecido como Identificador Único Global ou Identificador Único Universal. É um número hexadecimal de 16 bytes semelhante a 64261228-50A9-467C-85C5-D73C51A914F1.
<b>XML</b>	Armazena dados no formato XML.
<b>Hierarchyid</b>	Posição de uma hierarquia.
<b>Geography</b>	Representa dados de coordenadas terrestres.
<b>Geometry</b>	Representação de coordenadas euclidianas.

## 2.5. Campo de autonumeração (IDENTITY)

A coluna de identidade, ou campo de autonumeração, é definida pela propriedade **IDENTITY**. Ao atribuirmos essa propriedade a uma coluna, o SQL Server cria números em sequência para linhas que forem posteriormente inseridas na tabela em que a coluna de identidade está localizada.

É importante saber que uma tabela pode ter apenas uma coluna do tipo identidade e que não é possível inserir ou alterar seu valor, que é gerado automaticamente pelo SQL-Server. Veja um exemplo:

```
DROP TABLE ALUNOS;
GO
CREATE TABLE ALUNOS
(
    NUM_ALUNO           INT IDENTITY,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO    DATETIME,
    IDADE               TINYINT,
    E_MAIL              VARCHAR(50),
    FONE_RES            CHAR(8),
    FONE_COM            CHAR(8),
    FAX                 CHAR(8),
    CELULAR             CHAR(9),
    PROFISSAO           VARCHAR(40),
    EMPRESA             VARCHAR(50) );
```

## 2.6. Constraints

As constraints são objetos utilizados para impor restrições e aplicar validações aos campos de uma tabela ou à forma como duas tabelas se relacionam. Podem ser definidas no momento da criação da tabela ou posteriormente, utilizando o comando **ALTER TABLE**.

## 2.6.1. Nulabilidade

Além dos valores padrão, também é possível atribuir valores nulos a uma coluna, o que significa que ela não terá valor algum. O NULL (nulo) não corresponde a nenhum dado, não é vazio ou zero, é nulo. Ao criarmos uma coluna em uma tabela, podemos acrescentar o atributo NULL (que já é padrão), para que aceite valores nulos, ou então NOT NULL, quando não queremos que determinada coluna aceite valores nulos. Exemplo:

```
CREATE TABLE ALUNOS  
(  CODIGO      INT      NOT NULL,  
  NOME        VARCHAR(30) NOT NULL,  
  E_MAIL      VARCHAR(100) NULL  );
```

## 2.6.2. Tipos de constraints

São diversos os tipos de constraints que podem ser criados: primary key, unique, check, default e foreign key. Adiante, cada uma das constraints será descrita.

### 2.6.2.1. PRIMARY KEY (chave primária)

A chave primária identifica, de forma única, cada uma das linhas de uma tabela. Pode ser formada por apenas uma coluna ou por combinação de duas ou mais colunas. A informação definida como chave primária de uma tabela não pode ser duplicada dentro dessa tabela.

- **Exemplos**

Tabela CLIENTES

Código do Cliente

Tabela PRODUTOS

Código do Produto

As colunas que formam a chave primária não podem aceitar valores nulos e devem ter o atributo NOT NULL.

- **Comando**

```
CREATE TABLE tabela
(
    CAMPO_PK          tipo NOT NULL,
    ...,
    ...,
    CONSTRAINT NomeChavePrimária PRIMARY KEY (CAMPO_PK) )
```



Onde usamos o termo **CAMPO\_PK** na criação da **PRIMARY KEY**, também poderá ser uma combinação de campos separados por vírgula.

O comando será o seguinte, depois de criada a tabela:

```
ALTER TABLE tabela ADD
CONSTRAINT NomeChavePrimária PRIMARY KEY (CAMPO_PK)
```

A convenção para dar nome a uma constraint do tipo chave primária é **PK\_NomeTabela**, ou seja, **PK** (abreviação de **PRIMARY KEY**) seguido do nome da tabela para a qual estamos criando a chave primária.

## 2.6.2.2.UNIQUE

Além do(s) campo(s) que forma(m) a **PRIMARY KEY**, pode ocorrer de termos outras colunas que não possam aceitar dados em duplicidade. Nesse caso, usaremos a constraint **UNIQUE**.

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

- **Exemplos**

Na tabela CLIENTES

Campos CPF, RG e E-mail

Na tabela TIPOS\_PRODUTO

Descrição do tipo de produto

Na tabela UNIDADES\_MEDIDA

Descrição da unidade de medida

- **Comando**

```
CREATE TABLE tabela
(
    ...
    CAMPO_UNICO          tipo NOT NULL,
    ...,
    ...,
    CONSTRAINT NomeUnique UNIQUE (CAMPO_UNICO) )
```

Onde usamos o termo **CAMPO\_UNICO** na criação da **UNIQUE**, também poderá ser uma combinação de campos separados por vírgula.

O comando será o seguinte, depois de criada a tabela:

```
ALTER TABLE tabela ADD
CONSTRAINT NomeUnique UNIQUE (CAMPO_UNICO)
```

O nome da constraint deve ser sugestivo, como: **UQ\_NomeTabela\_NomeCampoUnico**.

## 2.6.2.3.CHECK

Nesse tipo de constraint, criamos uma condição (semelhante às usadas com a cláusula WHERE) para definir a integridade de um ou mais campos de uma tabela.

- **Exemplo**

Tabela CLIENTES

Data Nascimento < Data Atual  
Data inclusão <= Data Atual  
Data Nascimento < Data Inclusão  
Preço Venda >= 0  
Preço Compra >= 0  
Preço Venda >= Preço Compra  
Data Inclusão <= Data Atual

Tabela PRODUTOS

- **Comando**

```
CREATE TABLE tabela
(
    ...,
    ...,
    CONSTRAINT NomeCheck CHECK (Condição) )
```

O comando será o seguinte, depois de criada a tabela:

```
ALTER TABLE tabela ADD
CONSTRAINT NomeCheck CHECK (Condição)
```

O nome da constraint deve ser sugestivo, como **CH\_NomeTabela\_DescrCondicao**.

## 2.6.2.4. DEFAULT

Normalmente, quando inserimos dados em uma tabela, as colunas para as quais não fornecemos valor, terão como conteúdo **NULL**. Ao definirmos uma constraint do tipo **DEFAULT** para uma determinada coluna, este valor será atribuído à ela quando o **INSERT** não fornecer valor.

- **Exemplo**

Tabela PESSOAS

Data Inclusão DEFAULT Data Atual  
SexoDEFAULT ‘M’

- **Comando**

```
CREATE TABLE tabela
(
    ...,
    CAMPO_DEFAULT tipo [NOT NULL] DEFAULT valorDefault,
    ...,
)
```

O comando será o seguinte, depois de criada a tabela:

```
ALTER TABLE tabela ADD
CONSTRAINT NomeDefault DEFAULT (valorDefault) FOR CAMPO_DEFAULT
```

## 2.6.2.5. FOREIGN KEY (chave estrangeira)

É chamado de chave estrangeira ou FOREIGN KEY o campo da tabela DETALHE que se relaciona com a chave primária da tabela MESTRE.

	COD_DEP...	DEPTO
1	1	PESSOAL
2	2	C.P.D.
3	3	CONTROLE DE ESTOQUE
4	4	COMPRAS
5	5	PRODUCAO
6	6	DIRETORIA
7	7	TELEMARKETING
8	8	FINANCEIRO
9	9	RECURSOS HUMANOS
10	10	TREINAMENTO
11	11	PRESIDENCIA
12	12	PORTARIA
13	13	CONTROLADORIA
14	14	P.C.P.

Tabela Mestre  
DEPARTAMENTOS

	CODFUN	NOME	NUM_DEPE...	DATA_NASCIMENTO	COD_DEP...
1	1	OLAVO TRINDADE	1	1950-06-06 00:00:00.000	4
2	2	JOSE REIS	6	1952-10-09 00:00:00.000	2
3	3	MARCELO SOARES	1	1950-06-06 00:00:00.000	5
4	4	PAULO CESAR JUNIOR	2	1952-03-19 00:00:00.000	8
5	5	JOAO LIMA MACHADO DA SILVA	2	1955-10-30 00:00:00.000	4
6	7	CARLOS ALBERTO SILVA	0	1961-07-06 00:00:00.000	11
7	8	ELIANE PEREIRA	0	1955-01-14 00:00:00.000	6
8	9	RUDGE RAMOS SANTANA DA PENHA	3	1961-07-22 00:00:00.000	2
9	10	MARIA CARMEM	0	1954-03-14 00:00:00.000	5
10	11	FERNANDO OLIVEIRA	0	1954-07-06 00:00:00.000	3
11	12	JOAO ROBERTO OLIVEIRA	0	1953-05-18 00:00:00.000	4
12	13	OSMAR PRADO	0	1953-10-27 00:00:00.000	5

Tabela Detalhe  
EMPREGADOS

Observando a figura, notamos que o campo **COD\_DEPTO** da tabela **EMPREGADOS** (detalhe) é chave estrangeira, pois se relaciona com **COD\_DEPTO** (chave primária) da tabela **DEPARTAMENTOS** (mestre).

Podemos ter essa mesma estrutura sem termos criado uma chave estrangeira, embora a chave estrangeira garanta a integridade referencial dos dados, ou seja, com a chave estrangeira, será impossível existir um registro de **EMPREGADOS** com um **COD\_DEPTO** inexistente em **DEPARTAMENTOS**. Quando procurarmos o **COD\_DEPTO** do empregado em **DEPARTAMENTOS**, sempre encontraremos correspondência.

**Se a tabela mestre não possuir uma chave primária, não será possível criar uma chave estrangeira apontando para ela.**

Para criarmos uma chave estrangeira, temos que considerar as seguintes informações envolvidas:

- A tabela detalhe;
- O campo da tabela detalhe que se relaciona com a tabela mestre;
- A tabela mestre;
- O campo chave primária da tabela mestre.

O comando para a criação de uma chave estrangeira é o seguinte:

```
CREATE TABLE tabelaDetalhe
(
    ...
    CAMPO_FK tipo [NOT NULL],
    ...
    CONSTRAINT NomeChaveEstrangeira FOREIGN KEY(CAMPO_FK)
        REFERENCES tabelaMestre(CAMPO_PK_TABELA_MESTRE)
)
```

Ou utilize o seguinte comando depois de criada a tabela:

```
ALTER TABLE tabelaDetalhe ADD
    CONSTRAINT NomeChaveEstrangeira FOREIGN KEY(CAMPO_FK)
        REFERENCES tabelaMestre(CAMPO_PK_TABELA_MESTRE)
```

## 2.6.3. Criando constraints

A seguir, veremos como criar constraints com o uso de **CREATE TABLE** e **ALTER TABLE**, bem como criá-las graficamente a partir da interface do SQL Server Management Studio.

### 2.6.3.1. Criando constraints com CREATE TABLE

A seguir, temos um exemplo de criação de constraints com o uso de **CREATE TABLE**:

1. Primeiramente, crie o banco de dados **TESTE\_CONSTRAINT**:

```
CREATE DATABASE TESTE_CONSTRAINT;
GO
USE TESTE_CONSTRAINT;
```

2. Agora, crie a tabela **TIPO\_PRODUTO** com os tipos (categorias) de produto:

```
-- Tabela de tipos (categorias) de produto
CREATE TABLE TIPO_PRODUTO
(
    COD_TIPO          INT IDENTITY NOT NULL,
    TIPO              VARCHAR(30) NOT NULL,
    -- Convenção de nome: PK_NomeTabela
    CONSTRAINT PK_TIPO_PRODUTO PRIMARY KEY (COD_TIPO),
    -- Convenção De nome: UQ_NomeTabela_NomeCampo
    CONSTRAINT UQ_TIPO_PRODUTO_TIPO UNIQUE( TIPO ) );
```

3. Em seguida, teste a constraint **UNIQUE** criada:

```
-- Testando a constraint UNIQUE
INSERT TIPO_PRODUTO VALUES ('MOUSE');
INSERT TIPO_PRODUTO VALUES ('PEN-DRIVE');
INSERT TIPO_PRODUTO VALUES ('HARD DISK');
-- Vai dar erro porque viola a constraint UNIQUE
INSERT TIPO_PRODUTO VALUES ('HARD DISK');
```

## 4. O próximo passo é criar a tabela de produtos (**PRODUTOS**):

```
-- Tabela de produtos
CREATE TABLE PRODUTOS
(
    ID_PRODUTO      INT IDENTITY NOT NULL,
    DESCRICAO       VARCHAR(50),
    COD_TIPO        INT,
    PRECO_CUSTO     NUMERIC(10,2),
    PRECO_VENDA     NUMERIC(10,2),
    QTD_REAL        NUMERIC(10,2),
    QTD_MINIMA      NUMERIC(10,2),
    DATA_CADASTRO   DATETIME DEFAULT GETDATE(),
    SN_ATIVO        CHAR(1) DEFAULT 'S',
    CONSTRAINT PK_PRODUTOS PRIMARY KEY( ID_PRODUTO ),
    CONSTRAINT UQ_PRODUTOS_DESCRICAO UNIQUE( DESCRICAO ),

    CONSTRAINT CK_PRODUTOS_PRECOS
        CHECK( PRECO_VENDA >= PRECO_CUSTO ),
    CONSTRAINT CK_PRODUTOS_DATA_CAD
        CHECK( DATA_CADASTRO <= GETDATE() ),
    CONSTRAINT CK_PRODUTOS_SN_ATIVO
        CHECK( SN_ATIVO IN ('N','S') ),
    -- Convenção de nome: FK_TabelaDetalhe_TabelaMestre
    CONSTRAINT FK_PRODUTOS_TIPO_PRODUTO
        FOREIGN KEY (COD_TIPO)
        REFERENCES TIPO_PRODUTO (COD_TIPO) );
```

## 5. Veja um modelo para a criação de chave estrangeira:

```
-- Criação de chave estrangeira
-- CONSTRAINT FK_TabelaDetalhe_TabelaMestre
--     FOREIGN KEY (campoTabelaDetalhe)
--     REFERENCES TabelaMestre( campoPK_TabelaMestre )
```

# SQL 2014 - Módulo I

6. Feito isso, teste a constraint **DEFAULT** criada anteriormente. A sequência de código adiante gera os valores para os campos **DATA\_CADASTRO** e **SN\_ATIVO** que não foram mencionados no **INSERT**:

```
INSERT PRODUTOS
(DESCRICAO, COD_TIPO, PRECO_CUSTO, PRECO_VENDA,
 QTD_REAL, QTD_MINIMA)
VALUES ('TESTANDO INCLUSAO', 1, 10, 12, 10, 5 );
--  
SELECT * FROM PRODUTOS;
```

7. No código seguinte, teste a constraint **UNIQUE**:

```
-- Vai dar erro porque viola a constraint UNIQUE
INSERT PRODUTOS
(DESCRICAO, COD_TIPO, PRECO_CUSTO, PRECO_VENDA,
 QTD_REAL, QTD_MINIMA)
VALUES ('TESTANDO INCLUSAO', 10, 10, 12, 10, 5 );
```

8. No próximo código, teste a constraint **FOREIGN KEY**:

```
-- Vai dar erro porque viola a constraint FOREIGN KEY
INSERT PRODUTOS
(DESCRICAO, COD_TIPO, PRECO_CUSTO, PRECO_VENDA,
 QTD_REAL, QTD_MINIMA)
VALUES ('TESTANDO INCLUSAO 2', 10, 10, 12, 10, 5 );
```

9. Por fim, o código adiante testa a constraint **CHECK**:

```
-- Vai dar erro porque viola a constraint CHECK -
-- (CK_PRODUTOS_PRECOS)
INSERT PRODUTOS
(DESCRICAO, COD_TIPO, PRECO_CUSTO, PRECO_VENDA,
 QTD_REAL, QTD_MINIMA)
VALUES ('TESTANDO INCLUSAO 2', 1, 14, 12, 10, 5 );
```

### 2.6.3.2.Criando constraints com ALTER TABLE

O exemplo a seguir demonstra a criação de constraints utilizando **ALTER TABLE**:

```

USE TESTE_CONSTRAINT;

DROP TABLE PRODUTOS
GO
DROP TABLE TIPO_PRODUTO
GO
-- Criação da tabela TIPO_PRODUTO
CREATE TABLE TIPO_PRODUTO
(
    COD_TIPO      INT IDENTITY NOT NULL,
    TIPO          VARCHAR(30) NOT NULL
);
-- Criando as constraints com ALTER TABLE
ALTER TABLE TIPO_PRODUTO ADD
    CONSTRAINT PK_TIPO_PRODUTO PRIMARY KEY (COD_TIPO);

ALTER TABLE TIPO_PRODUTO ADD
    CONSTRAINT UQ_TIPO_PRODUTO_TIPO UNIQUE( TIPO );

-- Criando a tabela PRODUTOS
CREATE TABLE PRODUTOS
(
    ID_PRODUTO      INT IDENTITY NOT NULL,
    DESCRICAO       VARCHAR(50),
    COD_TIPO        INT,
    PRECO_CUSTO     NUMERIC(10,2),
    PRECO_VENDA     NUMERIC(10,2),
    QTD_REAL        NUMERIC(10,2),
    QTD_MINIMA      NUMERIC(10,2),
    DATA_CADASTRO   DATETIME,
    SN_ATIVO        CHAR(1)
);

-- Criando as constraints com ALTER TABLE
ALTER TABLE PRODUTOS ADD
    CONSTRAINT PK_PRODUTOS PRIMARY KEY( ID_PRODUTO );

ALTER TABLE PRODUTOS ADD
    CONSTRAINT UQ_PRODUTOS_DESCRICAO UNIQUE( DESCRICAO );

-- Criando várias constraints em um único ALTER TABLE
ALTER TABLE PRODUTOS ADD
    CONSTRAINT CK_PRODUTOS_PRECOS
        CHECK( PRECO_VENDA >= PRECO_CUSTO ),
    CONSTRAINT CK_PRODUTOS_DATA_CAD
        CHECK( DATA_CADASTRO <= GETDATE() ),
    CONSTRAINT CK_PRODUTOS_SN_ATIVO
        CHECK( SN_ATIVO IN ('N','S') ),
    CONSTRAINT FK_PRODUTOS_TIPO_PRODUTO
        FOREIGN KEY(COD_TIPO)
            REFERENCES TIPO_PRODUTO (COD_TIPO),
    CONSTRAINT DF_SN_ATIVO DEFAULT ('S') FOR SN_ATIVO,
    CONSTRAINT DF_DATA_CADASTRO DEFAULT (GETDATE())
        FOR DATA_CADASTRO;

```

## 2.6.3.3.Criando constraints graficamente

O exemplo a seguir demonstra a criação de constraints graficamente. Primeiramente, crie as tabelas **TIPO\_PRODUTO** e **PRODUTOS** no banco de dados **TESTE\_CONSTRAINT**:

```
USE TESTE_CONSTRAINT;

DROP TABLE PRODUTOS
GO
DROP TABLE TIPO_PRODUTO
GO
-- Criação da tabela TIPO_PRODUTO
CREATE TABLE TIPO_PRODUTO
(
    COD_TIPO          INT IDENTITY NOT NULL,
    TIPO              VARCHAR(30) NOT NULL
);

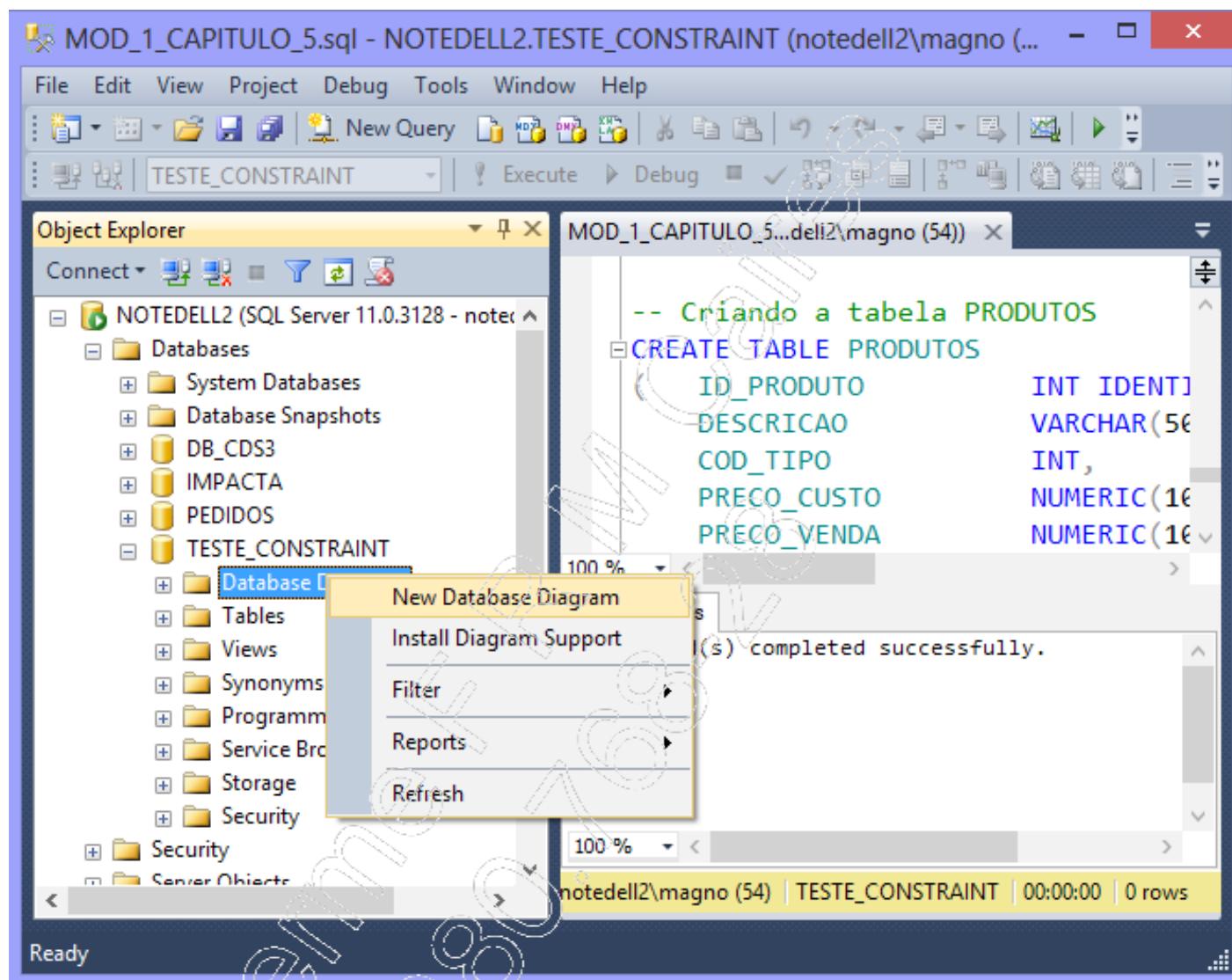
-- Criando a tabela PRODUTOS
CREATE TABLE PRODUTOS
(
    ID_PRODUTO        INT IDENTITY NOT NULL,
    DESCRICAO         VARCHAR(50),
    COD_TIPO          INT,
    PRECO_CUSTO       NUMERIC(10,2),
    PRECO_VENDA       NUMERIC(10,2),
    QTD_REAL          NUMERIC(10,2),
    QTD_MINIMA        NUMERIC(10,2),
    DATA_CADASTRO    DATETIME,
    SN_ATIVO          CHAR(1)  );

```

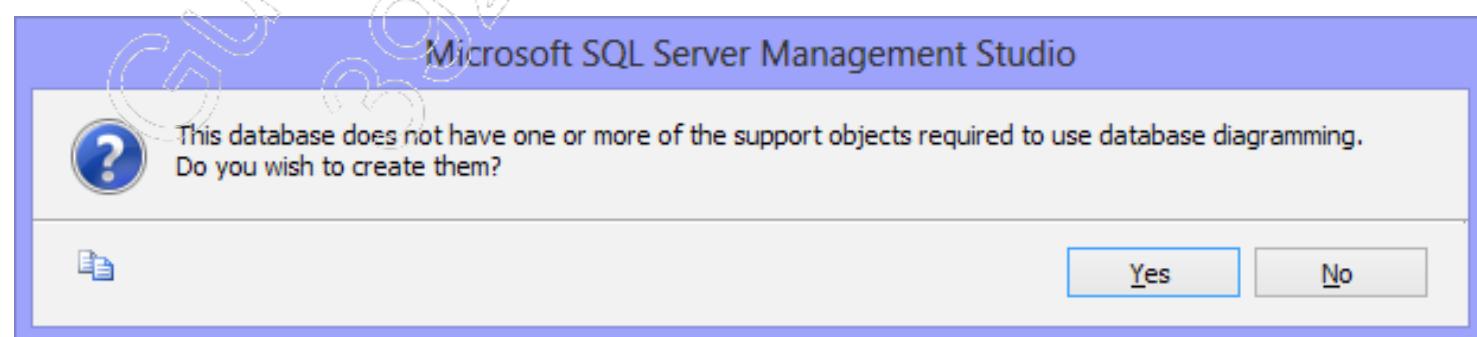
Depois, realize os seguintes passos:

1. No Object Explorer, selecione o banco de dados **TESTE\_CONSTRAINT** e abra os subitens do banco;

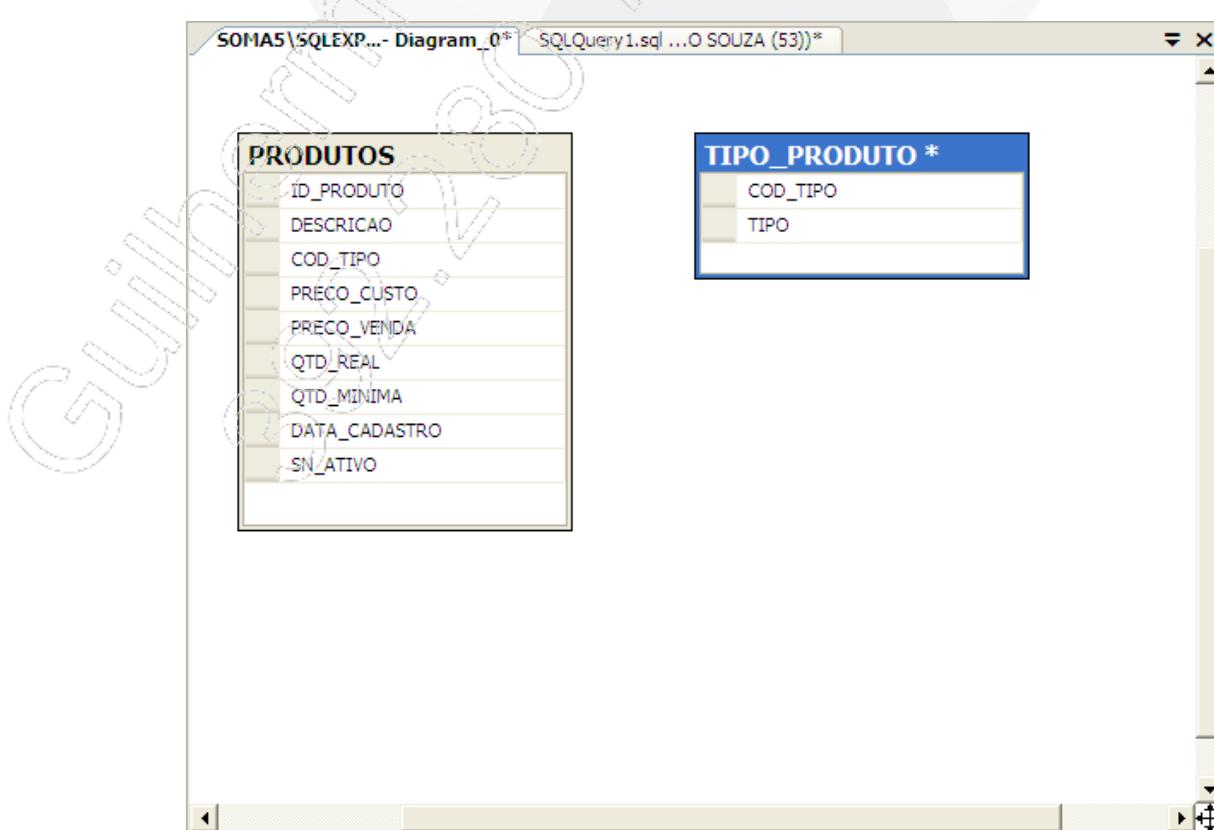
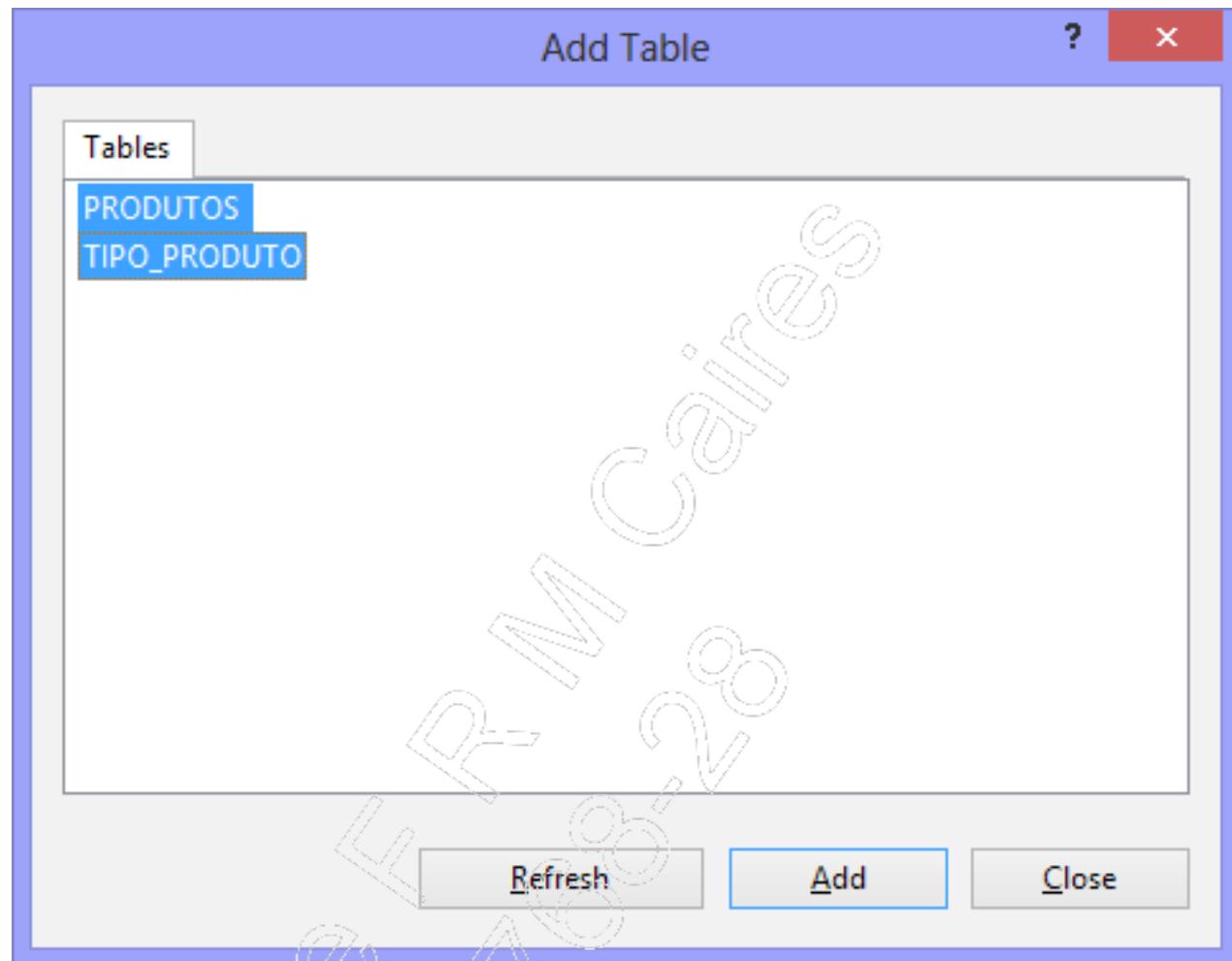
2. Clique com o botão direito do mouse no item **Database Diagrams** e selecione a opção **New Database Diagram**:



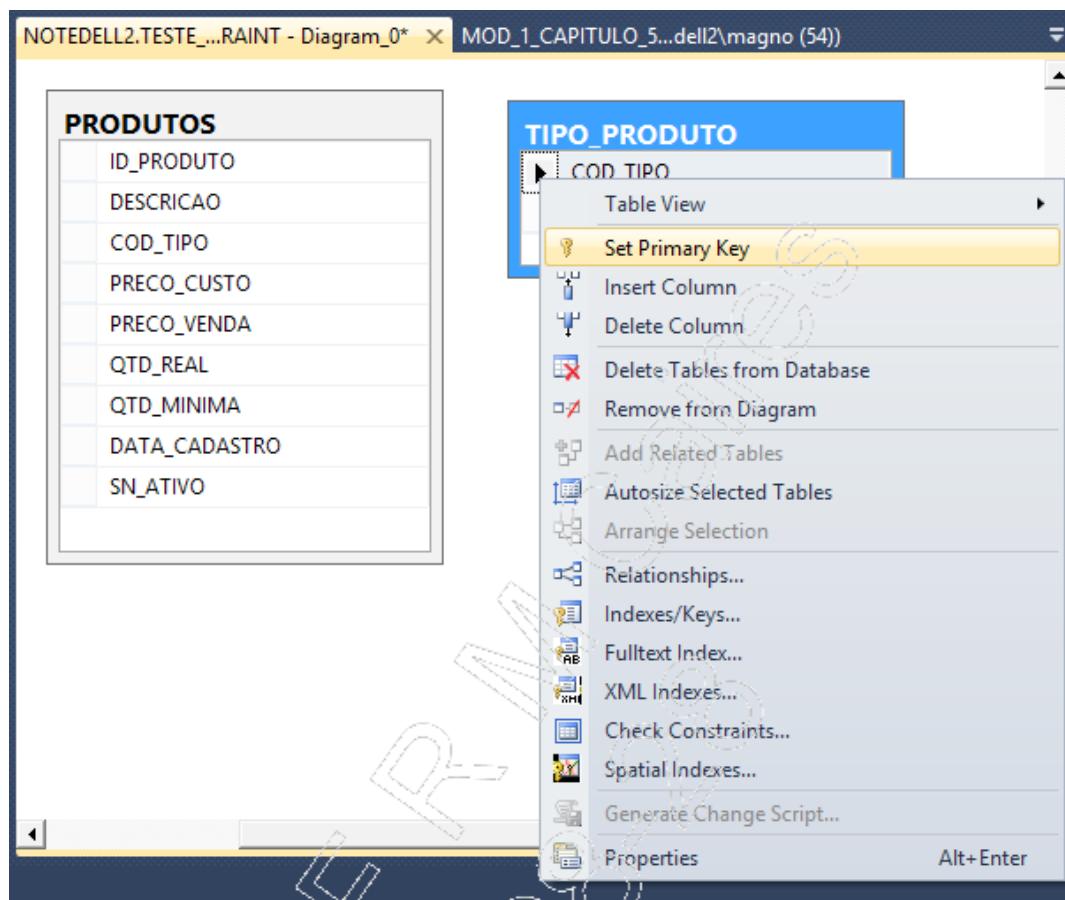
3. Na caixa de diálogo exibida, clique em **Yes (Sim)**:



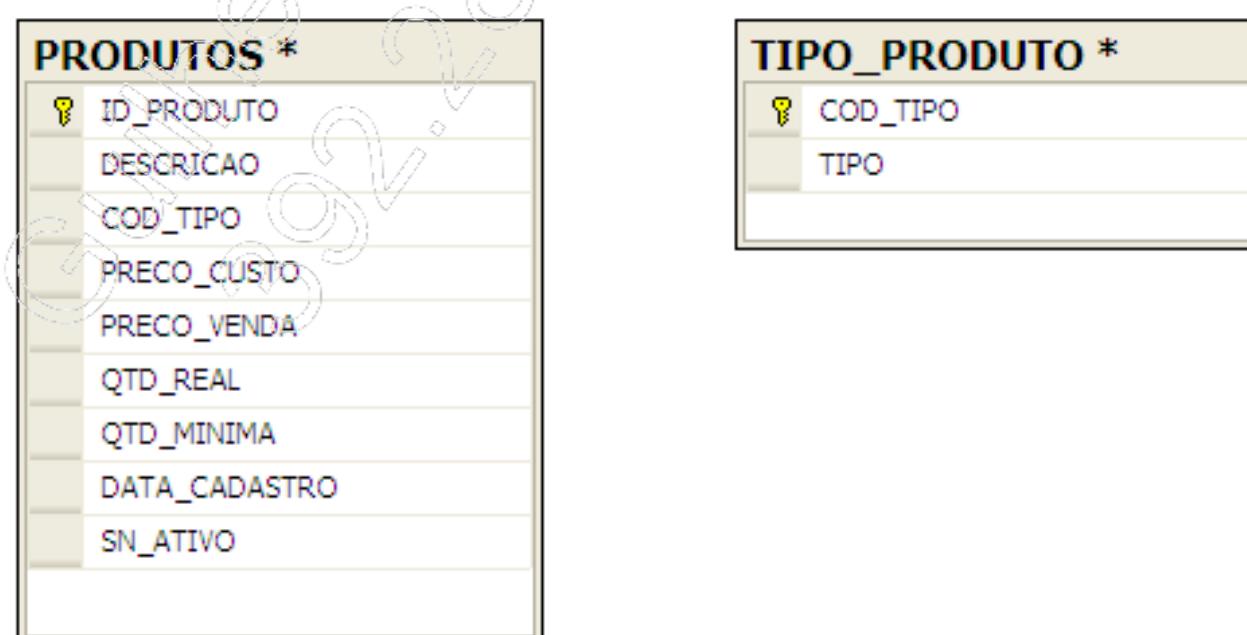
4. Uma janela com os nomes das tabelas existentes no banco de dados será exibida. Selecione as duas tabelas, clique em **Add (Adicionar)** e, depois, em **Close (Fechar)**. Note que as tabelas aparecerão na área de desenho do diagrama;



5. Clique com o botão direito do mouse no campo **COD\_TIPO** da tabela **TIPO\_PRODUTO** e selecione a opção **Set Primary Key (Definir Chave Primária)**;

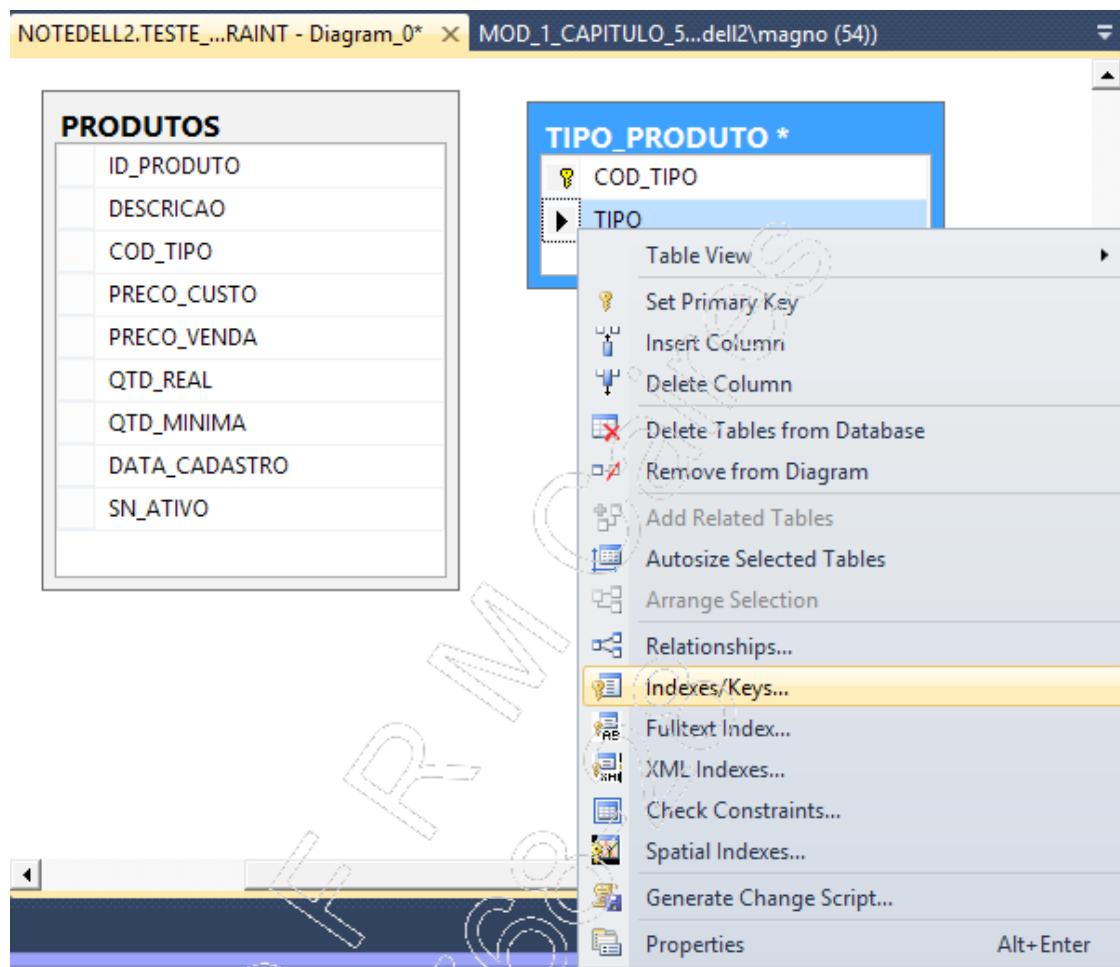


6. Clique com o botão direito do mouse no campo **ID\_PRODUTO** da tabela **PRODUTOS** e selecione a opção **Set Primary Key (Definir Chave Primária)**. Com isso, serão criadas as chaves primárias das duas tabelas;

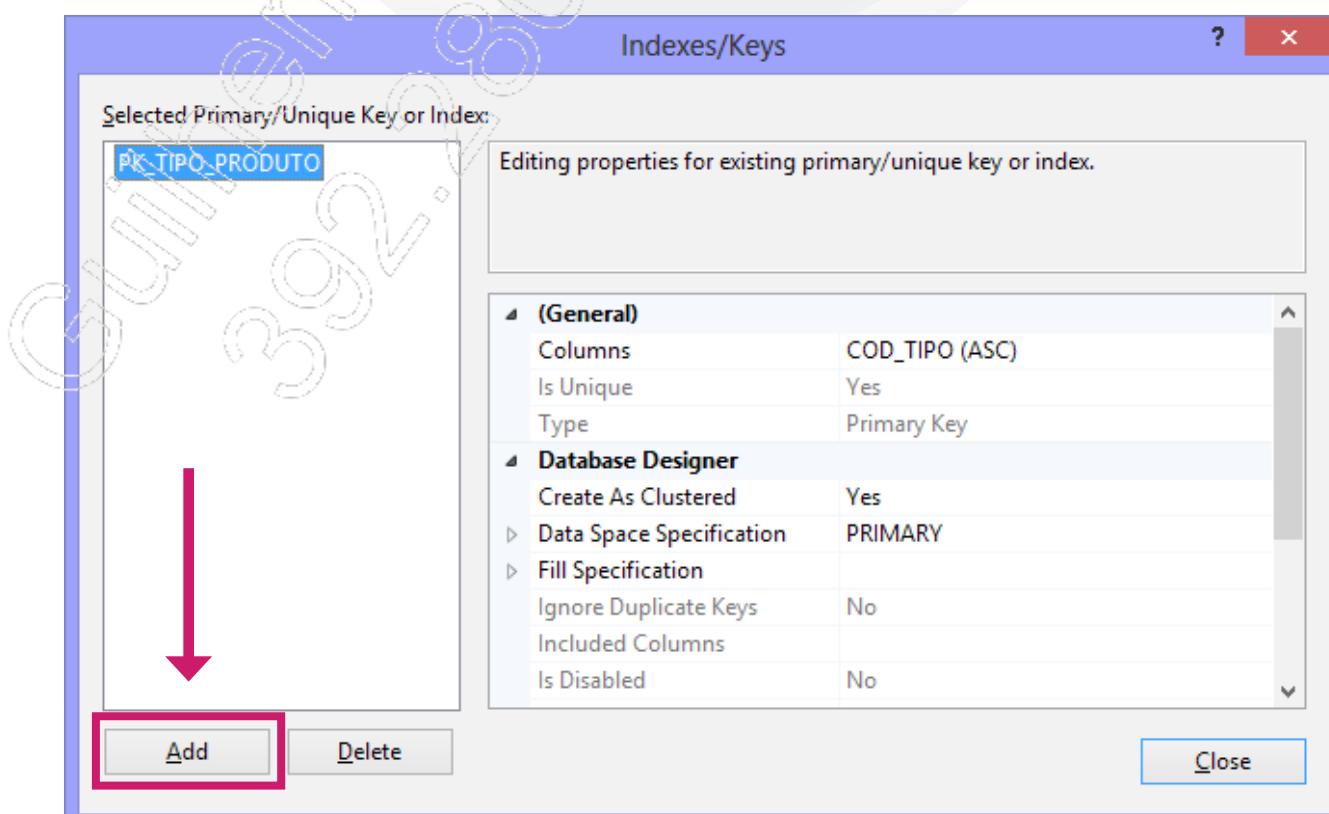


# SQL 2014 - Módulo I

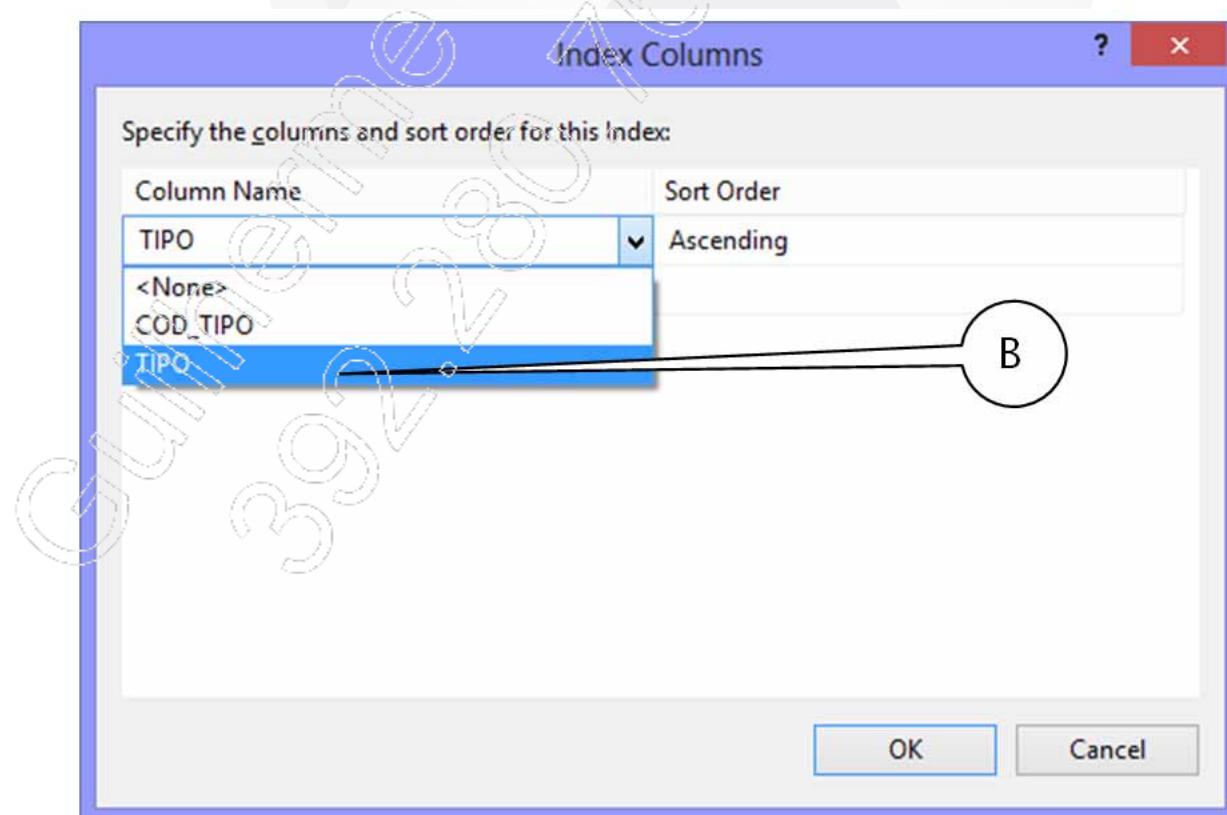
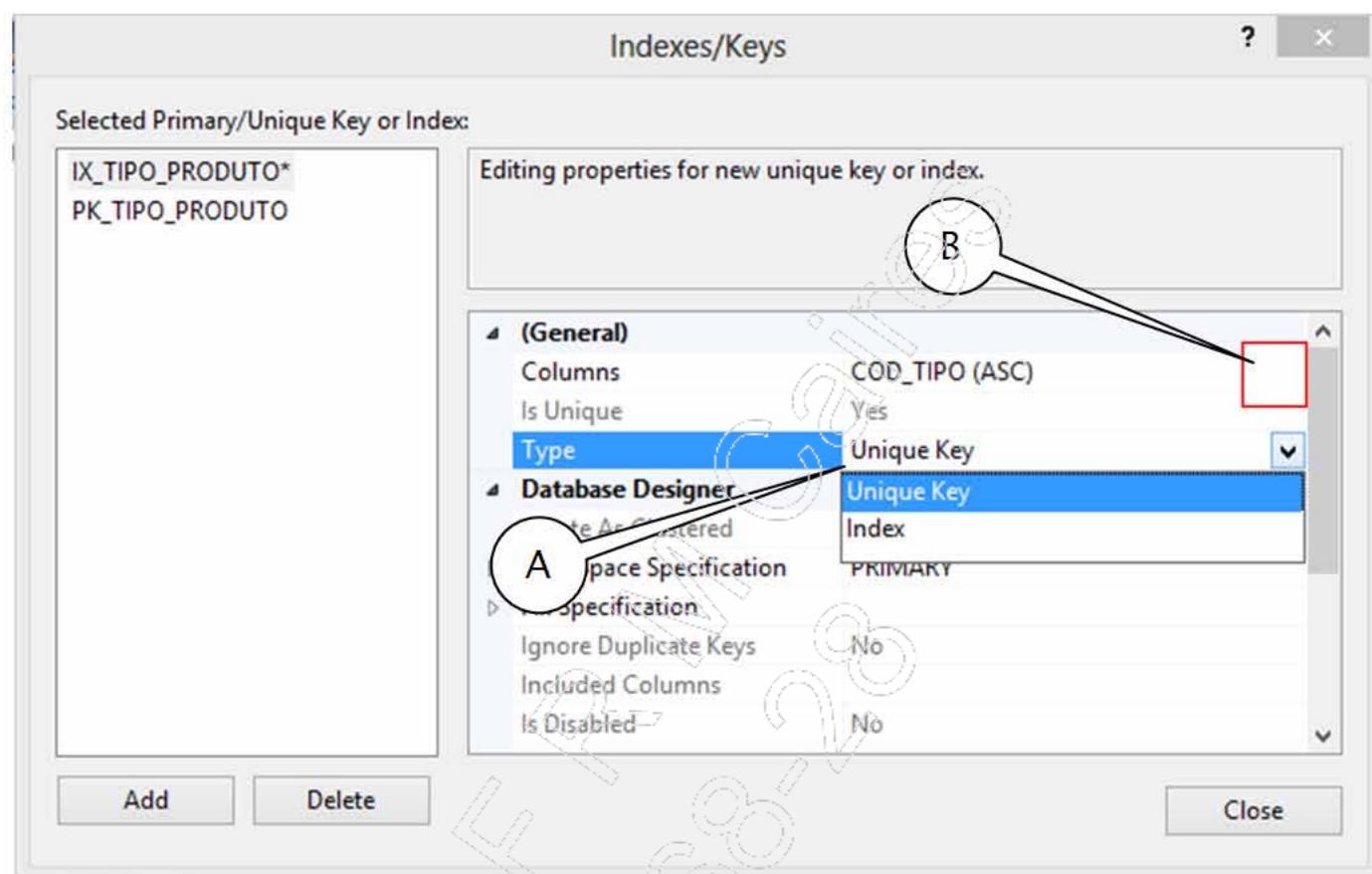
7. Para criar a chave única (**UNIQUE CONSTRAINT**), selecione a tabela **TIPO\_PRODUTO**, clique com o botão direito do mouse sobre ela e clique na opção **Indexes/Keys...** (Índices/Chaves...) para abrir a caixa de diálogo **Indexes/Keys**:



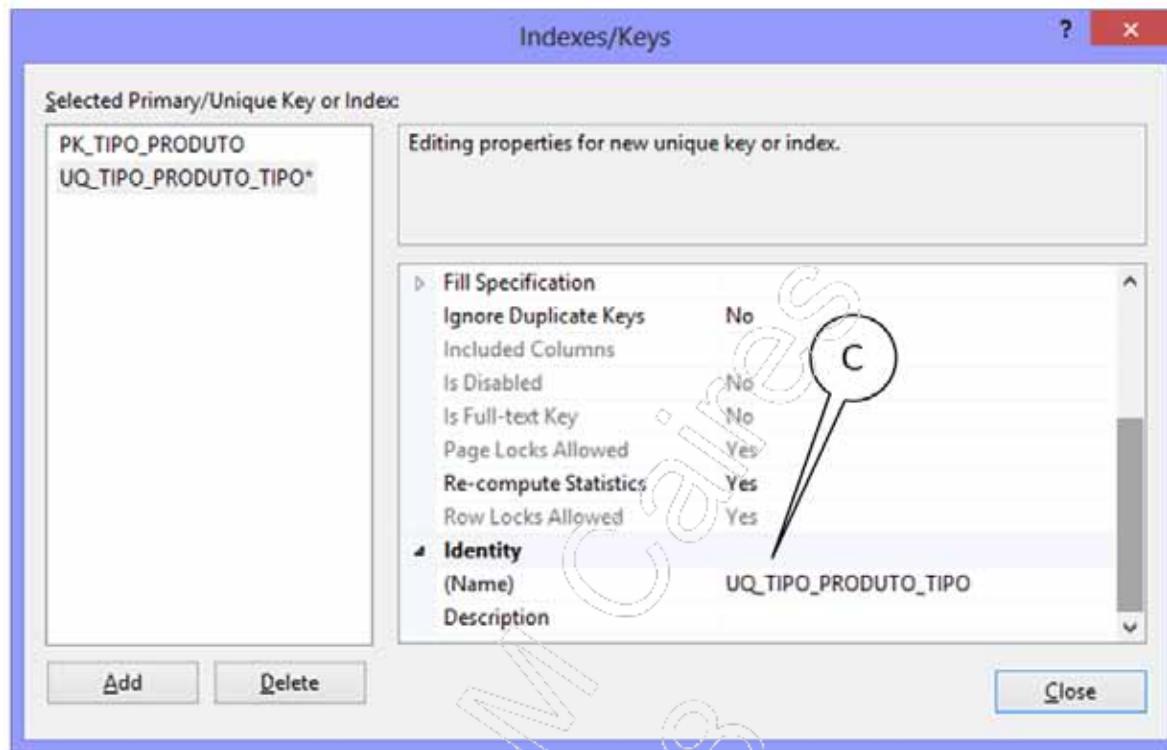
8. Clique no botão Add (Adicionar);



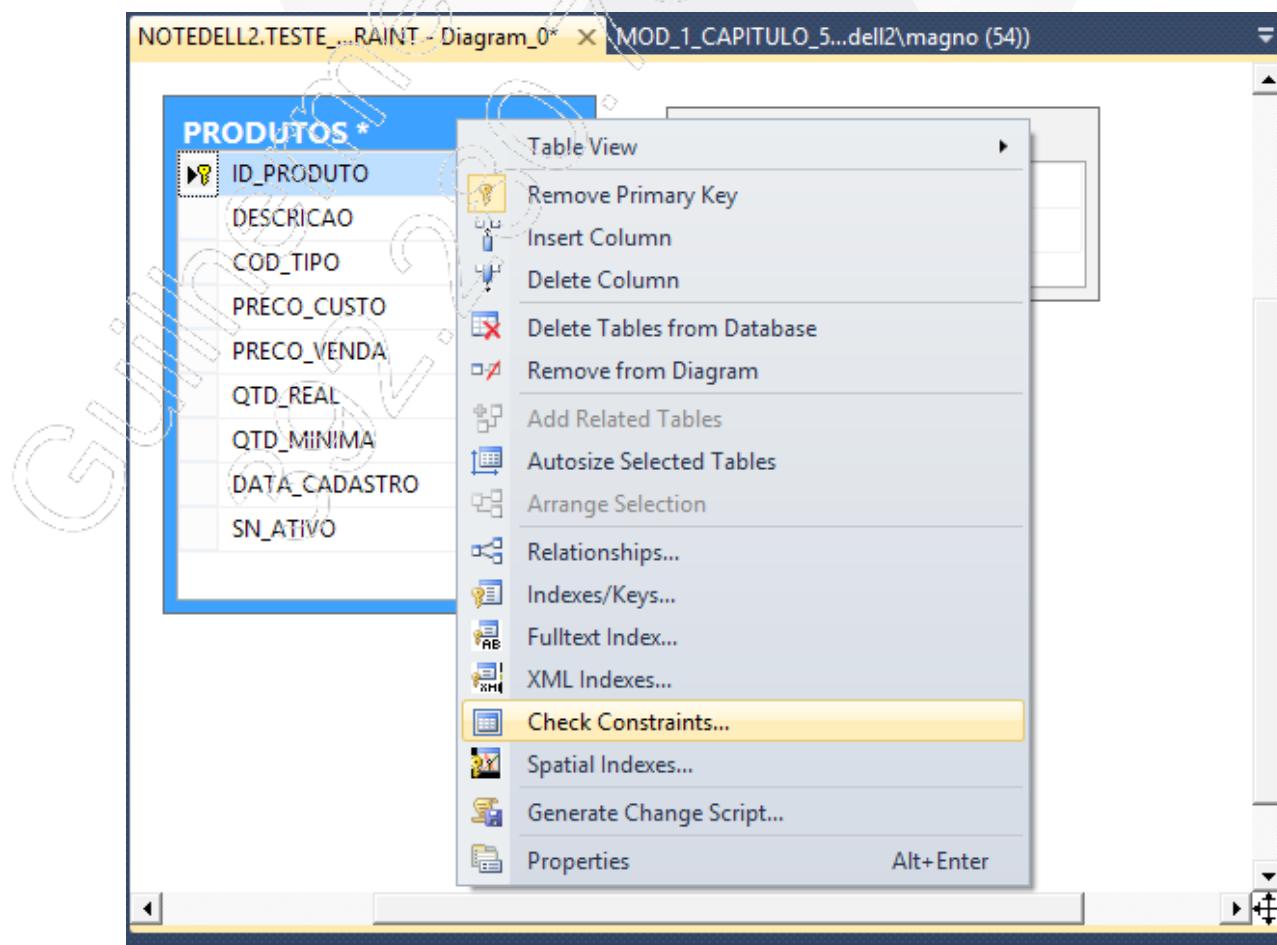
9. Altere as propriedades (A) **Type (Tipo)** para **Unique Key (Chave Exclusiva)** e (B) **Columns (Colunas)** para **TIPO**;



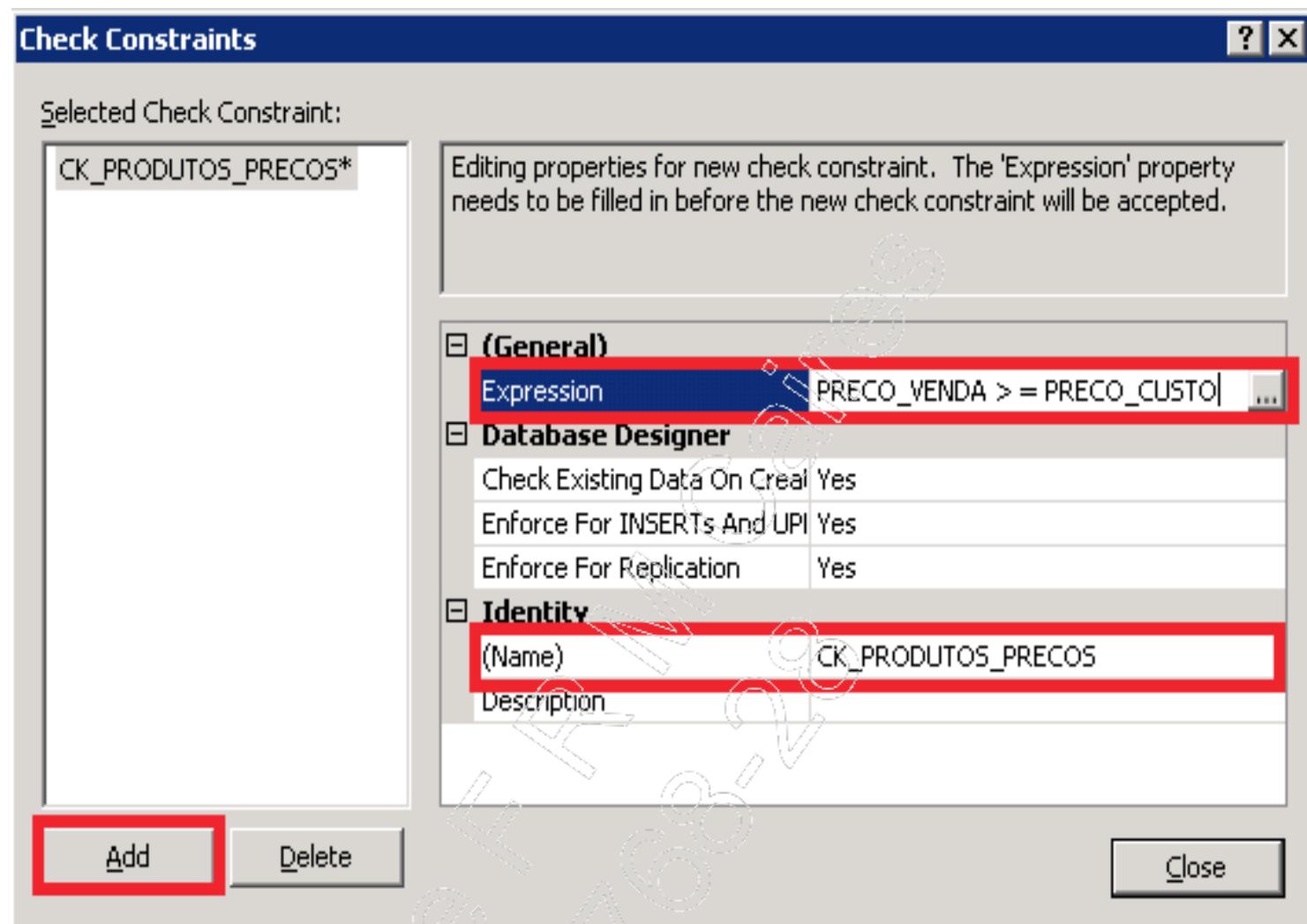
10. Altere as propriedades (C) Name (Nome) para UQ\_TIPO\_PRODUTO\_TIPO. Depois, clique em Close (Fechar);



11. Para criar as constraints CHECK, clique com o botão direito do mouse sobre a tabela PRODUTOS e selecione a opção Check Constraints... (Restrições de Verificação...);



12. Na seguinte caixa de diálogo que é aberta, clique no botão Add e depois altere as seguintes propriedades:

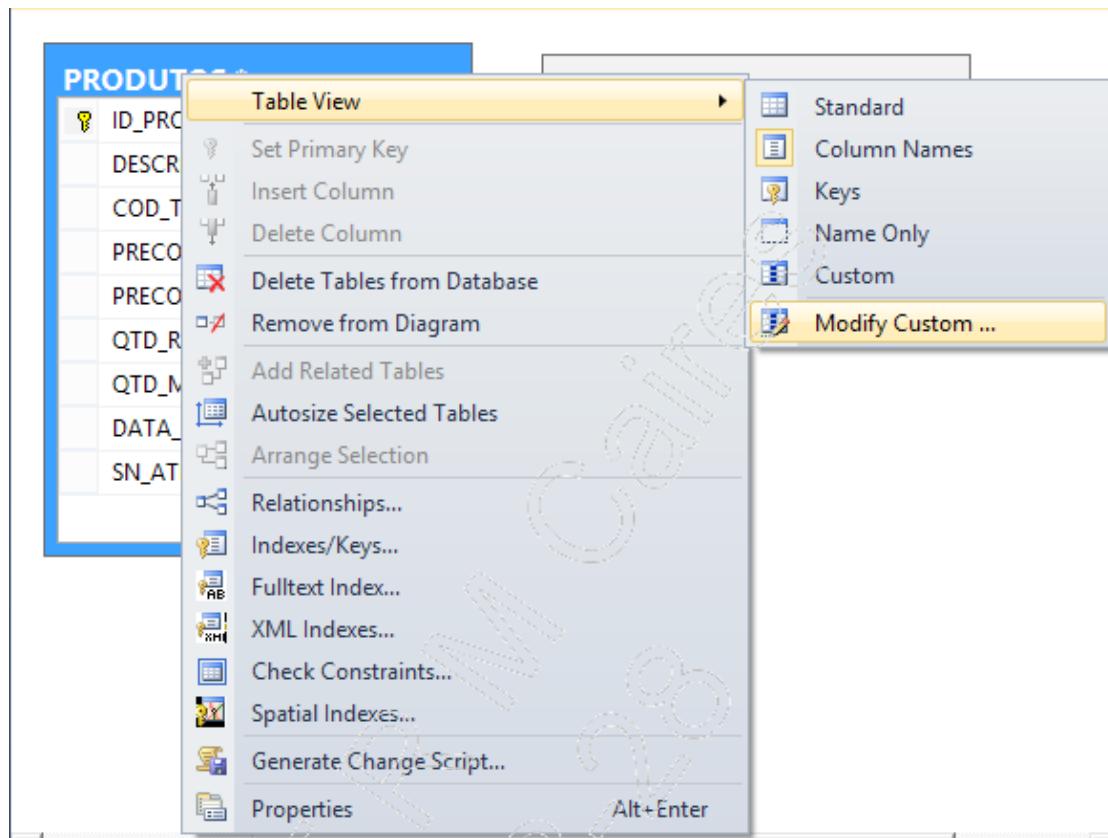


Add

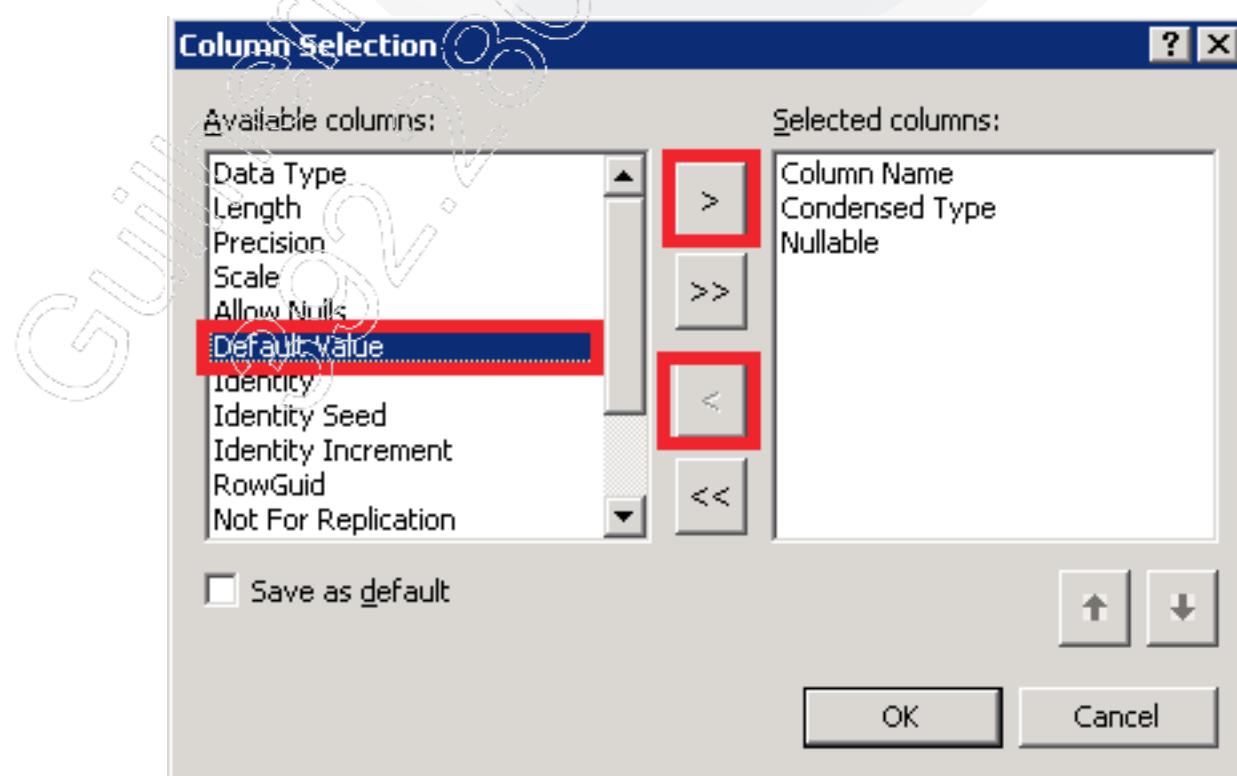
Delete

Close

13. Para definir os valores default (padrão) de cada campo, clique com o botão direito do mouse na tabela **PRODUTOS**, selecione **Table View (Exibição da Tabela...)** e depois **Modify Custom View (Modificar Personalização)**;



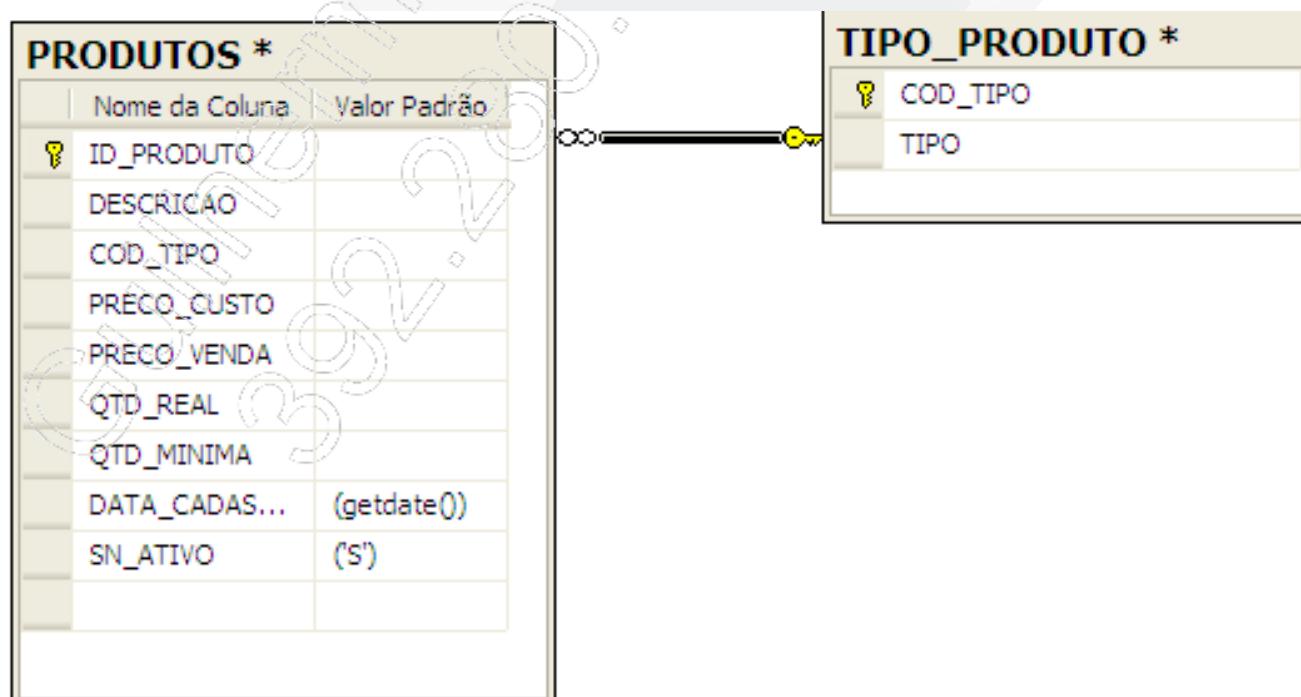
14. Na janela aberta, selecione as colunas **Condensed Type** e **Nullable** e remova-as clicando no botão <. Em seguida, adicione a coluna **Default Value** clicando no botão >. Feche a janela clicando em **OK**;



15. Para exibir os valores padrão, clique com o botão direito do mouse sobre a tabela **PRODUTOS**, selecione **Table View** e clique na opção **Custom**. Por fim, informe o valor padrão ao lado do nome do campo **DATA\_CADASTRO** e **SN\_ATIVO**;

	Column Name	Default Value
PK	ID_PRODUTO	
	DESCRICAO	
	COD_TIPO	
	PRECO_CUSTO	
	PRECO_VENDA	
	QTD_REAL	
	QTD_MINIMA	
	DATA_CADASTRO	GETDATE()
	SN_ATIVO	'S'

16. Para definir a chave estrangeira, selecione o campo **COD\_TIPO** da tabela **PRODUTOS** e arraste-o até o campo **COD\_TIPO** da tabela **TIPO\_PRODUTO**.



### 2.7. Normalização de dados

O processo de organizar dados e eliminar informações redundantes de um banco de dados é denominado normalização.

A normalização envolve a tarefa de criar as tabelas e definir os seus relacionamentos. O relacionamento entre as tabelas é criado de acordo com regras que visam à proteção dos dados e à eliminação de dados repetidos. Essas regras são denominadas **normal forms** ou formas normais.

A normalização apresenta grandes vantagens:

- Elimina dados repetidos, o que torna o banco mais compacto;
- Garante o armazenamento dos dados de forma lógica;
- Maior velocidade dos processos de classificar e indexar, já que as tabelas possuem uma quantidade menor de colunas;
- Permite o agrupamento de índices conforme a quantidade de tabelas aumenta. Além disso, reduz o número de índices por tabela. Dessa forma, permite melhor performance de atualização do banco de dados.

Entretanto, o processo de normalização pode aumentar a quantidade de tabelas e, consequentemente, a complexidade das associações exigidas entre elas para que os dados desejados sejam obtidos. Isso pode acabar prejudicando o desempenho da aplicação.

Outro aspecto negativo da normalização é que as tabelas, em vez de dados reais, podem conter códigos. Nesse caso, será necessário recorrer à tabela de pesquisa para obter os valores necessários. A normalização também pode dificultar a consulta ao modelo de dados.

## 2.7.1. Regras de normalização

A normalização inclui 3 regras principais: **first normal form (1NF)**, **second normal form (2NF)** e **third normal form (3NF)**.

Consideramos que um banco de dados está no **first normal form** quando a primeira regra (**1NF**) é cumprida. Se as três regras forem cumpridas, o banco de dados estará no **third normal form**.

É possível atingir outros níveis de normalização (4NF e 5NF), entretanto, o 3NF é considerado o nível mais alto requerido pela maior parte das aplicações.

Veja quais as regras que devem ser cumpridas para atingir cada nível de normalização:

- **First Normal Form (1NF)**

Para que um banco de dados esteja nesse nível de normalização, cada coluna deve conter um único valor e cada linha deve abranger as mesmas colunas. A fim de atendermos a esses aspectos, os conjuntos que se repetem nas tabelas individuais devem ser eliminados. Além disso, devemos criar uma tabela separada para cada conjunto de dados relacionados e identificar cada um deles com uma chave primária.

Uma tabela sempre terá esse formato:

CAMPO 1	CAMPO 2	CAMPO 3	CAMPO 4

# SQL 2014 - Módulo I

E nunca poderá ter esse formato:

CAMPO 1	CAMPO 2	CAMPO 3	CAMPO 4

Considere os seguintes exemplos:

- Uma pessoa tem apenas um nome, um RG, um CPF, mas pode ter estudado em **N** escolas diferentes e pode ter feito **N** cursos extracurriculares;
- Um treinamento da Impacta tem um único nome, uma única carga horária, mas pode haver **N** instrutores que ministram esse treinamento;
- Um aluno da Impacta tem apenas um nome, um RG, um CPF, mas pode ter **N** telefones.

Percebemos aqui que a tabela ALUNOS, que criamos anteriormente, precisa ser reestruturada para que respeite a primeira forma normal.

Sempre que uma linha de uma tabela tiver **N** informações relacionadas a ela, precisaremos criar outra tabela para armazenar essas **N** informações:

```
DROP TABLE ALUNOS;

CREATE TABLE ALUNOS
(
    NUM_ALUNO           INT IDENTITY PRIMARY KEY,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO     DATETIME,
    IDADE               TINYINT,
    E_MAIL               VARCHAR(50),
    PROFISSAO            VARCHAR(40),
    EMPRESA              VARCHAR(50) );

CREATE TABLE FONES
(
    NUM_ALUNO           INT,
    FONE                CHAR(9),
    TIPO                VARCHAR(15),
    PRIMARY KEY (NUM_ALUNO, FONE) );
```

No caso da tabela FONES, a chave primária é formada por dois campos, **NUM\_ALUNO** e **FONE**. Isso impedirá que se cadastre o mesmo telefone mais de uma vez para o mesmo aluno:

```
INSERT INTO ALUNOS  
(NOME, DATA_NASCIMENTO, IDADE, E_MAIL,  
PROFISSAO, EMPRESA )  
VALUES  
( 'CARLOS MAGNO', '1959.11.12', 53, 'magno@magno.com',  
'ANALISTA DE SISTEMAS', 'IMPACTA TECNOLOGIA'),  
( 'André da Silva', '1980.1.2', 33, 'andre@silva.com',  
'ANALISTA DE SISTEMAS', 'SOMA INFORMÁTICA'),  
( 'Marcelo Soares', '1983.4.21', 30, 'marcelo@soares.com',  
'INSTRUTOR', 'IMPACTA TECNOLOGIA'),  
( 'PEDRO PAULO', '1994.2.5', 19, 'pedro@pedro.com',  
'ESTUDANTE', 'COLÉGIO MONTE VIDEL'),  
( 'MARIA LUIZA', '1997.10.29', 15, 'luiiza@luiiza.com',  
'ESTUDANTE', 'COLÉGIO MONTE VIDEL') ;
```

```
INSERT INTO FONES  
( NUM_ALUNO, FONE, TIPO )  
VALUES  
( 1,'28739988','RESIDENCIAL'),  
( 1,'68336989','COMERCIAL'),  
( 1,'991259976','CELULAR'),  
( 2,'992736266','CELULAR'),  
( 3,'52417262','COMERCIAL'),  
( 3,'998271717','CELULAR'),  
( 5,'77162525','RESIDENCIAL');
```

- **Second Normal Form (2NF)**

No segundo nível de normalização, devemos criar tabelas separadas para conjuntos de valores que se aplicam a vários registros, ou seja, que se repetem.

Com a finalidade de criar relacionamentos, devemos relacionar essas novas tabelas com uma chave estrangeira e identificar cada grupo de dados relacionados com uma chave primária.

Em outras palavras, a segunda forma normal pede que evitemos campos descritivos (alfanuméricos) que se repitam várias vezes na mesma tabela. Além de ocupar mais espaço, a mesma informação pode ser escrita de formas diferentes. Veja o caso da tabela **ALUNOS**, em que existe um campo chamado **PROFISSAO** (descritivo) onde é possível grafarmos a mesma profissão de várias formas diferentes:

ANALISTA DE SISTEMAS  
ANALISTA SISTEMAS  
AN. SISTEMAS  
AN. DE SISTEMAS  
ANALISTA DE SIST.

Isso torna impossível que se gere um relatório filtrando os **ALUNOS** por **PROFISSAO**. A solução, neste caso, é criar uma tabela de profissões em que cada profissão tenha um código. Para isso, na tabela **ALUNOS**, substituiremos o campo **PROFISSAO** por **COD\_PROFISSAO**:

```
DROP TABLE ALUNOS;

CREATE TABLE ALUNOS
(    NUM_ALUNO           INT IDENTITY PRIMARY KEY,
     NOME                VARCHAR(30),
     DATA_NASCIMENTO    DATETIME,
     IDADE               TINYINT,
     E_MAIL              VARCHAR(50),
     COD_PROFISSAO      INT,
     COD_EMPRESA         INT );

CREATE TABLE PROFISSOES
( COD_PROFISSAO  INT IDENTITY PRIMARY KEY,
  PROFISSAO       VARCHAR(30) );

INSERT INTO PROFISSOES
( PROFISSAO )
VALUES ('ANALISTA DE SISTEMAS'), ('INSTRUTOR'), ('ESTUDANTE');

SELECT * FROM PROFISSOES;

CREATE TABLE EMPRESAS
( COD_EMPRESA        INT IDENTITY PRIMARY KEY,
  EMPRESA            VARCHAR(50) );

INSERT INTO EMPRESAS
( EMPRESA )
VALUES ('IMPACTA TECNOLOGIA'), ('SOMA INFORMÁTICA'), ('COLÉGIO MONTE VIDEL');

SELECT * FROM EMPRESAS;

INSERT INTO ALUNOS
(NOME, DATA_NASCIMENTO, IDADE, E_MAIL,
 COD_PROFISSAO, COD_EMPRESA )
VALUES
('CARLOS MAGNO', '1959.11.12', 53, 'magno@magno.com',
 1, 1),
('André da Silva', '1980.1.2', 33, 'andre@silva.com',
 1, 2),
('Marcelo Soares', '1983.4.21', 30, 'marcelo@soares.com',
 2, 1),
('PEDRO PAULO', '1994.2.5', 19, 'pedro@pedro.com',
 3, 3),
('MARIA LUIZA', '1997.10.29', 15, 'luiza@luiza.com',
 3, 3);
```

- **Third Normal Form (3NF)**

No terceiro nível de normalização, após ter concluído todas as tarefas do **1NF** e **2NF**, devemos eliminar os campos que não dependem de chaves primárias.

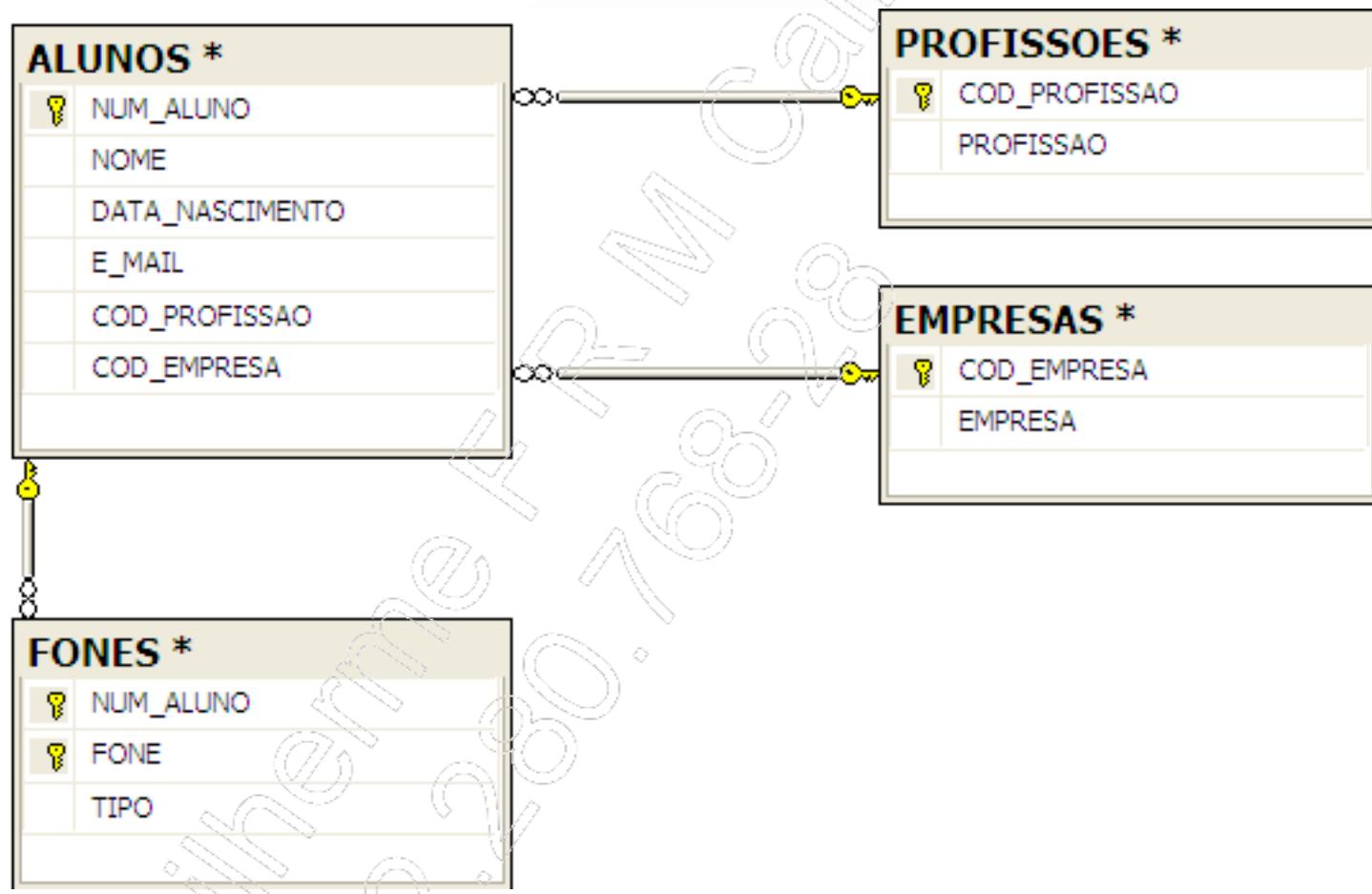
Cumpridas essas três regras, atingimos o nível de normalização requerido pela maioria dos programas.

Considere os seguintes exemplos:

- Em uma tabela de PRODUTOS, que tenha os campos **PRECO\_COMPRA** e **PRECO\_VENDA**, não devemos ter um campo **LUCRO**, pois ele não depende do código do produto (chave primária), mas sim dos preços de compra e de venda. O lucro será facilmente gerado através da expressão **PRECO\_VENDA - PRECO\_CUSTO**;
- Na tabela ALUNOS, não devemos ter o campo **IDADE**, pois ela não depende do número do aluno (chave primária), mas sim do campo **DATA\_NASCIMENTO**.

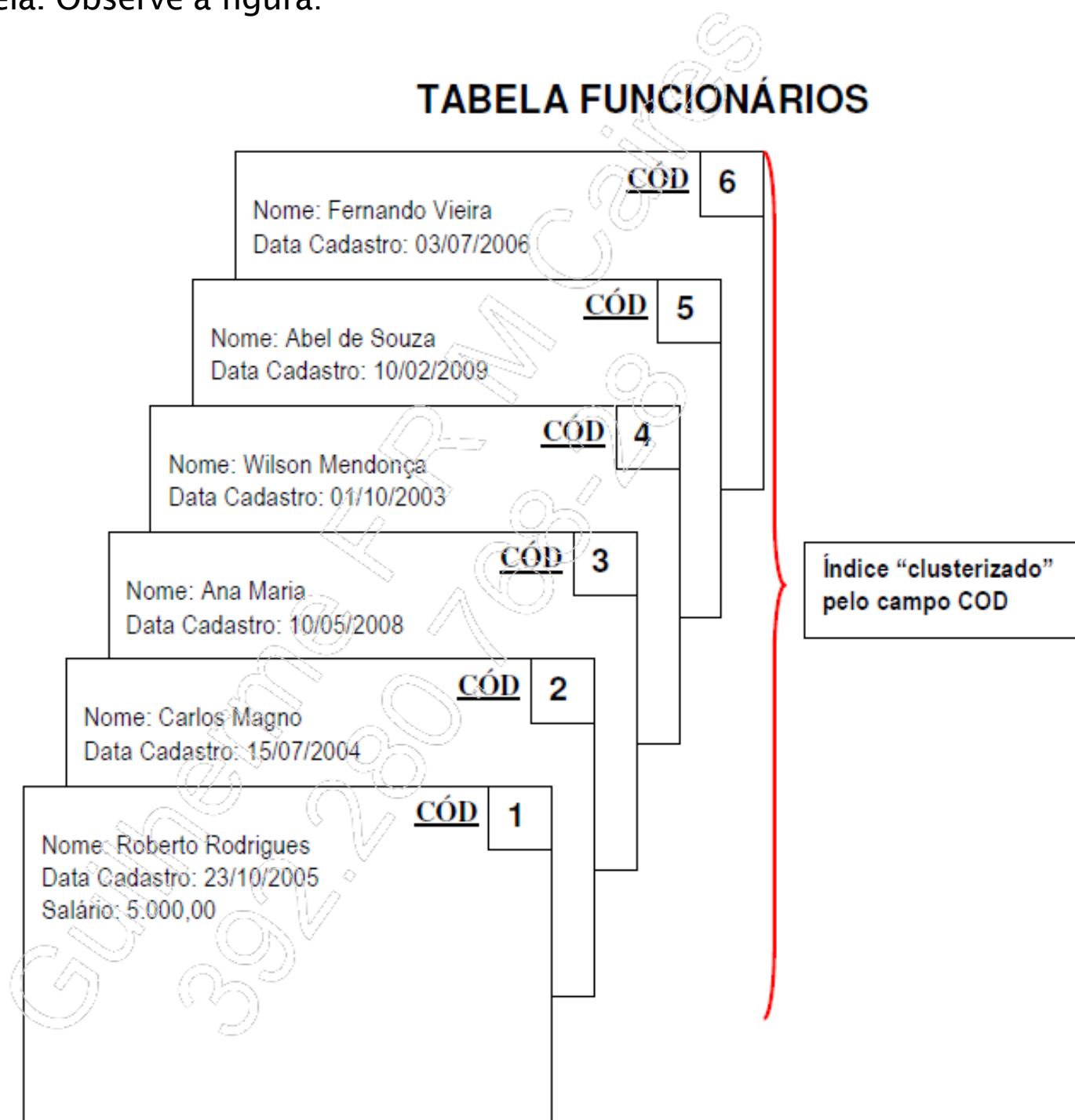
```
-- Eliminar o campo IDADE da tabela ALUNOS
ALTER TABLE ALUNOS DROP COLUMN IDADE;
-- Calcular a IDADE do ALUNO
SELECT *,
       CAST((GETDATE() - DATA_NASCIMENTO) AS FLOAT) / 365.25 AS IDADE
FROM ALUNOS;
```

Por fim, o banco de dados **SALA\_DE\_AULA** ficou com a seguinte estrutura:



## 2.8. Índices

Os índices tornam mais rápidos os procedimentos de ordenação e de busca na tabela. Observe a figura:



Duas estruturas diferentes de índices podem ser utilizadas:

- **Clustered (clusterizado):** Neste tipo de estrutura, a tabela é ordenada fisicamente. Por meio do índice clusterizado, é possível otimizar a performance de leituras baseadas na filtragem de dados de acordo com uma faixa de valores. É permitido o uso de apenas um índice clusterizado por tabela, já que as linhas de dados só podem ser ordenadas de uma maneira;
- **NonClustered (não clusterizado):** Neste tipo de estrutura, os dados de uma tabela são ordenados de maneira lógica. Os índices não clusterizados possuem valores chave, sendo que um ponteiro para a linha de dados é encontrado em cada entrada desses valores. Esse ponteiro é conhecido como localizador de linha.

### ÍNDICES NÃO “CLUSTERIZADOS”

Índice por NOME	
Abel de Souza	5
Ana Maria	3
Carlos Magno	2
Fernando Vieira	6
Roberto Rodrigues	1
Wilson Mendonça	4

Índice por DATA	
01/10/2003	4
15/07/2004	2
23/10/2005	1
03/07/2006	6
10/05/2008	3
10/02/2009	5

## 2.8.1. Criando índices

A criação de índices se dá por meio do comando **CREATE INDEX**. Sua sintaxe é exibida a seguir:

```
CREATE INDEX [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX <nome_do_
índice>
ON <nome_tabela_ou_view> ( <nome_coluna> [ASC | DESC] [, . . . ] )
```

Em que:

- **UNIQUE**: A duplicidade do campo chave do índice não será permitida se utilizarmos essa palavra;
- **CLUSTERED**: Indica que as linhas da tabela estarão fisicamente ordenadas pelo campo que é a chave do índice;
- **NONCLUSTERED**: Indica que o índice não interfere na ordenação das linhas da tabela (default);
- **<nome\_tabela\_ou\_view>**: Nome da tabela ou view para a qual o índice será criado;
- **<nome\_coluna>**: É a coluna da tabela que será a chave do índice;
- **ASC**: Esta palavra determina a ordenação ascendente (padrão);
- **DESC**: Esta palavra determina a ordenação descendente.

## 2.8.1.1. Excluindo índices

Para a exclusão de índices de um banco de dados, utilizamos o comando **DROP INDEX**, cuja sintaxe é a seguinte:

```
DROP INDEX <nome_tabela_ou_view>.<nome_indice>
```

Em que:

- **<nome\_tabela\_ou\_view>**: É o nome da tabela ou view com a qual o índice a ser excluído está associado;
- **<nome\_indice>**: É o nome do índice a ser excluído.

# Pontos principais

**Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.**

- Os objetos que fazem parte de um sistema são criados dentro de um objeto denominado **database**, ou seja, uma estrutura lógica formada por dois tipos de arquivo: um responsável pelo armazenamento de dados e outro que armazena as transações feitas. Para que um banco de dados seja criado no SQL Server, é necessário utilizar a instrução **CREATE DATABASE**;
- Os dados de um sistema são armazenados em objetos denominados tabelas (**table**). Cada uma das colunas de uma tabela refere-se a um atributo associado a uma determinada entidade. A instrução **CREATE TABLE** deve ser utilizada para criar tabelas dentro de bancos de dados já existentes;
- Cada elemento, como uma coluna, uma variável ou uma expressão, possui um tipo de dado. O tipo de dado especifica o tipo de valor que o objeto pode armazenar, como números inteiros, texto, data e hora etc.;
- Normalmente, as tabelas possuem uma coluna contendo valores capazes de identificar uma linha de forma exclusiva. Essa coluna recebe o nome de chave primária, cuja finalidade é assegurar a integridade dos dados da tabela;
- As constraints são objetos utilizados com a finalidade de definir regras referentes à integridade e à consistência nas colunas das tabelas que fazem parte de um sistema de banco de dados;
- Para assegurar a integridade dos dados de uma tabela, o SQL Server oferece cinco tipos diferentes de constraints: PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK e DEFAULT;

- Cada uma das constraints possui regras de utilização. Uma coluna que é definida como chave primária, por exemplo, não pode aceitar valores nulos. Em cada tabela, pode haver somente uma constraint de chave primária;
- Podemos criar constraints com uso de CREATE TABLE, ALTER TABLE ou graficamente (a partir da interface do SQL Server Management Studio);
- Normalização é o processo de organizar dados e eliminar informações redundantes de um banco de dados. Envolve a tarefa de criar as tabelas, bem como definir relacionamentos. O relacionamento entre as tabelas é criado de acordo com regras que visam à proteção dos dados e à eliminação de dados repetidos. Essas regras são denominadas **normal forms**, ou formas normais;
- Informações específicas de uma tabela ou de uma view podem ser localizadas de maneira mais rápida com a utilização de índices. Um índice (index) é uma coleção de páginas associadas com uma tabela ou view, a fim de acelerar o retorno de suas linhas. O índice contém chaves feitas de uma ou mais colunas da tabela ou view e indicadores que mapeiam o local de armazenamento do dado especificado.



# Criando um banco de dados

## Teste seus conhecimentos

2

Guilherme Caires  
392.280.1000



**IMPACTA**  
EDITORA

**1. Qual dos comandos a seguir cria um banco de dados chamado VENDAS?**

- a) CREATE TABLE VENDAS
- b) CREATE NEW DATA VENDAS
- c) CREATE VENDAS DATABASE
- d) CREATE DATABASE VENDAS
- e) NEW DATABASE VENDAS

## 2. Qual das alternativas possui uma afirmação correta a respeito do seguinte código?

```
CREATE TABLE ALUNOS
(
    COD_ALUNO      INT  IDENTITY           PRIMARY KEY,
    NOME            VARCHAR(40),
    DATA_NASCIMENTO DATETIME,
    IDADE           TINYINT
)
```

- a) A coluna NOME armazenará sempre 40 caracteres, independente do nome inserido nela.
- b) A coluna DATA\_NASCIMENTO poderá armazenar datas desde o ano 0001 até 9999.
- c) A coluna NOME armazenará no máximo 40 caracteres, ocupando apenas a quantidade de caracteres contida no nome inserido nela.
- d) A coluna IDADE poderá armazenar números inteiros no intervalo de -255 até +255.
- e) A coluna COD\_ALUNO poderá armazenar números inteiros de -32000 até +32000.

### 3. Qual das alternativas possui uma afirmação correta a respeito do seguinte código?

```
CREATE TABLE ALUNOS
(
    COD_ALUNO      INT  IDENTITY      PRIMARY KEY,
    NOME           VARCHAR(40),
    DATA_NASCIMENTO DATETIME,
    IDADE          TINYINT
)
```

- a) A coluna COD\_ALUNO será numerada automaticamente pelo SQL-SERVER.
- b) A coluna DATA\_NASCIMENTO poderá armazenar datas desde o ano 0001 até 9999.
- c) Ocorrerá erro na definição da coluna COD\_ALUNO porque não existe um tipo chamado INT, o correto é INTEGER.
- d) A coluna IDADE poderá armazenar números inteiros no intervalo de -255 até +255.
- e) A coluna COD\_ALUNO poderá armazenar números inteiros de -32000 até +32000.

## 4. Qual das alternativas possui uma afirmação correta a respeito do seguinte código?

```
CREATE TABLE ALUNOS
(
    COD_ALUNO      INT  IDENTITY      PRIMARY KEY,
    NOME           VARCHAR(40),
    DATA_NASCIMENTO DATETIME,
    IDADE          TINYINT
)
```

- a) A coluna DATA\_NASCIMENTO poderá armazenar datas desde o ano 0001 até 9999.
- b) De acordo com as regras de normalização, a coluna IDADE não deveria fazer parte da estrutura da tabela, pois é decorrente de um cálculo envolvendo o campo DATA\_NASCIMENTO.
- c) Ocorrerá erro na definição da coluna COD\_ALUNO porque não existe um tipo chamado INT, o correto é INTEGER.
- d) A coluna IDADE poderá armazenar números inteiros no intervalo de -255 até +255.
- e) A coluna COD\_ALUNO poderá armazenar números inteiros de -32000 até +32000.

**5. Qual dos comandos adiante insere uma linha com dados na tabela chamada ALUNOS, admitindo que a configuração de formato de data seja ‘yyyy.mm.dd’?**

```
COD_ALUNO      INT  IDENTITY      PRIMARY KEY,  
    NOME          VARCHAR(40),  
    DATA_NASCIMENTO DATETIME,  
    E_MAIL        VARCHAR(100)
```

- a) `INSERT INTO ALUNOS (COD_ALUNO, NOME, DATA_NASCIMENTO, E_MAIL)  
VALUES (1, 'MAGNO', '1959.11.12', 'magno@magno.com.br')`
- b) `INSERT INTO ALUNOS (COD_ALUNO, NOME, DATA_NASCIMENTO, E_MAIL)  
VALUES (1, 'MAGNO', 1959.11.12, 'magno@magno.com.br')`
- c) `INSERT INTO ALUNOS (NOME, DATA_NASCIMENTO, E_MAIL)  
VALUES ('MAGNO', '1959.11.12', 'magno@magno.com.br')`
- d) `INSERT INTO ALUNOS (NOME, DATA_NASCIMENTO, E_MAIL)  
VALUES ('MAGNO', 1959.11.12, 'magno@magno.com.br')`
- e) `INSERT INTO ALUNOS (NOME, DATA_NASCIMENTO, E_MAIL)  
VALUES (MAGNO, '1959.11.12', 'magno@magno.com.br')`

**6. Qual tipo de campo é mais adequado para armazenar a quantidade de pessoas que moram em uma casa?**

- a) CHAR(2)
- b) INT
- c) SMALLINT
- d) TINYINT
- e) NUMERIC(4,2)

**7. Qual tipo de campo é mais adequado para armazenar o preço de um produto?**

- a) CHAR(8)
- b) INT
- c) NUMERIC(8)
- d) NUMERIC(2,10)
- e) NUMERIC(10,2)

**8. Qual tipo de campo é mais adequado para armazenar o CEP (código de endereçamento postal)?**

- a) CHAR(8)
- b) INT
- c) VARCHAR(8)
- d) NUMERIC(8)
- e) NUMERIC(9)

# Criando um banco de dados

## Mãos à obra!

2

Guilherme Caires  
392.280.1000



**IMPACTA**  
EDITORA

## Laboratório 1

### A - Criando constraints com ALTER TABLE

1. Abra o script chamado **Cap02\_CRIA\_PEDIDOS\_VAZIO.sql** e execute todo o código. Isso irá criar um banco de dados chamado **PEDIDOS\_VAZIO**, cuja estrutura é a mesma do banco de dados **PEDIDOS** já utilizado;
2. Coloque em uso o banco de dados **PEDIDOS\_VAZIO**;
3. Crie chaves estrangeiras para a tabela **PEDIDOS**:
  - Com **CLIENTES**;
  - Com **VENDEDORES**.
4. Crie chaves estrangeiras para a tabela **PRODUTOS**:
  - Com **TIPOPRODUTO**;
  - Com **UNIDADES**.
5. Crie chaves estrangeiras para a tabela **ITENSPEDIDO**:
  - Com **PEDIDOS**;
  - Com **PRODUTOS**;
  - Com **TABCOR**.
6. Crie uma chave única para o campo **UNIDADE** da tabela **UNIDADES**;
7. Crie uma chave única para o campo **TIPO** da tabela **TIPOPRODUTO**;
8. Crie constraints **CHECK** para a tabela **PRODUTOS**, considerando os seguintes aspectos:
  - O preço de venda não pode ser menor que o preço de custo;
  - O preço de custo precisa ser maior que zero;
  - O campo **QTD\_REAL** não pode ser menor que zero.

9. Crie constraints **CHECK** para a tabela **ITENS\_PEDIDO**, considerando os seguintes aspectos:

- O campo **QUANTIDADE** deve ser maior ou igual a um;
- O campo **PR\_UNITARIO** deve ser maior que zero;
- O campo **DESCONTO** não pode ser menor que zero e maior que 10.

10. Crie valores default para **PRODUTOS**, considerando os seguintes aspectos:

- Zero para **PRECO\_CUSTO** e **PRECO\_VENDA**;
- Zero para **QTD\_REAL**, **QTD\_MINIMA** e **QTD\_ESTIMADA**;
- Zero para **COD\_TIPO** e **COD\_UNIDADE**.

11. Crie índices para os seguintes campos da tabela **CLIENTES**:

- Campo **NOME**;
- Campo **FANTASIA**;
- Campo **ESTADO**;
- Campo **CEP**;
- Campo **CGC**.

12. Crie índices para os seguintes campos:

- Campos **NOME** e **FANTASIA** da tabela **VENDEDORES**;
- Campo **DESCRICAO** da tabela **PRODUTOS**;
- Campo **DATA\_EMISSAO** da tabela **PEDIDOS**.



# Inserção de dados

# 3

- ✓ Constantes;
- ✓ Inserindo dados;
- ✓ Utilizando TOP em uma instrução INSERT;
- ✓ OUTPUT.

## 3.1. Constantes

As constantes, ou literais, são informações fixas que, como o nome sugere, não se alteram no decorrer do tempo. Por exemplo, o seu nome escrito no papel é uma constante e a sua data de nascimento é outra constante. Existem regras para escrever constantes no SQL:

- **Constantes de cadeia de caracteres (CHAR e VARCHAR)**

São sequências compostas por quaisquer caracteres existentes no teclado. Este tipo de constante deve ser escrita entre apóstrofos:

```
'IMPACTA TECNOLOGIA', 'SQL-SERVER', 'XK-1808/2',  
'CAIXA D''AGUA'
```

Se o conteúdo do texto possuir o caractere apóstrofo, ele deve ser colocado duas vezes, como mostrado em CAIXA D'AGUA.

- **Cadeias de caracteres Unicode**

Semelhante ao caso anterior, estas constantes devem ser precedidas pela letra maiúscula N (identificador):

```
N' IMPACTA TECNOLOGIA', N' SQL-SERVER', N' XK-1808/2'
```

- **Constantes binárias**

São cadeias de números hexadecimais e apresentam as seguintes características:

- Não são incluídas entre aspas;
- Possuem o prefixo 0x.

Veja um exemplo:

```
0xff, 0x0f, 0x01a0
```

- **Constantes datetime**

Utilizam valores de data incluídos em formatos específicos. Devem ser incluídos entre aspas simples:

```
'2009.1.15', '20080115', '01/15/2008', '22:30:10', '2009.1.15  
22:30:10'
```

O formato da data pode variar dependendo de configurações do SQL. Podemos também utilizar o comando **SET DATEFORMAT** para definir o formato durante uma seção de trabalho.

- **Constantes bit**

Não incluídas entre aspas, as constantes bit são representadas por **0** ou **1**. Uma constante desse tipo será convertida em **1**, caso um número maior do que **1** seja utilizado.

```
0, 1
```

- **Constantes float e real**

São constantes representadas por notação científica:

```
2.53E4 2.53 x 104      2.53 x 10000 25300  
4.5E-2 4.5 / 102      4.5 / 100      0.045
```

- **Constantes integer**

São representadas por uma cadeia de números sem pontos decimais e não incluídos entre aspas. As constantes **integer** não aceitam números decimais, somente números inteiros:

```
1528  
817215  
5
```

**Nunca utilize o separador de milhar.**

- **Constantes decimal**

São representadas por cadeias numéricas com ponto decimal e não incluídas entre aspas:

```
162.45  
5.78
```

O separador decimal sempre será o ponto, independentemente das configurações regionais do Windows.

- **Constantes uniqueidentifier**

É uma cadeia de caracteres que representa um GUID. Pode ser especificada como uma cadeia de binários ou em um formato de caracteres:

```
0xff19966f868b11d0b42d00c04fc964ff  
'6F9619FF-8B86-D011-B42D-00C04FC964FF'
```

- **Constantes money**

São precedidas pelo caractere cifrão (\$). Este tipo de dado sempre reserva 4 posições para a parte decimal. Os algarismos além da quarta casa decimal serão desprezados.

```
$1543.56  
$12892.6534  
$56.275639
```

No último exemplo, será armazenado apenas 56.2756.

## 3.2. Inserindo dados

Para acrescentar novas linhas de dados em uma tabela, utilize o comando **INSERT**, que possui a seguinte sintaxe:

```
INSERT [INTO] <nome_tabela>
[ ( <lista_de_colunas> ) ]
{ VALUES ( <lista_de_expressoess1> )
    [, (<lista_de_expressoess2>)] [, ...] |
<comando_select> }
```

Em que:

- **<lista\_de\_colunas>**: É uma lista de uma ou mais colunas que receberão dados. Os nomes das colunas devem ser separados por vírgula e a lista deve estar entre parênteses;
- **VALUES (<lista\_de\_expressoess>)**: Lista de valores que serão inseridos em cada uma das colunas especificadas em **<lista\_de\_colunas>**.

Para inserir uma única linha em uma tabela, o código é o seguinte:

```
INSERT INTO ALUNOS
(NOME, DATA_NASCIMENTO, IDADE, E_MAIL,
FONE_RES, FONE_COM, FAX, CELULAR,
PROFISSAO, EMPRESA )
VALUES
('CARLOS MAGNO', '1959.11.12', 53, 'magno@magno.com',
'23456789','23459876','','', '998765432',
'ANALISTA DE SISTEMAS', 'IMPACTA TECNOLOGIA');

-- Consultar os dados inseridos na tabela
SELECT * FROM ALUNOS;
```

# SQL 2014 - Módulo I

Podemos inserir várias linhas em uma tabela com o uso de vários comandos **INSERT** ou um único:

```
INSERT INTO ALUNOS
(NOME, DATA_NASCIMENTO, IDADE, E_MAIL,
FONE_RES, FONE_COM, FAX, CELULAR, PROFISSAO, EMPRESA)
VALUES
('André da Silva', '1980.1.2', 33, 'andre@silva.com',
'23456789','23459876','','998765432',
'ANALISTA DE SISTEMAS', 'SOMA INFORMÁTICA'),
('Marcelo Soares', '1983.4.21', 30, 'marcelo@soares.com',
'23456789','23459876','','998765432',
'INSTRUTOR', 'IMPACTA TECNOLOGIA');

-- Consultar os dados da tabela
SELECT * FROM ALUNOS;
```

Podemos também fazer **INSERT** de **SELECT**:

```
CREATE TABLE ALUNOS2
(
    NUM_ALUNO           INT,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO     DATETIME,
    IDADE               TINYINT,
    E_MAIL               VARCHAR(50),
    FONE_RES             CHAR(8),
    FONE_COM             CHAR(8),
    FAX                 CHAR(8),
    CELULAR              CHAR(9),
    PROFISSAO            VARCHAR(40),
    EMPRESA              VARCHAR(50) );

INSERT INTO ALUNOS2
SELECT * FROM ALUNOS;
```

Não é necessário determinar os nomes das colunas na sintaxe do comando **INSERT** quando os valores forem inseridos na mesma ordem física das colunas no banco de dados. Já para valores inseridos aleatoriamente, é preciso especificar exatamente a ordem das colunas. Esses dois modos de utilização do comando **INSERT** são denominados **INSERT** posicional e **INSERT** declarativo.

### 3.2.1.INSERT posicional

O comando **INSERT** é classificado como posicional quando não especifica a lista de colunas que receberão os dados de **VALUES**. Nesse caso, a lista de valores precisa conter todos os campos, exceto o **IDENTITY**, na ordem física em que foram criadas no comando **CREATE TABLE**. Veja o exemplo a seguir:

```
INSERT INTO ALUNOS
VALUES
('MARIA LUIZA', '1997.10.29', 15, 'luiiza@luiiza.com',
'23456789','23459876','','', '998765432',
'ESTUDANTE', 'COLÉGIO MONTE VIDEL');

-- Consultando os dados
SELECT * FROM ALUNOS;
```

### 3.2.2.INSERT declarativo

O **INSERT** é classificado como declarativo quando especifica as colunas que receberão os dados da lista de valores. Veja o próximo exemplo:

```
INSERT INTO ALUNOS
(NOME, DATA_NASCIMENTO, IDADE, E_MAIL,
FONE_RES, FONE_COM, FAX, CELULAR,
PROFISSAO, EMPRESA )
VALUES
('PEDRO PAULO', '1994.2.5', 19, 'pedro@pedro.com',
'23456789','23459876','','', '998765432',
'ESTUDANTE', 'COLÉGIO MONTE VIDEL');

-- Consultando os dados
SELECT * FROM ALUNOS;
```

Quando utilizamos a instrução **INSERT** dentro de aplicativos, stored procedures ou triggers, deve ser usado o **INSERT** declarativo, pois, se houver alteração na estrutura da tabela (inclusão de novos campos), ele continuará funcionando, enquanto que o **INSERT** posicional provocará erro.

## 3.3. Utilizando TOP em uma instrução INSERT

A cláusula **TOP** em uma instrução **INSERT** define a quantidade ou a porcentagem de linhas que serão inseridas em uma tabela. Isso é muito utilizado para preencher rapidamente tabelas novas com informações existentes.

O exemplo adiante demonstra o uso de **TOP** em uma instrução **INSERT**. Criamos a tabela **CLIENTES\_MG**, copiamos 20 registros da tabela **CLIENTES** para a tabela **CLIENTES\_MG** e, por fim, exibimos esta última:

```
-- Colocar o banco de dados PEDIDOS em uso
USE PEDIDOS;

-- Criar a tabela
CREATE TABLE CLIENTES_MG
( CODIGO      INT      PRIMARY KEY,
  NOME VARCHAR(50),
  ENDERECO VARCHAR(60),
  BAIRRO   VARCHAR(30),
  CIDADE    VARCHAR(30),
  FONE VARCHAR(18) )

-- Copiar 20 registros da tabela CLIENTES para a tabela CLIENTES_MG
INSERT TOP( 20 ) INTO CLIENTES_MG
SELECT CODCLI, NOME, ENDERECO, BAIRRO, CIDADE, FONE1
FROM CLIENTES
WHERE ESTADO = 'MG'

-- Consultar CLIENTES_MG
SELECT * FROM CLIENTES_MG
```

O resultado do código anterior é o seguinte:

	CODIGO	NOME	ENDERECO	BAIRRO	CIDADE	FONE
1	20	BRINDES ART LTDA.	R. BARAO RIO BRANCO, 1.285	CENTRO	PASSOS	035 5214004
2	22	AGUIMAR LUIZ DA SILVA	R.CARDOBA, 26 AP.101	STA.CRUZ	CONTAGEM	NULL
3	24	ASA BRINDES LTDA	R.GENOVEVA DE SOUZA, 1.787	SAGRADA FAMILIA	B.HORIZONTE	031 4613503
4	40	BRAGA BRINDES LTDA.	R.SINVAL CORREIA, 12	NULL	JUIZ DE FORA	032 2115304
5	54	BRINDES MG INDUSTR...	R. RIO BRANCO, 233	AMAZONAS	CONTAGEM	031 3331962
6	76	CLEMENTE GONCALVE...	AV.ALEGARIO MACIEL, 742 L...	CENTRO	BELO HORIZ...	031 2125116
7	77	CONTATO BRINDES P...	R.ALANDINA, 481	CAICARA	BELO HORIZ...	031 4156200
8	82	CONP.MINEIRA DE IMP...	R.SANTOS, 1.931	JD.AMERICA	B.HORIZONTE	031 3732252
9	85	CONDOR PROPAGAND...	R. ESPINOSA, 53	CARLOS PRATES	B.HORIZONTE	031 4117505
10	107	ELMIRO ESPERENDEU...	R.GREGORIO BARBOSA,96	CENTRO	FREI GASPAR	NULL
11	109	ENFOR LTDA	R.SANTOS,1931	JARDIN.AMERICA	BELO HORIZ...	0313732252
12	112	EUDELCIO ALVES FRA...	AV.TEREZINA,2056	UMUARAMA	UBERLANDIA	0342323152
13	126	GRAFICA ELDORADO L...	R.JOAQUIM PEREGRINO,33	NOSSA SENHOR...	PARA DE MI...	0372314577
14	131	HANNAS PERSONALIZ...	R. JUCA FLAVIA,101	INCONFIDENTES	CONTAGEM	031 3623087
15	165	JOSE ADRIANO MARTI...	R.GERALDO GONCALVES FE...	NULL	TEOFILO OT...	0335216983
16	167	JOSE DA LUZ PERIERA...	PRACA DR. MARCOS FROTA...	NULL	VARGINHA	0352215115
17	170	LOURIVAL MATOS ASS...	R.AVES E SILVA,49 SALA 04	NULL	VARGINHA	NULL
18	180	MR SILK SCREEN LTDA	R.CEL JOSE CUSTODIO,48-B ...	CENTRO	CAMPEESTRE	035743 1554
19	202	EIDER PERPETUO	R.RIO PARAOPEBA,1364	RIACHO DAS PE...	CONTAGEM	031 3515683
20	227	LORIVAL MATOS ASSU...	R.RIO DE JANEIRO,419	NULL	VARGINHA	035 2222157

Consulta executada com êxito. SOMAS\SQLEXPRESS2008 (10.0 ... | SOMA5\CARLOS MAGNO SOU... | PEDIDOS | 00:00:00 | 20 linhas

## 3.4. OUTPUT

Para verificar se o procedimento executado pelo comando **INSERT**, **DELETE** ou **UPDATE** foi executado corretamente, podemos utilizar a cláusula **OUTPUT** existente nesses comandos. Essa cláusula mostra os dados que o comando afetou.

Usaremos os prefixos **deleted** ou **inserted** para acessar os dados de antes ou depois da operação:

COMANDO	deleted (antes)	inserted (depois)
DELETE	SIM	NÃO
INSERT	NÃO	SIM
UPDATE	SIM	SIM

A cláusula **OUTPUT** é responsável por retornar resultados com base em linhas que tenham sido afetadas por uma instrução **INSERT**, **UPDATE**, **DELETE** ou **MERGE**. Os resultados retornados podem ser usados por um aplicativo como mensagens, bem como podem ser inseridos em uma tabela ou variável de tabela.

A cláusula **OUTPUT** garante que qualquer uma dessas instruções, mesmo que possua erros, retorne linhas ao cliente. Contudo, é importante ressaltar que o resultado não deve ser usado caso ocorra um erro ao executar a instrução.

## 3.4.1. OUTPUT em uma instrução INSERT

Em uma instrução **INSERT**, a cláusula **OUTPUT** retorna informações das linhas afetadas pela instrução, ou seja, linhas inseridas. Isso pode ser útil para retornar o valor de identidade ou as colunas computadas na instrução. Os resultados retornados também podem ser usados como mensagens de um aplicativo.

A cláusula **OUTPUT** não pode ser utilizada em uma instrução **INSERT** caso o alvo da instrução seja uma tabela remota, expressão de tabela comum ou visualização. Também não pode possuir ou ser referenciada por uma constraint **FOREIGN KEY**.

A seguir, temos um exemplo que demonstra o uso de **OUTPUT** em uma instrução **INSERT**. Primeiramente, vamos criar uma cópia da tabela **EMPREGADOS** chamada **EMP\_TEMP**:

```
IF OBJECT_ID('EMP_TEMP','U') IS NOT NULL  
    DROP TABLE EMP_TEMP;  
CREATE TABLE EMP_TEMP  
( CODFUN      INT PRIMARY KEY,  
  NOME        VARCHAR(30),  
  COD_DEPTO   INT,  
  COD_CARGO   INT,  
  SALARIO     NUMERIC(10,2) );
```

O próximo passo é inserir dados na tabela criada e exibir os registros inseridos:

```
INSERT INTO EMP_TEMP OUTPUT INSERTED.*  
SELECT CODFUN, NOME, COD_DEPTO, COD_CARGO, SALARIO  
FROM EMPREGADOS;  
GO
```

Agora, excluiremos todos os registros da tabela **EMP\_TEMP** e, em seguida, acrescentaremos novos dados e exibiremos algumas colunas:

```
DELETE FROM EMP_TEMP;  
INSERT INTO EMP_TEMP  
OUTPUT INSERTED.CODFUN, INSERTED.NOME, INSERTED.COD_DEPTO  
SELECT CODFUN, NOME, COD_DEPTO, COD_CARGO, SALARIO  
FROM EMPREGADOS WHERE COD_DEPTO = 2;  
GO
```

# SQL 2014 - Módulo I

Depois, declararemos uma variável tabular, acrescentaremos dados e os armazenaremos na variável criada:

```
-- Declarar variável tabular
DECLARE @REG_INSERT TABLE (      CODFUN      INT,
                                NOME        VARCHAR(30),
                                COD_DEPTO   INT,
                                COD_CARGO    INT,
                                SALARIO     NUMERIC(10,2) );

-- Inserir dados e armazenar em variável tabular
INSERT INTO EMP_TEMP
OUTPUT INSERTED.* INTO @REG_INSERT
SELECT CODFUN, NOME, COD_DEPTO, COD_CARGO, SALARIO
FROM EMPREGADOS WHERE COD_DEPTO = 3;
```

Para exibir os registros inseridos, utilizamos a seguinte instrução:

```
SELECT * FROM @REG_INSERT;
```

Já para exibir todos os registros, a instrução utilizada é a seguinte:

```
SELECT * FROM EMP_TEMP;
GO
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Para acrescentar novas linhas de dados em uma tabela, utilizamos o comando **INSERT**;
- No **INSERT** posicional não é necessário mencionar o nome dos campos, porém será obrigatório incluir todos os campos, exceto o campo autonumerável. Já no **INSERT** declarativo, os campos deverão ser informados no comando;
- Podemos restringir a quantidade de inserções do comando por meio da cláusula **TOP**;
- Para auditar e mostrar quais dados foram inseridos, utilizamos a cláusula **OUTPUT**.



3

# Inserção de dados

Teste seus conhecimentos

Guilherme  
392.280.168  
ERP de  
Caires



**IMPACTA**  
EDITORA

## 1. Quantos registros o comando a seguir inserirá na tabela ALUNOS?

```
INSERT ALUNOS
(      NOME,      DATA_NASCIMENTO,      IDADE,      E_MAIL,
FONE_RES,      FONE_COM,                  FAX,          CELULAR,
PROFISSAO,    EMPRESA)
VALUES
('André da Silva', '1980.1.2', 33, 'andre@silva.com',
 '23456789','23459876','','998765432',
 'ANALISTA DE SISTEMAS', 'SOMA INFORMÁTICA'),
('Marcelo Soares', '1983.4.21', 30, 'marcelo@soares.com',
 '23456789','23459876','','998765432',
 'INSTRUTOR', 'IMPACTA TECNOLOGIA'),
('MARIA LUIZA', '1997.10.29', 15, 'luiiza@luiiza.com',
 '23456789','23459876','','998765432',
 'ESTUDANTE', 'COLÉGIO MONTE VIDEL');
```

- a) 1
- b) 2
- c) 3
- d) Não é possível inserir registros utilizando esse comando.
- e) Existe um erro de sintaxe, pois é necessário informar a palavra INTO após o comando INSERT.

**2. Qual das alternativas possui uma afirmação correta a respeito do seguinte código?**

```
INSERT INTO EMP_TEMP OUTPUT DELETED.*  
SELECT CODFUN, NOME, COD_DEPTO, COD_CARGO, SALARIO  
FROM EMPREGADOS;
```

- a) Após a inserção, o SQL apresentará os registros inseridos na tabela EMP\_TEMP.
- b) Após a execução do comando, não será apresentada nenhuma informação.
- c) A sintaxe está errada, pois a cláusula DELETED não é compatível com o INSERT.
- d) A cláusula OUTPUT está localizada no meio do comando e o correto é no final.
- e) As alternativas B e D estão corretas.

### 3. Qual das alternativas possui uma afirmação correta a respeito do seguinte código?

```
INSERT TOP( 20 ) INTO CLIENTES_MG  
SELECT CODCLI, NOME, ENDERECO, BAIRRO, CIDADE, FONE1  
FROM CLIENTES
```

- a) A sintaxe está errada.
- b) A cláusula TOP somente pode ser utilizada no comando SELECT.
- c) O comando INSERT realizará a inserção de todos os registros da tabela CLIENTES.
- d) Serão inseridos 20 registros na tabela CLIENTE\_MG, a partir da consulta da tabela CLIENTES.
- e) Serão inseridos 20 registros na tabela CLIENTE\_MG, a partir da consulta da tabela CLIENTES, que possuam estado não nulo.

**4. Como pode ser classificado o comando INSERT?**

- a) Posicional e Declarativo.
- b) Declarativo.
- c) Posicional.
- d) Interrogativo.
- e) Nenhuma das alternativas anteriores está correta.

**5. Qual afirmação está errada?**

- a) Ao declararmos uma data, é recomendado utilizar o formato 'YYYY-MM-DD'.
- b) Para valores numéricos, é necessário substituir o sinal de vírgula por ponto.
- c) Tipos STRING UNICODE devem ser precedidos da letra N.
- d) Não é possível utilizar um apóstrofo no conteúdo do texto.
- e) Constantes binárias possuem o prefixo 0x.



3

# Inserção de dados

Mãos à obra!

Guilherme Ribeiro Caires  
392.280.100-28



**IMPACTA**  
EDITORA

## Laboratório 1

### A – Criando um banco de dados para administrar as vendas de uma empresa

1. Crie um banco de dados chamado **PEDIDOS\_VENDA** e coloque-o em uso;
2. Nesse banco de dados, crie uma tabela chamada **PRODUTOS** com os seguintes campos:

<b>Código do produto</b>	Inteiro, autonumeração e chave primária
<b>Nome do produto</b>	Alfanumérico
<b>Código da unid. de medida</b>	Inteiro
<b>Código da categoria</b>	Inteiro
<b>Quantidade em estoque</b>	Numérico
<b>Quantidade mínima</b>	Numérico
<b>Preço de custo</b>	Numérico
<b>Preço de venda</b>	Numérico
<b>Características técnicas</b>	Texto longo
<b>Fotografia</b>	Binário longo

3. Crie a tabela **UNIDADES** para armazenar unidades de medida:

<b>Código da unidade</b>	Inteiro, autonumeração e chave primária
<b>Nome da unidade</b>	Alfanumérico

4. Na tabela **UNIDADES**, insira os seguintes dados: **PEÇAS, METROS, QUILOGRAMAS, DÚZIAS, PACOTE, CAIXA**:

5. Crie a tabela **CATEGORIAS** para armazenar as categorias dos produtos:

<b>Código da categoria</b>	Inteiro, autonumeração e chave primária
<b>Nome da categoria</b>	Alfanumérico

6. Na tabela **CATEGORIAS**, insira os seguintes dados: **MOUSE**, **PEN-DRIVE**, **MONITOR DE VIDEO**, **TECLADO**, **CPU**, **CABO DE REDE**;

7. Insira os produtos a seguir utilizando a cláusula **OUTPUT** para mostrar os valores inseridos:

Produto	Unidade	Categoria	Quant.	Qtd. Mínima	Preço Custo	Preço Venda
Caneta Azul	1	1	150	40	0,50	0,75
Caneta Verde	1	1	50	40	0,50	0,75
Caneta Vermelha	1	1	80	35	0,50	0,75
Lápis	1	1	400	80	0,50	0,80
Régua	1	1	40	10	1,00	1,50



# Consultando dados

4

- ✓ SELECT;
- ✓ Ordenação de dados;
- ✓ Filtragem de consultas;
- ✓ Operadores relacionais;
- ✓ Operadores lógicos;
- ✓ Consulta de intervalos com BETWEEN;
- ✓ Consulta com base em caracteres;
- ✓ Consulta de valores em uma lista de elementos;
- ✓ Valores nulos;
- ✓ Substituição de valores nulos;
- ✓ Manipulação de campos do tipo datetime;
- ✓ Alteração do idioma a partir do SSMS.

## 4.1. Introdução

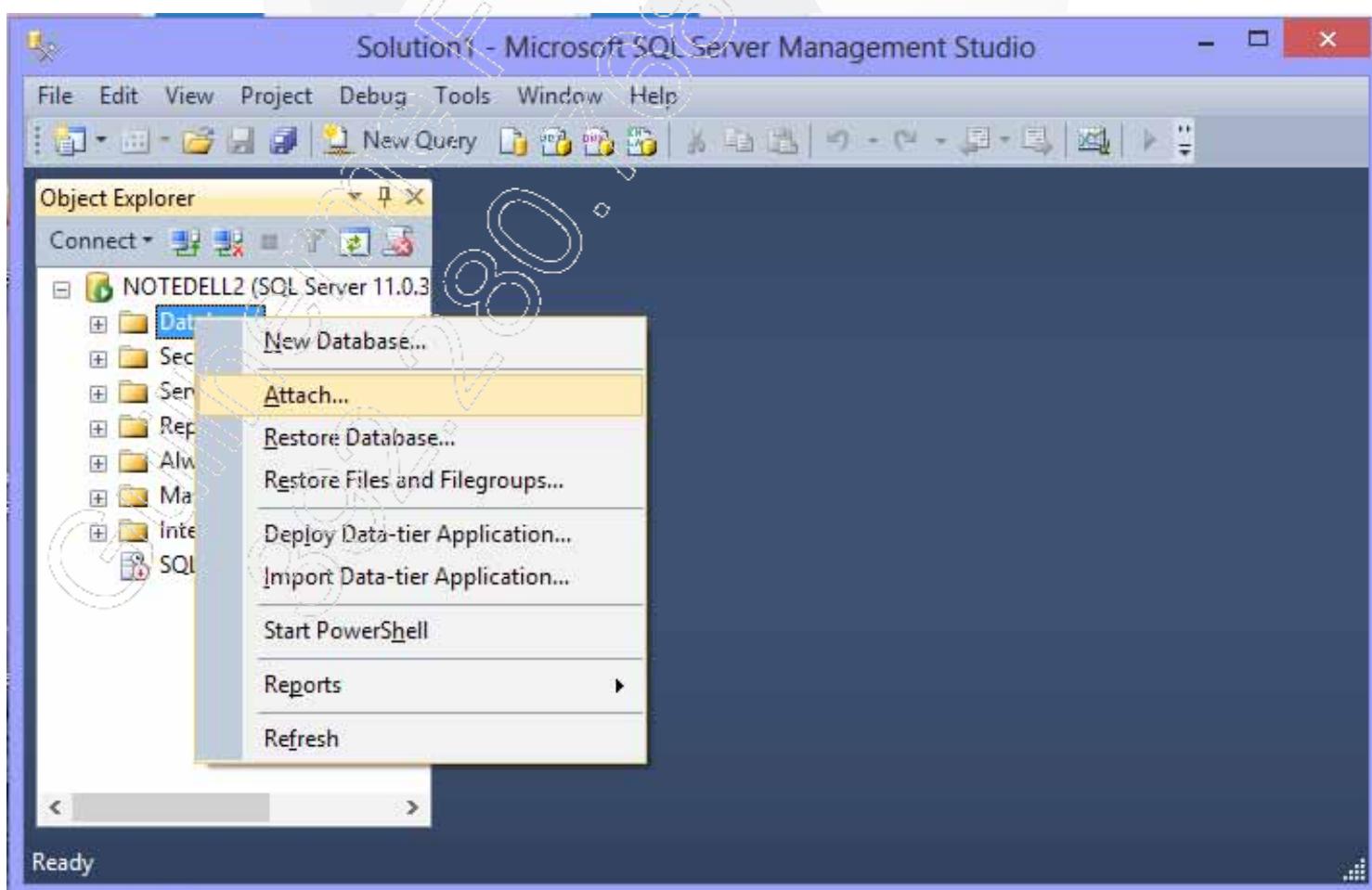
Na linguagem SQL, o principal comando utilizado para a realização de consultas é o **SELECT**. Por meio dele, torna-se possível consultar dados pertencentes a uma ou mais tabelas de um banco de dados.

No decorrer deste capítulo, serão apresentadas as técnicas de utilização do comando **SELECT**, bem como algumas diretrizes para a realização de diferentes tipos de consultas SQL.

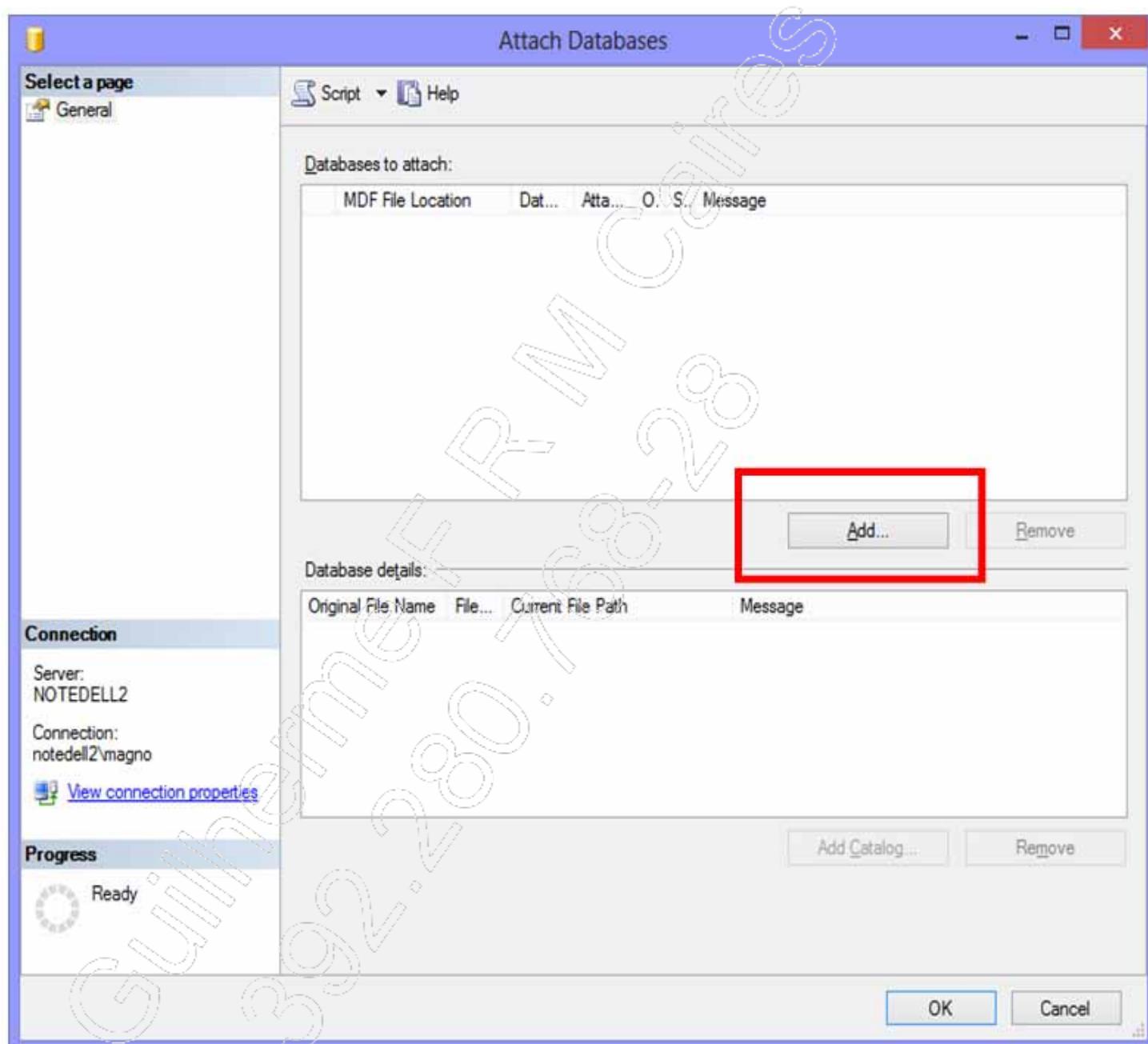
Para que possamos fazer exemplos que demonstrem as técnicas mais apuradas de consulta, precisamos ter um banco de dados com um volume razoável de informações já cadastradas.

Siga os passos adiante:

1. No Object Explorer, clique com o botão direito do mouse sobre o item **Databases** e selecione a opção **Attach**:

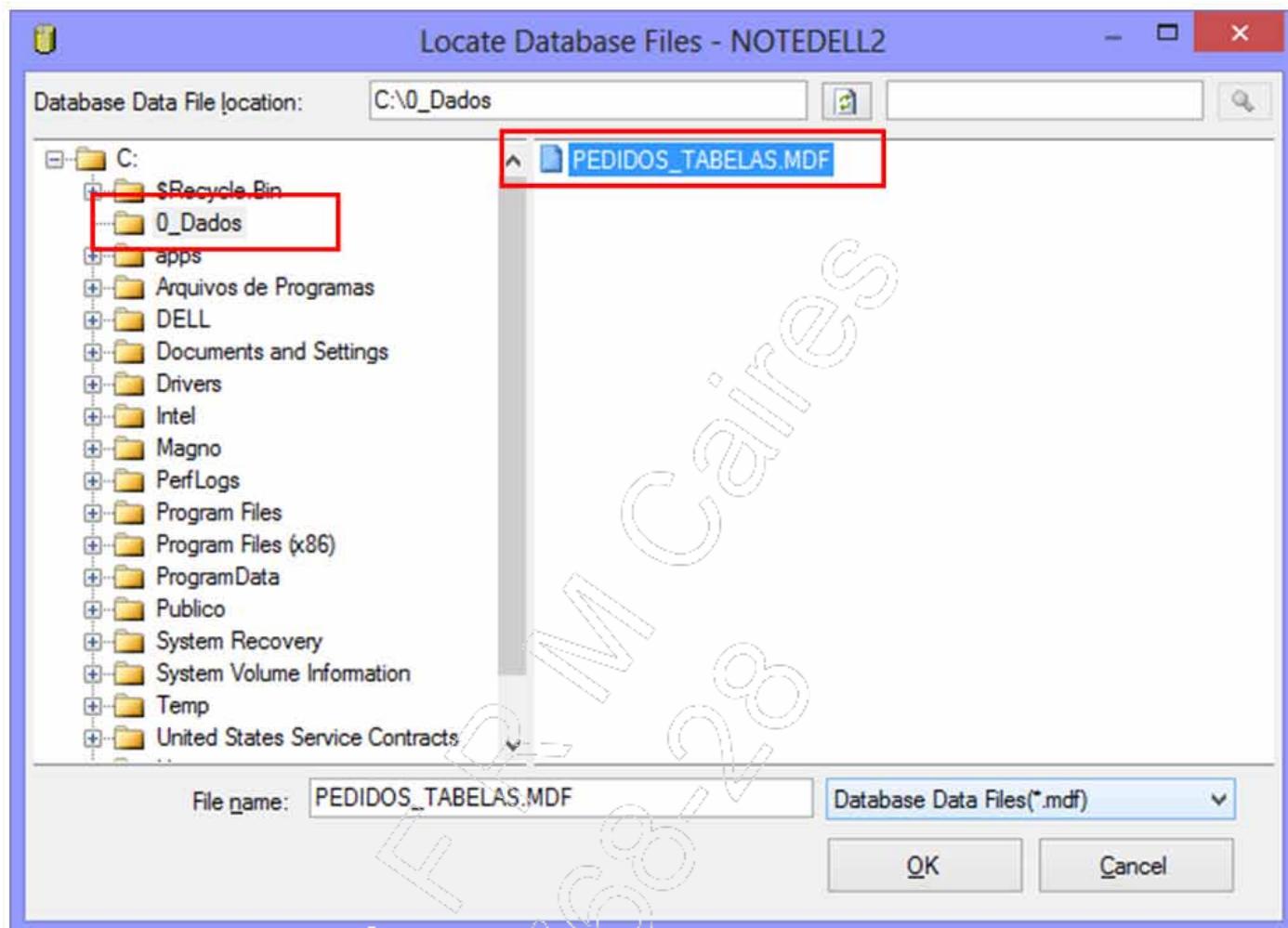


2. Na próxima tela, clique no botão Add:

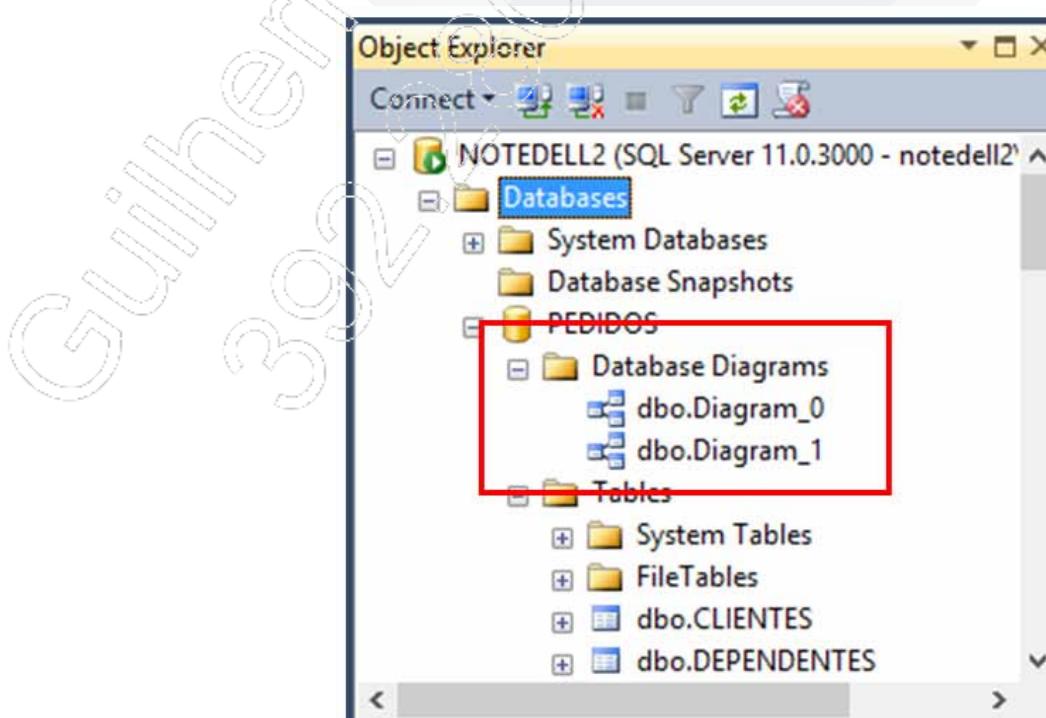


# SQL 2014 - Módulo I

3. Em seguida, procure a pasta **Dados** e selecione o arquivo **PEDIDOS\_TABELAS.MDF**:

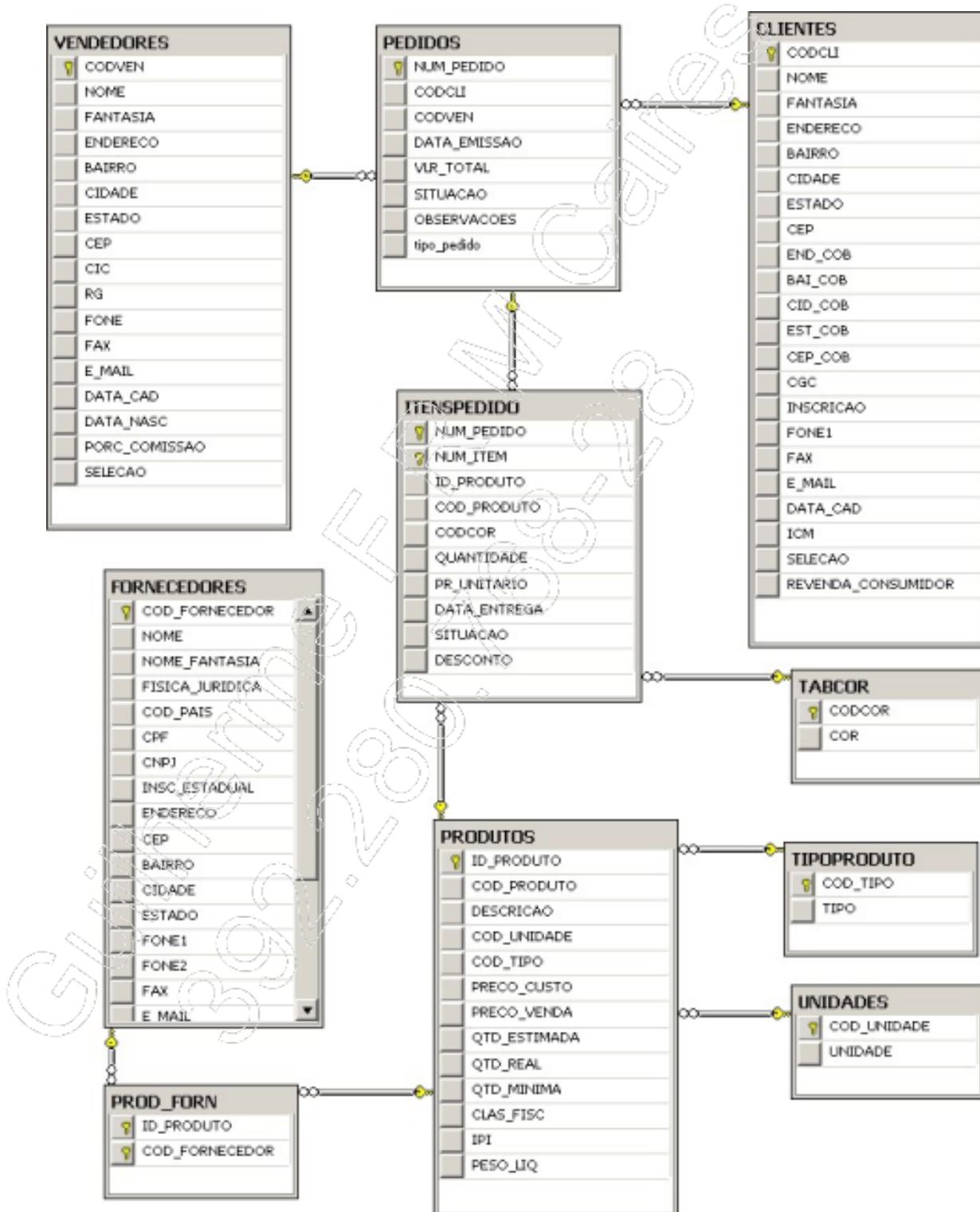


4. Confirme a operação clicando no botão **OK**.

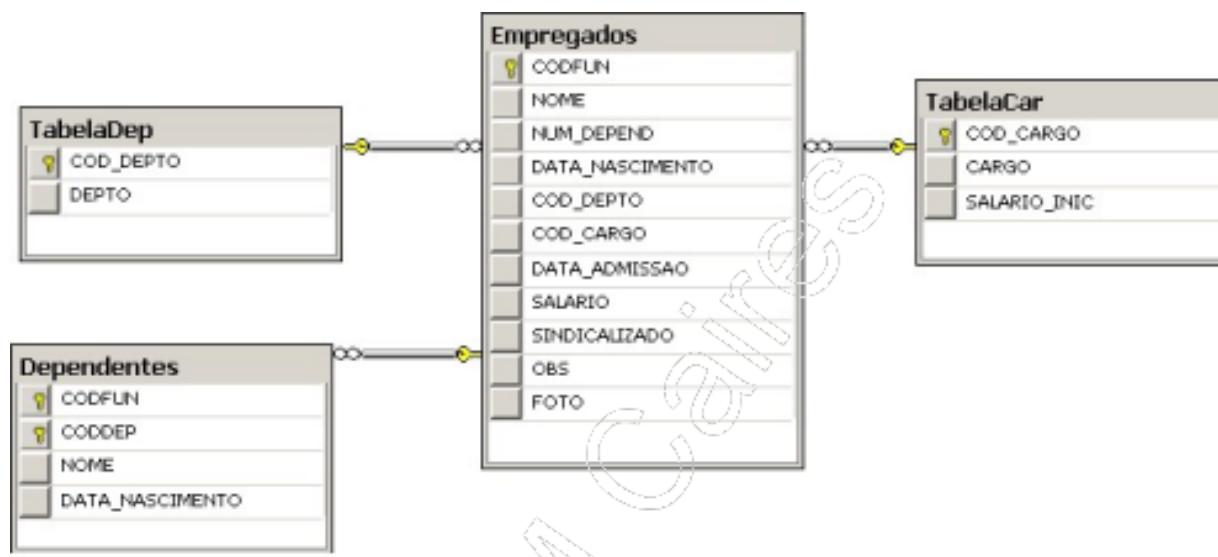


Observe que o banco de dados aparecerá no Object Explorer. Neste banco, foram criados dois diagramas que mostram as tabelas existentes nele. Você conseguirá visualizar o diagrama executando um duplo-clique sobre o nome:

- **DIAGRAMA DE PEDIDOS**



- **DIAGRAMA DE EMPREGADOS**



## 4.2. SELECT

O comando **SELECT** pertence ao grupo de comandos denominado DML (Data Manipulation Language ou Linguagem de Manipulação de Dados), que é composto de comandos para consulta (**SELECT**), inclusão (**INSERT**), alteração (**UPDATE**) e exclusão de dados de tabela (**DELETE**).

A sintaxe de **SELECT**, com seus principais argumentos e cláusulas, é exibida a seguir:

```
SELECT [DISTINCT] [TOP (N) [PERCENT] [WITH TIES]] <lista_de_colunas> [INTO <nome_tabela>]
    FROM tabela1 [JOIN tabela2 ON expressaoJoin [, JOIN tabela3 ON
exprJoin [,...]]]
    [WHERE <condicaoFiltroLinhas>]
    [GROUP BY <listaExprGrupo> [HAVING <condicaoFiltroGrupo>]]
    [ORDER BY <campo1> {[DESC] | [ASC]} [, <campo2> {[DESC] | [ASC]} [,....]]]
```

Em que:

- **[DISTINCT]**: Palavra que especifica que apenas uma única instância de cada linha faça parte do conjunto de resultados. **DISTINCT** é utilizada com o objetivo de evitar a existência de linhas duplicadas no resultado da seleção;
- **[TOP (N) [PERCENT] [WITH TIES]]**: Especifica que apenas um primeiro conjunto de linhas ou uma porcentagem de linhas seja retornado. N pode ser um número ou porcentagem de linhas;
- **<lista\_de\_colunas>**: Colunas que serão selecionadas para o conjunto de resultados. Os nomes das colunas devem ser separados por vírgulas. Caso tais nomes não sejam especificados, todas as colunas serão consideradas na seleção;
- **[INTO nome\_tabela]**: **nome\_tabela** é o nome de uma nova tabela a ser criada com base nas colunas especificadas em **<lista\_de\_colunas>** e nas linhas especificadas por meio da cláusula **WHERE**;
- **FROM tabela1 [JOIN tabela2 ON exprJoin [, JOIN tabela3 ON exprJoin [...]]]**:
  - A cláusula **FROM** define tabelas utilizadas no **SELECT**;
  - **expressaoJoin** é a expressão necessária para relacionar as tabelas da cláusula **FROM**;
  - **tabela1, tabela2,...** são as tabelas que possuem os valores utilizados na condição de filtragem **<condicaoFiltroLinhas>**.
- **[WHERE <condicaoFiltroLinhas>]**: A cláusula **WHERE** aplica uma condição de filtro que determinará quais linhas farão parte do resultado. Essa condição é especificada em **<condicaoFiltroLinhas>**;

- **[GROUP BY <listaExprGrupo>]:**
  - A cláusula **GROUP BY** agrupa uma quantidade de linhas em um conjunto de linhas. Nele, as linhas são resumidas por valores de uma ou várias colunas ou expressões;
  - <listaExprGrupo> representa a expressão na qual será realizada a operação por **GROUP BY**.
- **[HAVING <condicaoFiltroGrupo>]]:** A cláusula **HAVING** define uma condição de busca para o grupo de linhas a ser retornado por **GROUP BY**;
- **[ORDER BY <campo1> {[DESC] | [ASC]} [, <campo2> {[DESC] | [ASC]} [, ...]]]:**
  - A cláusula **ORDER BY** é utilizada para determinar a ordem em que os resultados são retornados;
  - Já **campo1**, **campo2** são as colunas utilizadas na ordenação dos resultados.
- {[DESC]/[ASC]}: **ASC** determina que os valores das colunas especificadas em **campo1**, **campo2** sejam retornados em ordem ascendente, enquanto **DESC** retorna esses valores em ordem descendente. As duas opções são opcionais e a barra indica que são excludentes entre si, ou seja, não podem ser utilizadas simultaneamente. As chaves indicam um grupo excludente de opções. Se nenhuma delas for utilizada, **ASC** será assumido.

Para consultar uma lista de colunas de uma determinada tabela em um banco de dados, basta utilizar a seguinte sintaxe:

```
SELECT <lista_de_colunas> FROM <tabela>
```

Em que:

- <lista\_de\_colunas>: Representa o nome da coluna ou colunas a serem selecionadas. Quando a consulta envolve mais de uma coluna, elas deverão ser separadas por vírgula;
- <tabela>: É o nome da tabela a partir de onde será feita a consulta.

Para especificar o banco de dados de origem das tabelas, a partir do qual as informações serão consultadas, utilize a instrução **USE** seguida pelo nome do banco de dados, da seguinte maneira:

```
USE <nome_banco_de_dados>
```

Essa instrução deve ser especificada na parte inicial da estrutura de código, anteriormente às instruções destinadas à consulta. Os exemplos adiante demonstrarão como utilizá-la junto ao **SELECT**.

### 4.2.1. Consultando todas as colunas

O código a seguir consulta todas as colunas da tabela **EMPREGADOS** do banco de dados **PEDIDOS**:

```
USE PEDIDOS;  
SELECT * FROM EMPREGADOS;
```

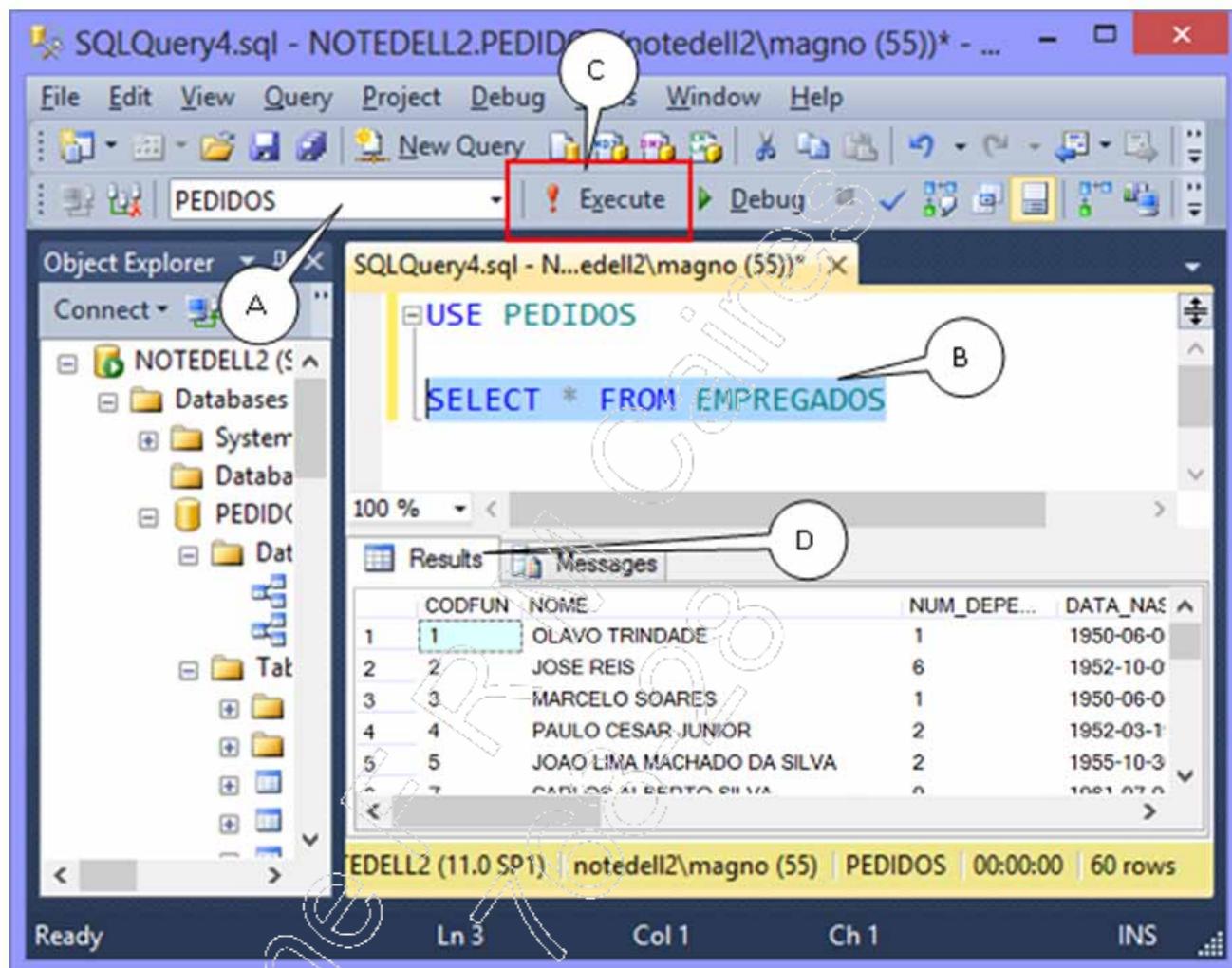
### 4.2.2. Consultando colunas específicas

Para consultar colunas específicas de uma tabela, deve-se especificar o(s) nome(s) da(s) coluna(s), como mostrado adiante:

```
SELECT <Coluna1, Coluna2, ...> FROM <tabela>
```

# SQL 2014 - Módulo I

O código a seguir consulta todas as colunas da tabela EMPREGADOS:



- A - Efeito do comando **USE PEDIDOS**. Também é possível selecionar o banco de dados por aqui;
- B - Instrução que queremos executar. É necessário selecionar o comando antes de executar;
- C - Botão que executa o comando selecionado. É importante saber que se nada estiver selecionado, o SQL tentará executar todos os comandos do script;
- D - Resultado da execução do comando **SELECT**.

Veja o seguinte exemplo, em que é feita a consulta nas colunas **CODFUN**, **NOME** e **SALARIO** da tabela **EMPREGADOS**:

```
SELECT CODFUN, NOME, SALARIO FROM EMPREGADOS;
```

Confira o resultado:

	CODFUN	NOME	SALARIO
1	1	OLAVO TRINDADE	3000.00
2	2	JOSE REIS	600.00
3	3	MARCELO SOARES	2400.00
4	4	PAULO CESAR JUNIOR	600.00
5	5	JOAO LIMA MACHADO DA SILVA	1200.00
6	7	CARLOS ALBERTO SILVA	4500.00
7	8	ELIANE PEREIRA	1200.00
8	9	RUDGE RAMOS SANTANA DA PENHA	800.00

O próximo exemplo efetua cálculos gerando colunas virtuais (não existentes fisicamente nas tabelas):

```
SELECT CODFUN, NOME, SALARIO, SALARIO * 1.10
FROM EMPREGADOS;
```

Veja o resultado:

	CODFUN	NOME	SALARIO	(Nenhum nome de colu...)
1	1	CLAVO TRINDADE	3000.00	3300.0000
2	2	JOSE REIS	600.00	660.0000
3	3	MARCELO SOARES	2400.00	2640.0000
4	4	PAULO CESAR JUNIOR	600.00	660.0000
5	5	JOAO LIMA MACHADO DA SILVA	1200.00	1320.0000
6	7	CARLOS ALBERTO SILVA	4500.00	4950.0000
7	8	ELIANE PEREIRA	1200.00	1320.0000

Observe que não existe identificação para a coluna calculada.

## 4.2.3. Redefinindo os identificadores de coluna com uso de alias

O nome de uma coluna ou tabela pode ser substituído por uma espécie de apelido, que é criado para facilitar a visualização. Esse apelido é chamado de alias.

Costuma-se utilizar a cláusula **AS** a fim de facilitar a identificação do alias, no entanto, não é uma obrigatoriedade. A sintaxe para a utilização de alias é descrita a seguir:

```
SELECT <Coluna1> [ [AS] <nome_alias>],  
       <Coluna2> [ [AS] <nome_alias>] [, ...]  
FROM <tabela>
```

Veja os seguintes exemplos de consulta com uso de alias:

- **Definindo um título para a coluna calculada**

```
SELECT CODFUN, NOME, SALARIO,  
       SALARIO * 1.10 AS SALARIO_MAIS_10_POR_CENTO  
FROM EMPREGADOS;
```

Confira o resultado:

CODFUN	NOME	SALARIO	SALARIO_MAIS_10_POR_CEN...
1	OLAVO TRINDADE	3000.00	3300.0000
2	JOSE REIS	600.00	660.0000
3	MARCELO SOARES	2400.00	2640.0000
4	PAULO CESAR JUNIOR	600.00	660.0000
5	JOAO LIMA MACHADO DA SILVA	1200.00	1320.0000
6	CARLOS ALBERTO SILVA	4500.00	4950.0000
7	ELIANE PEREIRA	1200.00	1320.0000
8	RUDGE RAMOS SANTANA DA PENHA	800.00	880.0000
9	MARIA CARMEM	1200.00	1320.0000
10	FERNANDO OLIVEIRA	1200.00	1320.0000
11	JOAO ROBERTO OLIVEIRA	1200.00	1320.0000
12	OSMAR PRADO	2400.00	2640.0000
13	CASSIANO OLIVEIRA	1200.00	1320.0000
14	MARCO ANTONIO	2400.00	2640.0000
15	ALTAMIR CARCIO	3300.00	3630.0000
16	ANA LUISA MARIA	1200.00	1320.0000

Na verdade, qualquer coluna da tabela pode receber um alias:

```
SELECT CODFUN AS Código,  
NOME AS Nome, SALARIO AS Salario  
FROM EMPREGADOS;
```

Se o alias contiver caracteres como espaço, ou outros caracteres especiais, o SQL gera erro, a não ser que este nome seja delimitado por colchetes, apóstrofo ou aspas:

```
SELECT CODFUN AS Código, NOME AS Nome, SALARIO AS Salario,  
       DATA_ADMISSAO AS [Data de Admissão]  
FROM EMPREGADOS;  
-- ou  
SELECT CODFUN AS Código, NOME AS Nome, SALARIO AS Salario,  
       DATA_ADMISSAO AS 'Data de Admissão'  
FROM EMPREGADOS;  
-- ou  
SELECT CODFUN AS Código, NOME AS Nome, SALARIO AS Salario,  
       DATA_ADMISSAO AS "Data de Admissão"  
FROM EMPREGADOS;  
  
-- Campo calculado  
SELECT CODFUN AS Código,  
       NOME AS Nome,  
       SALARIO AS Salario,  
       SALARIO * 1.10 [Salário com 10% de Aumento]  
FROM EMPREGADOS
```

## 4.3. Ordenando dados

Utilizamos a cláusula **ORDER BY** em conjunto com o comando **SELECT** para retornar os dados em uma determinada ordem.

### 4.3.1. Retornando linhas na ordem ascendente

A cláusula **ORDER BY** pode ser utilizada com a opção **ASC**, que faz com que as linhas sejam retornadas em ordem ascendente. Caso a opção **ASC** seja omitida, o padrão da ordenação será ascendente.

Veja um exemplo:

```
SELECT * FROM EMPREGADOS ORDER BY NOME;  
SELECT * FROM EMPREGADOS ORDER BY NOME ASC;  
SELECT * FROM EMPREGADOS ORDER BY SALARIO;  
SELECT * FROM EMPREGADOS ORDER BY SALARIO ASC;  
  
SELECT * FROM EMPREGADOS ORDER BY DATA_ADMISSAO;
```

### 4.3.2. Retornando linhas na ordem descendente

A cláusula **ORDER BY** pode ser utilizada com a opção **DESC**, a qual faz com que as linhas sejam retornadas em ordem descendente.

Veja um exemplo:

```
SELECT * FROM EMPREGADOS ORDER BY NOME DESC;  
SELECT * FROM EMPREGADOS ORDER BY SALARIO DESC;  
SELECT * FROM EMPREGADOS ORDER BY DATA_ADMISSAO DESC;
```

Caso não especifiquemos **ASC** ou **DESC**, os dados da tabela serão retornados em ordem ascendente.

### 4.3.3. Ordenando por nome, alias ou posição

É possível utilizar a cláusula **ORDER BY** para ordenar dados retornados. Para isso, utilizamos como identificação da coluna a ser ordenada o seu próprio nome físico (caso exista), o seu alias ou a posição em que aparece na lista do **SELECT**.

- **Usando o alias ou a posição da coluna como identificação do campo ordenado**

```
-- Pela coluna SALARIO
SELECT CODFUN AS Código,
       NOME AS Nome,
       SALARIO AS Salário,
       SALARIO * 1.10 [Salário com 10% de aumento]
FROM EMPREGADOS
ORDER BY Salário;

-- Idem anterior
SELECT CODFUN AS Código,
       NOME AS Nome,
       SALARIO AS Salário,
       SALARIO * 1.10 [Salário com 10% de aumento]
FROM EMPREGADOS
ORDER BY 3;

-- Pela coluna SALARIO * 1.10
SELECT CODFUN AS Código,
       NOME AS Nome,
       SALARIO AS Salário,
       SALARIO * 1.10 [Salário com 10% de aumento]
FROM EMPREGADOS
ORDER BY [Salário com 10% de Aumento];

-- Idem anterior
SELECT CODFUN AS Código,
       NOME AS Nome,
       SALARIO AS Salário,
       SALARIO * 1.10 [Salário com 10% de aumento]
FROM EMPREGADOS
ORDER BY 4;
```

# SQL 2014 - Módulo I

Veja outro exemplo de retorno de dados de acordo com o nome da coluna:

```
SELECT CODFUN, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY SALARIO;  
  
--  
SELECT CODFUN, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY DATA_ADMISSAO;
```

- **Ordenando por várias colunas**

Quando a coluna ordenada contém informação repetida, essa informação formará grupos. Observe o exemplo:

```
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO;
```

COD_DEP...	NOME	DATA_ADMISSAO	SALARIO
9    1	ROGÉRIO FREITAS	1980-01-01 00:00:00.000	4500.00
10    1	RONALDO MATIAS	1990-01-01 00:00:00.000	3300.00
11    1	JOSÉ CARLOS MOREIRA	2000-01-01 00:00:00.000	8300.00
12    1	JOÃO CARLOS DE OLIVEIRA	2000-01-01 00:00:00.000	5000.00
13    1	JOSÉ CARLOS SILVA	1998-01-01 00:00:00.000	3300.00
14    1	CASSIANO OLIVEIRA	1993-02-03 00:00:00.000	1200.00
15    1	ROBERTO PINHEIRO	1981-12-12 00:00:00.000	8300.00
16    2	SEBASTIÃO SILVA	1988-04-06 00:00:00.000	8300.00
17    2	EURICO BRANDÃO	1988-01-09 00:00:00.000	800.00
18    2	JOSE REIS	1987-05-02 00:00:00.000	600.00
19    2	RUDGE RAMOS SANTANA DA PENHA	1985-12-23 00:00:00.000	800.00
20    2	MARIA DA PENHA	1983-07-15 00:00:00.000	4500.00
21    2	MARIANO DE OLIVEIRA	1993-04-03 00:00:00.000	3330.00
22    2	LUIS FERNANDO LEMOS	2005-01-01 00:00:00.000	600.00
23    2	JOAQUIM ALBERTO	2003-04-05 00:00:00.000	500.00
24    3	MARIANA DA SILVA	2006-01-01 00:00:00.000	500.00

Nesse caso, pode ser útil ordenar outra coluna dentro do grupo formado pela primeira:

```
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO, NOME;
```

	COD_DEPTO	NOME	DATA_ADMISSAO	SALARIO
1	NULL	JORGE DOS SANTOS ROCHA JUNIOR	2000-07-01 00:00:00.000	NULL
2	NULL	SEVERINO CARLOS MACIEIRA	2000-07-01 00:00:00.000	NULL
3	1	ARNALDO MOURA	1990-01-01 00:00:00.000	890.00
4	1	CASSIANO OLIVEIRA	1993-02-03 00:00:00.000	1200.00
5	1	JOÃO CARLOS DE OLIVEIRA	2000-01-01 00:00:00.000	5000.00
6	1	JORGE ROBERTO SOUZA	2001-10-10 00:00:00.000	4500.00
7	1	JOSÉ CARLOS MOREIRA	2000-01-01 00:00:00.000	8300.00
8	1	JOSÉ CARLOS SILVA	1998-01-01 00:00:00.000	3300.00
9	1	PEDRO PAULO SOUZA	2006-06-24 00:00:00.000	890.00
10	1	ROBERTO CARLOS DA SILVA	2006-06-24 00:00:00.000	4500.00
11	1	ROBERTO MARILDO	2001-09-11 00:00:00.000	800.00
12	1	ROBERTO PINHEIRO	1981-12-12 00:00:00.000	8300.00
13	1	ROGÉRIO FREITAS	1980-01-01 00:00:00.000	4500.00
14	1	RONALDO MATIAS	1990-01-01 00:00:00.000	3300.00
15	2	EURICO BRANDÃO	1988-01-09 00:00:00.000	800.00
16	2	JOAQUIM ALBERTO	2003-04-05 00:00:00.000	500.00

# SQL 2014 - Módulo I

Note que, dentro de cada departamento, os dados estão ordenados pela coluna **NOME**:

```
--  
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO, SALARIO;  
  
--  
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO, DATA_ADMISSAO;  
-- Continua valendo o uso do "alias" ou da posição da  
-- coluna  
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY 1, 3;
```

O uso da opção **DESC** (ordenação descendente) é independente para cada coluna no **ORDER BY**:

```
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO DESC, SALARIO;  
  
--  
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO, SALARIO DESC;  
  
--  
SELECT COD_DEPTO, NOME, DATA_ADMISSAO, SALARIO  
FROM EMPREGADOS  
ORDER BY COD_DEPTO DESC, SALARIO DESC;
```

### 4.3.4. ORDER BY com TOP

A cláusula **TOP** mostra as **N** primeiras linhas de um **SELECT**, no entanto, se a usarmos sem a cláusula **ORDER BY**, o resultado fica sem sentido. Veja o exemplo:

```
-- Lista os 5 primeiros empregados de acordo com a chave
-- primária
SELECT TOP 5 * FROM EMPREGADOS;
```

	CODFUN	NOME	NUM_DEPE...	DATA_NASCIMENTO	COD_DEP...	COD_CAR...
1	1	OLAVO TRINDADE	1	1950-06-06 00:00:00.000	4	17
2	2	JOSE REIS	6	1952-10-09 00:00:00.000	2	14
3	3	MARCELO SOARES	1	1950-06-06 00:00:00.000	5	2
4	4	PAULO CESAR JUNIOR	2	1952-03-19 00:00:00.000	8	14
5	5	JOAO LIMA MACHADO DA SILVA	2	1955-10-30 00:00:00.000	4	3

Não sabemos quem são esses cinco funcionários listados. Provavelmente, são os primeiros a serem inseridos na tabela **EMPREGADOS**, mas nem isso podemos afirmar com certeza.

Já nos exemplos a seguir, conseguimos compreender o resultado:

```
-- Lista os 5 empregados mais antigos
SELECT TOP 5 * FROM EMPREGADOS
ORDER BY DATA_ADMISSAO;
```

```
-- Lista os 5 empregados mais novos
SELECT TOP 5 * FROM EMPREGADOS
ORDER BY DATA_ADMISSAO DESC;
```

```
-- Lista os 5 empregados que ganham menos
SELECT TOP 5 * FROM EMPREGADOS
ORDER BY SALARIO;
```

```
-- Lista os 5 empregados que ganham mais
SELECT TOP 5 * FROM EMPREGADOS
ORDER BY SALARIO DESC;
```

## 4.3.5. ORDER BY com TOP WITH TIES

**TOP WITH TIES** é permitida apenas em instruções **SELECT** e quando uma cláusula **ORDER BY** é especificada. Indica que se o conteúdo do campo ordenado na última linha da cláusula **TOP** se repetir em outras linhas, estas deverão ser exibidas também.

Observe a sequência:

```
SELECT CODFUN, NOME, SALARIO  
FROM EMPREGADOS  
ORDER BY SALARIO DESC;
```

	CODFUN	NOME	SALARIO
1	18	ROBERTO PINHEIRO	8300.00
2	19	SEBASTIÃO SILVA	8300.00
3	66	JOSÉ CARLOS MOREIRA	8300.00
4	43	JOÃO CARLOS DE OLIVEIRA	5000.00
5	26	ANA MARIA OLIVEIRA	5000.00
6	27	RICARDO SOUZA	5000.00
7	25	MARIA DA PENHA	4500.00
8	7	CARLOS ALBERTO SILVA	4500.00
9	53	ROGÉRIO FREITAS	4500.00
10	51	JORGE ROBERTO SOUZA	4500.00
11	59	MANOEL RIBEIRO	4500.00
12	72	ROBERTO CARLOS DA SILVA	4500.00
13	58	ALBERTO HELENA SILVA	3330.00
14	28	MARIANO DE OLIVEIRA	3330.00
15	16	ALTAMIR CARCIO	3300.00
16	31	MANOEL SANTOS	3300.00

Este exemplo lista os empregados em ordem descendente de salário. Note que no sétimo registro o salário é de 4500.00 e este valor se repete nos cinco registros seguintes. Se aplicarmos a cláusula **TOP 7**, qual dos seis funcionários com salário de 4500.00 será mostrado, já que o valor é o mesmo?

```
-- Listar os 7 funcionários que ganham mais  
SELECT TOP 7 CODFUN, NOME, SALARIO  
FROM EMPREGADOS  
ORDER BY SALARIO DESC;
```

	CODFUN	NOME	SALARIO
1	18	ROBERTO PINHEIRO	8300.00
2	19	SEBASTIÃO SILVA	8300.00
3	66	JOSÉ CARLOS MOREIRA	8300.00
4	26	ANA MARIA OLIVEIRA	5000.00
5	27	RICARDO SOUZA	5000.00
6	43	JOÃO CARLOS DE OLIVEIRA	5000.00
7	7	CARLOS ALBERTO SILVA	4500.00

Por qual razão o SQL selecionou o funcionário de **CODFUN 7** como último da lista, se existem outros cinco funcionários com o mesmo salário? Porque ele tem a menor chave primária.

# SQL 2014 - Módulo I

Na maioria das consultas, quando um fato como esse ocorre (empate na última linha), o critério para desempate, se houver, dificilmente será pela menor chave primária. Então, seria interessante que a consulta mostrasse todas as linhas em que o salário fosse o mesmo da última:

```
-- Listar os 7 empregados que ganham mais, inclusive  
-- aqueles empatados com o último  
SELECT TOP 7 WITH TIES CODFUN, NOME, SALARIO FROM EMPREGADOS  
ORDER BY SALARIO DESC;
```

	CODFUN	NOME	SALARIO
1	18	ROBERTO PINHEIRO	8300.00
2	19	SEBASTIÃO SILVA	8300.00
3	66	JOSÉ CARLOS MOREIRA	8300.00
4	43	JOÃO CARLOS DE OLIVEIRA	5000.00
5	26	ANA MARIA OLIVEIRA	5000.00
6	27	RICARDO SOUZA	5000.00
7	25	MARIA DA PENHA	4500.00
8	7	CARLOS ALBERTO SILVA	4500.00
9	53	ROGÉRIO FREITAS	4500.00
10	51	JORGE ROBERTO SOUZA	4500.00
11	59	MANOEL RIBEIRO	4500.00
12	72	ROBERTO CARLOS DA SILVA	4500.00

Também podemos usar a cláusula **TOP** com percentual. A tabela EMPREGADOS possui sessenta linhas, então, se pedirmos pra ver 10% das linhas, deverão aparecer seis. Confira:

```
-- Mostrar 10% das linhas da tabela EMPREGADOS  
SELECT TOP 10 PERCENT CODFUN, NOME, SALARIO FROM EMPREGADOS  
ORDER BY SALARIO DESC;
```

São exibidas as seguintes linhas:

	CODFUN	NOME	SALARIO
1	18	ROBERTO PINHEIRO	8300.00
2	19	SEBASTIÃO SILVA	8300.00
3	66	JOSÉ CARLOS MOREIRA	8300.00
4	43	JOÃO CARLOS DE OLIVEIRA	5000.00
5	26	ANA MARIA OLIVEIRA	5000.00
6	27	RICARDO SOUZA	5000.00

Query executed successfully.

SOLSERVER1 (11.0 RTM) | SA (54) | PEDIDOS | 00:00:00 | 6 rows

## 4.4. Filtrando consultas

O exemplo a seguir demonstra o que vimos até aqui sobre a instrução **SELECT**:

```
SELECT [TOP (n) | PERCENT] [WITH TIES]
      <lista_de_colunas> | *
  FROM <nome_da_tabela>
  [WHERE <criterio_de_filtro>]
  [ORDER BY <coluna1> [ASC | DESC] [, <coluna2> [ASC | DESC] [, ...]]]
```

A cláusula **WHERE** determina um critério de filtro e que somente as linhas que respeitem esse critério sejam exibidas. A expressão contida no critério de filtro deve retornar **TRUE** (verdadeiro) ou **FALSE** (falso).

## 4.5. Operadores relacionais

A tabela a seguir exibe os operadores relacionais:

Operador	Descrição
=	Compara, se igual.
<> ou !=	Compara, se diferentes.
>	Compara, se maior que.
<	Compara, se menor que.
>=	Compara, se maior que ou igual.
<=	Compara, se menor que ou igual.



Operadores relacionais sempre terão dois operandos, um à esquerda e outro à sua direita.

Considere os seguintes exemplos:

- Mostrando os funcionários com SALÁRIO abaixo de 1000

```
SELECT CODFUN, NOME, COD_CARGO, SALARIO FROM EMPREGADOS  
WHERE SALARIO < 1000  
ORDER BY SALARIO;
```

- Mostrando os funcionários com SALÁRIO acima de 5000

```
SELECT CODFUN, NOME, COD_CARGO, SALARIO FROM EMPREGADOS  
WHERE SALARIO > 5000  
ORDER BY SALARIO;
```

- Mostrando os funcionários com campo COD\_DEPTO menor ou igual a 3

```
SELECT * FROM EMPREGADOS  
WHERE COD_DEPTO <= 3  
ORDER BY COD_DEPTO;
```

- Mostrando os funcionários com campo COD\_DEPTO igual a 2

```
SELECT * FROM EMPREGADOS
WHERE COD_DEPTO = 2
ORDER BY COD_DEPTO;
```

- Mostrando os funcionários com campo COD\_DEPTO diferente de 2

```
SELECT * FROM EMPREGADOS
WHERE COD_DEPTO <> 2
ORDER BY COD_DEPTO;
```

Embora pareça estranho, os sinais relacionais também podem ser usados para campos alfanuméricos. Veja os seguintes exemplos:

- Mostrando os funcionários que tenham NOME alfabeticamente maior que RAQUEL

```
SELECT CODFUN, NOME, SALARIO
FROM EMPREGADOS
WHERE NOME > 'RAQUEL'
ORDER BY NOME;
```

CODFUN	NOME	SALARIO
1	69 RENAN CARLOS DE OLIVEIRA	3300.00
2	27 RICARDO SOUZA	5000.00
3	72 ROBERTO CARLOS DA SILVA	4500.00
4	49 ROBERTO MARILDO	800.00
5	18 ROBERTO PINHEIRO	8300.00
6	53 ROGÉRIO FREITAS	4500.00
7	61 RONALDO MATIAS	3300.00
8	9 RUDGE RAMOS SANTANA DA PENHA	800.00
9	19 SEBASTIÃO SILVA	8300.00
10	68 SEVERINO CARLOS MACIEIRA	NULL
11	21 SILVIO OLIVEIRA	500.00

- Mostrando os funcionários que tenham NOME alfabeticamente menor que ELIANA

```
SELECT CODFUN, NOME, SALARIO  
FROM EMPREGADOS  
WHERE NOME < 'ELIANA'  
ORDER BY NOME;
```

	CODFUN	NOME	SALARIO
1	58	ALBERTO HELENA SILVA	3330.00
2	16	ALTAMIR CARCIO	3300.00
3	17	ANA LUISA MARIA	1200.00
4	47	ANA LUIZA SOUSA	800.00
5	26	ANA MARIA OLIVEIRA	5000.00
6	57	ANTONIO CARLOS	500.00
7	55	ARLINDO SOARES	500.00
8	48	ARMANDO LEMOS BRITO	1200.00
9	22	ARNALDO FARIA	800.00
10	52	ARNALDO MOURA	890.00
11	50	AUGUSTO SILVEIRA DA SILVA	890.00
12	7	CARLOS ALBERTO SILVA	4500.00
13	30	CARLOS MAGNO P SOUZA	1200.00
14	29	CARLOS ROBERTO DA SILVA	2400.00
15	46	CARLOS ROBERTO JUNIOR	3300.00
16	14	CASSIANO OLIVEIRA	1200.00

## 4.6. Operadores lógicos

A filtragem de dados em uma consulta também pode ocorrer com a utilização dos operadores lógicos **AND**, **OR** ou **NOT**, cada qual permitindo uma combinação específica de expressões, conforme apresentado adiante:

- O operador **AND** combina duas expressões e exige que sejam verdadeiras, ou seja, **TRUE**;
- O operador **OR** verifica se pelo menos uma das expressões retornam **TRUE**;
- O operador **NOT** inverte o resultado lógico da expressão à sua direita, ou seja, se a expressão é verdadeira ele retorna falso e vice-versa.

Acompanhe os seguintes exemplos:

- **Mostrando funcionários do departamento 2 que ganhem mais de 5000**

```
SELECT * FROM EMPREGADOS
WHERE COD_DEPTO = 2 AND SALARIO > 5000;
```

	CODFUN	NOME	NUM_DEPEND	DATA_NASCIMENTO	COD_DEPTO	COD_CARGO	DATA_ADMISS
1	19	SEBASTIÃO SILVA	0	1951-12-19 00:00:00.000	2	1	1988-04-06 00:

- **Mostrando funcionários do departamento 2 ou aqueles que ganhem mais de 5000**

```
SELECT * FROM EMPREGADOS
WHERE COD_DEPTO = 2 OR SALARIO > 5000;
```

CODFUN	NOME	NUM_DEPE...	DATA_NAS...
1	JOSE REIS	6	1952-10-09
2	RUDGE RAMOS SANTANA DA PENHA	3	1961-07-22
3	ROBERTO PINHEIRO	4	1950-04-12
4	SEBASTIÃO SILVA	0	1951-12-19
5	EURICO BRANDÃO	0	1950-04-19
6	MARIA DA PENHA	0	1958-02-12
7	MARIANO DE OLIVEIRA	3	1954-01-18
8	LUIS FERNANDO LEMOS	0	1978-07-23
9	JOAQUIM ALBERTO	0	1991-03-04
10	JOSÉ CARLOS MOREIRA	0	1900-01-01

# SQL 2014 - Módulo I

É importante saber onde utilizar o **AND** e o **OR**. Vamos supor que foi pedido para listar todos os funcionários do **COD\_DEPTO** igual a 2 e também igual a 5. Se fossemos escrever o comando exatamente como foi pedido, digitaríamos o seguinte:

```
SELECT * FROM EMPREGADOS  
WHERE COD_DEPTO = 2 AND COD_DEPTO = 5;
```

No entanto, essa consulta não vai produzir nenhuma linha de resultado. Isso porque um mesmo empregado não está cadastrado nos departamentos 2 e 5 simultaneamente. Um empregado está cadastrado ou (**OR**) no departamento 2 ou (**OR**) no departamento 5.

Precisamos entender que a pessoa que solicita a consulta está visualizando o resultado pronto e acaba utilizando “e” (**AND**) no lugar de “ou” (**OR**). Sendo assim, é importante saber que, na execução do **SELECT**, ele avalia os dados linha por linha. E então, o correto é o seguinte:

```
SELECT * FROM EMPREGADOS  
WHERE COD_DEPTO = 2 OR COD_DEPTO = 5;
```

Veja outros exemplos da utilização de **AND** e **OR**:

- **Mostrando funcionários com SALARIO entre 3000 e 5000**

```
SELECT * FROM EMPREGADOS  
WHERE SALARIO >= 3000 AND SALARIO <= 5000  
ORDER BY SALARIO;
```

- **Mostrando funcionários com SALARIO abaixo de 3000 ou acima de 5000**

```
SELECT * FROM EMPREGADOS  
WHERE SALARIO < 3000 OR SALARIO > 5000  
ORDER BY SALARIO;  
-- Também pode ser feito usando o operador NOT. Aqueles  
-- que não estão entre 3000 e 5000, estão fora dessa  
-- faixa.  
SELECT * FROM EMPREGADOS  
WHERE NOT (SALARIO >= 3000 AND SALARIO <= 5000)  
ORDER BY SALARIO;
```

## 4.7. Consultando intervalos com BETWEEN

A cláusula **BETWEEN** permite filtrar dados em uma consulta tendo como base uma faixa de valores, ou seja, um intervalo entre um valor menor e outro maior. Podemos utilizá-la no lugar de uma cláusula **WHERE** com várias expressões contendo os operadores  $\geq$  e  $\leq$  ou interligadas pelo operador **OR**.

A funcionalidade da cláusula **BETWEEN** assemelha-se à dos operadores **AND**,  $\geq$  e  $\leq$ , no entanto, vale considerar que, por meio dela, a consulta torna-se ainda mais simples de ser realizada.

Os dois comandos a seguir são equivalentes, ou seja, exibem o mesmo resultado:

- **Funcionários com SALARIO entre 3000 e 5000**

```
SELECT * FROM EMPREGADOS  
WHERE SALARIO >= 3000 AND SALARIO <= 5000  
ORDER BY SALARIO;  
  
SELECT * FROM EMPREGADOS  
WHERE SALARIO BETWEEN 3000 AND 5000  
ORDER BY SALARIO;
```

O operador **BETWEEN** também pode ser usado para dados do tipo data ou alfanuméricos:

```
SELECT * FROM EMPREGADOS  
WHERE DATA_ADMISSAO BETWEEN '2000.1.1' AND '2000.12.31'  
ORDER BY DATA_ADMISSAO;
```

Além de **BETWEEN**, podemos utilizar **NOT BETWEEN**, que permite consultar os valores que não se encontram em uma determinada faixa de valores. O exemplo a seguir pesquisa valores não compreendidos no intervalo especificado:

```
SELECT * FROM EMPREGADOS  
WHERE SALARIO < 3000 OR SALARIO > 5000  
ORDER BY SALARIO;
```

Em vez de <, > e **OR**, podemos utilizar **NOT BETWEEN** mais o operador **AND** para pesquisar os mesmo valores da consulta anterior:

```
SELECT * FROM EMPREGADOS  
WHERE SALARIO NOT BETWEEN 3000 AND 5000  
ORDER BY SALARIO;  
-- OU ENTÃO  
SELECT * FROM EMPREGADOS  
WHERE NOT SALARIO BETWEEN 3000 AND 5000  
ORDER BY SALARIO;
```

## 4.8. Consulta com base em caracteres

O operador **LIKE** é usado para fazer pesquisas em dados do tipo string (**CHAR**, **VARCHAR**, **NCHAR** e **NVARCHAR**). É útil quando não sabemos de forma exata o dado que queremos pesquisar. Por exemplo, sabemos que o nome da pessoa começa com MARIA, mas não sabemos o restante do nome, ou sabemos que o nome contém a palavra RAMOS, mas não sabemos o nome completo.

Veja os seguintes exemplos:

- **Nomes que começam com MARIA**

```
SELECT * FROM EMPREGADOS  
WHERE NOME LIKE 'MARIA%';
```

O sinal % é um curinga que equivale a uma quantidade qualquer de caracteres, inclusive nenhum.

- **Nomes que começam com MA**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE 'MA%';
```

- **Nomes que começam com M**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE 'M%';
```

Na verdade, esse recurso de consulta pode buscar palavras que estejam contidas no texto, seja no início, no meio ou no final dele. Acompanhe outros exemplos:

- **Nomes terminando com MARIA**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE '%MARIA';
```

- **Nomes terminando com SOUZA**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE '%SOUZA';
```

- **Nomes terminando com ZA**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE '%ZA';
```

- **Nomes contendo a palavra MARIA**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE '%MARIA%';
```

- **Nomes contendo a palavra SOUZA**

```
SELECT * FROM EMPREGADOS
WHERE NOME LIKE '%SOUZA%';
```

Outro caractere curinga que pode ser utilizado em consultas com **LIKE** é o subscrito (\_), que equivale a um único caractere qualquer. Veja alguns exemplos:

- Nomes iniciados por qualquer caractere, mas que o segundo caractere seja a letra A

```
SELECT * FROM Empregados  
WHERE NOME LIKE '_A%';
```

- Nomes cujo penúltimo caractere seja a letra Z

```
SELECT * FROM Empregados  
WHERE NOME LIKE '%Z_';
```

- Nomes terminados em LU e seguidos de 3 outras letras

```
SELECT * FROM Empregados  
WHERE NOME LIKE '%LU__';
```

Podemos, também, fornecer várias opções para um determinado caractere da chave de busca, como no exemplo a seguir, que busca nomes que contenham **SOUZA** ou **SOUSA**:

```
SELECT * FROM EMPREGADOS  
WHERE NOME LIKE '%SOU[SZ]A%';
```

Veja outro exemplo, que busca nomes contendo **JOSÉ** ou **JOSE**:

```
SELECT * FROM EMPREGADOS  
WHERE NOME LIKE '%JOS[EÉ]%' ;
```

Além disso, podemos utilizar o operador **NOT LIKE**, que atua de forma oposta ao operador **LIKE**. Com **NOT LIKE**, obtemos como resultado de uma consulta os valores que não possuem os caracteres ou sílabas determinadas.

O exemplo a seguir busca nomes que não contenham a palavra **MARIA**:

```
SELECT * FROM EMPREGADOS  
WHERE NOME NOT LIKE '%MARIA%';
```

O exemplo a seguir busca nomes que não contenham a sílaba **MA**:

```
SELECT * FROM EMPREGADOS  
WHERE NOME NOT LIKE '%MA%';
```

## 4.9. Consultando valores pertencentes ou não a uma lista de elementos

O operador **IN**, que pode ser utilizado no lugar do operador **OR** em determinadas situações, permite verificar se o valor de uma coluna está presente em uma lista de elementos.

O operador **NOT IN**, por sua vez, ao contrário de **IN**, permite obter como resultado o valor de uma coluna que não pertence a uma determinada lista de elementos.

O exemplo a seguir busca todos os empregados cujo código do departamento (**COD\_DEPTO**) seja 1, 3, 4 ou 7:

```
SELECT * FROM EMPREGADOS  
WHERE COD_DEPTO IN (1,3,4,7)  
ORDER BY COD_DEPTO;
```

O exemplo adiante busca, nas colunas **NOME** e **ESTADO** da tabela **CLIENTES**, os clientes dos estados do Amazonas (AM), Paraná (PR), Rio de Janeiro (RJ) e São Paulo (SP):

```
SELECT NOME, ESTADO FROM CLIENTES  
WHERE ESTADO IN ('AM', 'PR', 'RJ', 'SP');
```

Já o próximo exemplo busca, nas colunas **NOME** e **ESTADO** da tabela **CLIENTES**, os clientes que não são dos estados do Amazonas (AM), Paraná (PR), Rio de Janeiro (RJ) e São Paulo (SP):

```
SELECT NOME, ESTADO FROM CLIENTES  
WHERE ESTADO NOT IN ('AM', 'PR', 'RJ', 'SP');
```

## 4.10. Lidando com valores nulos

Quando um **INSERT** não faz referência a uma coluna existente em uma tabela, o conteúdo dessa coluna ficará nulo (**NULL**):

```
-- Este INSERT menciona apenas a coluna NOME,  
-- as outras colunas da tabela ficarão  
-- com seu conteúdo NULO  
INSERT INTO EMPREGADOS (NOME)  
VALUES ('JOSE EMANUEL');  
  
-- Ver o resultado  
SELECT * FROM EMPREGADOS;
```

Confira o resultado:

59	70	PEDRO PAULO SOUZA	0	1980-07-01 00:00:00.000	1	6
60	72	ROBERTO CARLOS DA SILVA	0	2000-01-01 00:00:00.000	1	11
61	73	JOSE EMANUEL	NULL	NULL	NULL	NULL

Com relação a valores nulos, vale considerar as seguintes informações:

- Não é um valor zero, nem uma string vazia. É **NULL**;
- Valores nulos não aparecem quando o campo faz parte da cláusula **WHERE**, a não ser que a cláusula deixe explícito que deseja visualizar também os nulos;
- Se envolvido em cálculos, retorna sempre um valor **NULL**.

Observe a consulta a seguir e note que o valor nulo que inserimos anteriormente não aparece nem entre os menores salários e nem entre os maiores:

```
SELECT CODFUN, NOME, SALARIO FROM EMPREGADOS
WHERE SALARIO < 800 OR SALARIO > 8000
ORDER BY SALARIO;
```

	CODFUN	NOME	SALARIO
1	21	SILVIO OLIVEIRA	500.00
2	23	JOAQUIM JUNIOR FILHO	500.00
3	35	MARIANA DA SILVA	500.00
4	40	JOAQUIM ALBERTO	500.00
5	45	MARIA LUIZA	500.00
6	55	ARLINDO SOARES	500.00
7	57	ANTONIO CARLOS	500.00
8	38	LUIS FERNANDO LEMOS	600.00
9	2	JOSE REIS	600.00
10	4	PAULO CESAR JUNIOR	600.00
11	18	ROBERTO PINHEIRO	8300.00
12	19	SEBASTIÃO SILVA	8300.00
13	66	JOSÉ CARLOS MOREIRA	8300.00

Caso faça parte de cálculo, o resultado é sempre **NULL**:

```
SELECT CODFUN, NOME, SALARIO, PREMIO_MENSAL,
       SALARIO + PREMIO_MENSAL AS RENDA_TOTAL
FROM EMPREGADOS

-- filtrar somente os nulos
WHERE SALARIO IS NULL;
```

	CODFUN	NOME	SALARIO	PREMIO_MENSAL	RENDATOTAL
1	44	JORGE DOS SANTOS ROCHA JUNIOR	NULL	528.71	NULL
2	68	SEVERINO CARLOS MACIEIRA	NULL	528.71	NULL
3	73	JOSE EMANUEL	NULL	NULL	NULL

Para lidar com valores nulos, evitando problemas no resultado final da consulta, pode-se empregar funções como **IS NULL**, **IS NOT NULL**, **NULIF** e **COALESCE**, as quais serão apresentadas nos tópicos subsequentes.

O exemplo a seguir busca, na tabela **EMPREGADOS**, os registros cujo **COD\_CARGO** seja nulo:

```
SELECT * FROM EMPREGADOS  
WHERE COD_CARGO IS NULL;
```

Este exemplo busca, na tabela **EMPREGADOS**, os registros cuja **DATA\_NASCIMENTO** seja nula:

```
SELECT * FROM EMPREGADOS  
WHERE DATA_NASCIMENTO IS NULL;
```

O exemplo adiante busca, na tabela **EMPREGADOS**, os registros cuja **DATA\_NASCIMENTO** não seja nula:

```
SELECT * FROM EMPREGADOS  
WHERE DATA_NASCIMENTO IS NOT NULL;
```

## 4.11. Substituindo valores nulos

Em síntese, as funções **ISNULL** e **COALESCE** permitem retornar outros valores quando valores nulos são encontrados durante a filtragem de dados. Adiante, apresentaremos a descrição dessas funções.

### 4.11.1. ISNULL

Esta função permite definir um valor alternativo que será retornado, caso o valor de argumento seja nulo. Veja os exemplos adiante:

- Exemplo 1

```
SELECT
    CODFUN, NOME, SALARIO, PREMIO_MENSAL,
    ISNULL(SALARIO,0) + ISNULL(PREMIO_MENSAL,0) AS RENDA_TOTAL
FROM EMPREGADOS
WHERE SALARIO IS NULL;
```

	CODFUN	NOME	SALARIO	PREMIO_MENSAL	RENDATOTAL
1	44	JORGE DOS SANTOS ROCHA JUNIOR	NULL	528.71	528.71
2	68	SEVERINO CARLOS MACIEIRA	NULL	528.71	528.71
3	73	JOSE EMANUEL	NULL	NULL	0.00

- Exemplo 2

```
SELECT
    CODFUN, NOME,
    ISNULL(DATA_NASCIMENTO,'1900.1.1') AS DATA_NASC
FROM EMPREGADOS;
```

## 4.11.2. COALESCE

Esta função é responsável por retornar o primeiro argumento não nulo em uma lista de argumentos testados.

O código a seguir tenta exibir o campo **EST\_COB** dos clientes da tabela **CLIENTES**. Caso esse campo seja **NULL**, o código tenta exibir o campo **ESTADO**. Caso este último também seja **NULL**, será retornado **NC**:

```
SELECT
    CODCLI, NOME, COALESCE(EST_COB, ESTADO, 'NC') AS EST_COBRANCA
FROM CLIENTES
ORDER BY 3;
```

### 4.12. Manipulando campos do tipo datetime

O tipo de dado **datetime** é utilizado para definir valores de data e hora. Aceita valores entre 1º de janeiro de 1753 até 31 de dezembro de 9999. O formato no qual digitamos a data depende de configurações do servidor ou usuário.

Veja algumas funções utilizadas para retornar dados desse tipo:

- **SET DATEFORMAT**

É utilizada para determinar a forma de digitação de data/hora durante uma sessão de trabalho.

```
SET DATEFORMAT (ordem)
```

A ordem das porções de data é definida por **ordem**, que pode ser um dos valores da tabela adiante:

Valor	Ordem
<b>mdy</b>	Mês, dia e ano (Formato padrão americano).
<b>dmy</b>	Dia, mês e ano.
<b>ymd</b>	Ano, mês e dia.
<b>ydm</b>	Ano, dia e mês.
<b>myd</b>	Mês, ano e dia.
<b>dym</b>	Dia, ano e mês.

O exemplo a seguir determina o formato ano/mês/dia para o valor de data a ser retornado:

```
SET DATEFORMAT YMD;
```

- **GETDATE()**

Retorna a data e hora atual do sistema, sem considerar o intervalo de fuso-horário. O valor é derivado do sistema operacional do computador no qual o SQL Server é executado:

```
SELECT GETDATE() AS DATA_ATUAL;
```

	DATA_ATUAL
1	2013-09-13 11:53:01.113

Para sabermos qual será a data daqui a 45 dias, utilizamos o seguinte código:

```
SELECT GETDATE() + 45;
```

Já no código a seguir, **GETDATE()** é utilizado para saber a quantos dias cada funcionário da tabela **EMPREGADOS** foi admitido:

```
SELECT CODFUN, NOME, CAST(GETDATE() - DATA_ADMISSAO AS INT) AS DIAS_NA_EMPRESA  
FROM EMPREGADOS
```

- **DAY()**

Esta função retorna um valor inteiro que representa o dia da data especificada como argumento **data** na sintaxe adiante:

```
DAY(data)
```

# SQL 2014 - Módulo I

O argumento **data** pode ser uma expressão, literal de string, variável definida pelo usuário ou expressão de coluna.

Veja os exemplos a seguir:

```
-- Número do dia correspondente à data de hoje  
SELECT DAY(GETDATE());  
  
-- Todos os funcionários admitidos no dia primeiro de  
-- qualquer mês e qualquer ano  
SELECT * FROM EMPREGADOS  
WHERE DAY(DATA_ADMISSAO) = 1;
```

- **MONTH()**

Esta função retorna um valor inteiro que representa o mês da data especificada como argumento **data** da sintaxe adiante:

```
MONTH(data)
```

O argumento **data** pode ser uma expressão, literal de string, variável definida pelo usuário ou expressão de coluna.

Veja os exemplos:

```
-- Número do mês correspondente à data de hoje  
SELECT MONTH(GETDATE())  
  
-- Empregados admitidos em dezembro  
SELECT * FROM EMPREGADOS  
WHERE MONTH(DATA_ADMISSAO) = 12
```

- **YEAR()**

Esta função retorna um valor inteiro que representa o ano da data especificada como argumento **data** na sintaxe adiante:

```
YEAR(data)
```

O argumento **data** pode ser uma expressão, literal de string, variável definida pelo usuário ou expressão de coluna.

O exemplo a seguir retorna os empregados admitidos no ano 2000:

```
SELECT * FROM EMPREGADOS
WHERE YEAR (DATA_ADMISSAO) = 2000;
```

Já o exemplo a seguir retorna os empregados admitidos no mês de dezembro do ano 2000:

```
SELECT * FROM EMPREGADOS
WHERE YEAR (DATA_ADMISSAO) = 2000 AND
MONTH (DATA_ADMISSAO) = 12;
```

- **DATEPART()**

Esta função retorna um valor inteiro que representa uma porção (especificada no argumento **parte**) de data ou hora definida no argumento **data** da sintaxe adiante:

```
DATEPART (parte, data)
```

O argumento **data** pode ser uma expressão, literal de string, variável definida pelo usuário ou expressão de coluna, enquanto **parte** pode ser um dos valores descritos na próxima tabela:

Valor	Parte retornada	Abreviação
<b>year</b>	Ano	yy, yyyy
<b>quarter</b>	Trimestre (1/4 de ano)	qq, q
<b>month</b>	Mês	mm, m
<b>dayofyear</b>	Dia do ano	dy, y
<b>day</b>	Dia	dd, d
<b>week</b>	Semana	wk, ww
<b>weekday</b>	Dia da semana	dw
<b>hour</b>	Hora	hh
<b>minute</b>	Minuto	mi, n

Valor	Parte retornada	Abreviação
<b>second</b>	Segundo	<b>ss, s</b>
<b>millisecond</b>	Milissegundo	<b>ms</b>
<b>microsecond</b>	Microssegundo	<b>mcs</b>
<b>nanosecond</b>	Nanossegundo	<b>ns</b>
<b>TZoffset</b>	Diferença de fuso-horário	<b>tz</b>
<b>ISO_WEEK</b>	Retorna a numeração da semana associada a um ano	<b>isowk, isoww</b>

Vale dizer que o resultado retornado será o mesmo, independentemente de termos especificado um valor ou a respectiva abreviação.

O exemplo a seguir utiliza **DATEPART** para retornar os empregados admitidos em dezembro de 2000. Para isso, são especificadas as partes de ano (**YEAR**) e mês (**MONTH**):

```
SELECT * FROM EMPREGADOS  
WHERE DATEPART(YEAR, DATA_ADMISSAO) = 2000 AND  
DATEPART(MONTH, DATA_ADMISSAO) = 12;
```

- **DATENAME()**

Esta função retorna como resultado uma string de caracteres que representa uma porção da data ou hora definida no argumento **data** da sintaxe adiante:

```
DATENAME (parte, data)
```

O argumento **data** pode ser uma expressão, literal de string, variável definida pelo usuário ou expressão de coluna, enquanto **parte**, que representa a referida porção, pode ser um dos valores descritos na tabela anterior.

O exemplo a seguir é utilizado para obter os funcionários que foram admitidos em uma sexta-feira:

```
-- Funcionários admitidos em uma sexta-feira
SELECT
    CODFUN, NOME, DATA_ADMISSAO,
    DATENAME(WEEKDAY, DATA_ADMISSAO) AS DIA_SEMANA,
    DATENAME(MONTH, DATA_ADMISSAO) AS MES
FROM EMPREGADOS
WHERE DATEPART(WEEKDAY, DATA_ADMISSAO) = 6;
```

O resultado retornado por **DATENAME()** dependerá do idioma configurado no servidor SQL.

- **DATEADD()**

Esta função retorna um novo valor de **datetime** ao adicionar um intervalo a uma porção da data ou hora definida no argumento **data** da sintaxe adiante:

```
DATEADD(parte, numero, data)
```

O argumento **parte** é a porção de data ou hora que receberá o acréscimo definido em **numero**. O argumento **data** pode ser uma expressão, literal de string, variável definida pelo usuário ou expressão de coluna, enquanto **parte** pode ser um dos valores descritos na tabela anterior, com exceção de **TZoffset**.

O código a seguir retorna o dia de hoje mais 45 dias:

```
SELECT DATEADD( DAY, 45, GETDATE() );
```

Este código retorna o dia de hoje mais 6 meses:

```
SELECT DATEADD( MONTH, 6, GETDATE() );
```

# SQL 2014 - Módulo I

Já o código a seguir retorna o dia de hoje mais 2 anos:

```
SELECT DATEADD( YEAR, 2, GETDATE() );
```

- **DATEDIFF()**

Esta função obtém como resultado um número de data ou hora referente aos limites de uma porção de data ou hora, cruzados entre duas datas especificadas nos argumentos **data\_inicio** e **data\_final** da seguinte sintaxe:

```
DATEDIFF(parte, data_inicio, data_final)
```

O argumento **parte** representa a porção de **data\_inicio** e **data\_final** que especificará o limite cruzado.

O exemplo a seguir retorna a quantidade de dias vividos até hoje por uma pessoa nascida em 12 de novembro de 1959:

```
SELECT DATEDIFF( DAY, '1959.11.12', GETDATE() );
```

O exemplo a seguir retorna a quantidade de meses vividos até hoje pela pessoa citada no exemplo anterior:

```
SELECT DATEDIFF( MONTH, '1959.11.12', GETDATE() );
```

O exemplo a seguir retorna a quantidade de anos vividos até hoje por essa mesma pessoa:

```
SELECT DATEDIFF( YEAR, '1959.11.12', GETDATE() );
```

Na utilização de **DATEDIFF()** para obter a diferença em anos ou meses, o valor retornado não é exato. O código a seguir retorna 1. No entanto, a diferença somente seria 1 no dia 20 de fevereiro de 2009, veja:

```
SELECT DATEDIFF( MONTH, '2013.1.20', '2013.2.15' );
```

No próximo exemplo, o resultado retornado também é 1, mas a diferença somente seria 1 no dia 20 de junho de 2009:

```
SELECT DATEDIFF( YEAR, '2012.12.20', '2013.1.15' );
```

- **DATEFROMPARTS()**

Esta função retorna uma data (DATE) a partir dos parâmetros ano, mês e dia especificados nos argumentos da seguinte sintaxe:

```
DATEFROMPARTS (ano, mês, dia)
```

No exemplo a seguir, vamos retornar a data 25 de dezembro de 2013 a partir dos seguintes parâmetros:

```
SELECT DATEFROMPARTS (2013,12,25);
```

O resultado é mostrado a seguir:

(No column name)	2013-12-25
1	2013-12-25

- **TIMEFROMPARTS()**

Esta função retorna um horário (TIME) a partir dos parâmetros hora, minuto, segundo, milissegundo e precisão dos milissegundos especificados nos argumentos da seguinte sintaxe:

```
TIMEFROMPARTS (hora, minuto, segundo, milissegundo, precisão)
```

No exemplo a seguir, vamos retornar o horário 10:25:15 a partir dos seguintes parâmetros:

```
SELECT TIMEFROMPARTS (10,25,15,0,0);
```

O resultado é mostrado a seguir:

(No column name)	
1	10:25:15

- **DATETIMEFROMPARTS()**

Esta função retorna um valor de data e hora (DATETIME) a partir dos parâmetros ano, mês, dia, hora, minuto, segundo e milissegundo da seguinte sintaxe:

```
DATETIMEFROMPARTS (ano, mês, dia, hora, minuto, segundo, milissegundo);
```

Caso algum valor esteja incorreto, a função retornará um erro. Agora, se o parâmetro for nulo, a função retornará valor nulo. No exemplo a seguir, vamos retornar o valor 15 de setembro de 2013 às 14:00:15.000:

```
SELECT DATETIMEFROMPARTS (2013,9,15,14,0,15,0);
```

O resultado é mostrado a seguir:

(No column name)	
1	2013-09-15 14:00:15.000

- **DATETIME2FROMPARTS()**

Retorna um valor do tipo **datetime2** a partir dos parâmetros ano, mês, dia, hora, minuto, segundo, fração de segundo e a precisão da fração de segundo da seguinte sintaxe:

```
DATETIME2FROMPARTS (ano, mês, dia, hora, minuto, segundo, fração, precisão);
```

Caso a precisão receba o valor 0, é necessário que a indicação de milissegundos também seja 0, senão a função retornará um erro. No exemplo a seguir, vamos retornar o valor 10 de outubro de 2013 às 21:00:59.0000:

```
SELECT DATETIME2FROMPARTS (2013,10,10,21,0,59,0,0);
```

O resultado é mostrado a seguir:

Results	
(No column name)	
1	2013-10-10 21:00:59

- **SMALLDATETIMEFROMPARTS()**

Esta função retorna um valor do tipo **smalldatetime** a partir dos parâmetros ano, mês, dia, hora e minuto da seguinte sintaxe:

```
SMALLDATETIMEFROMPARTS (ano, mês, dia, hora, minuto);
```

No exemplo a seguir, vamos retornar a data 17 de setembro de 2013 às 10:25:00:

```
SELECT SMALLDATETIMEFROMPARTS (2013,9,17,10,25);
```

O resultado é mostrado a seguir:

Results	
(No column name)	
1	2013-09-17 10:25:00

- **DATETIMEOFFSETFROMPARTS()**

Esta função retorna um valor do tipo **datetimeoffset** representando um deslocamento de fuso horário a partir dos parâmetros ano, mês, dia, hora, minuto, segundo, fração, deslocamento de hora, deslocamento de minuto e a precisão decimal dos segundos:

```
DATETIMEOFFSETFROMPARTS (ano, mês, dia, hora, minuto, segundo,  
fração, desloc_hora, desloc_minuto, precisão);
```

No exemplo a seguir, vamos representar o deslocamento de 12 horas de fuso sem frações de segundos na data 17 de setembro de 2013 às 13:22:00:

```
SELECT DATETIMEOFFSETFROMPARTS (2013,9,17,13,22,0,0,12,0,0);
```

O resultado é mostrado a seguir:

(No column name)
1 2013-09-17 13:22:00 +12:00

- **EOMonth()**

Esta função retorna o último dia do mês a partir dos parâmetros **data\_início** e **adicionar\_mês** (Opcional) da seguinte sintaxe:

```
EOMONTH (data_início [, adicionar_mês]);
```

O valor retornado é do tipo data (DATE). No exemplo a seguir, vamos retornar o último dia do mês atual:

```
SELECT EOMONTH (GETDATE ()) ;
```

O resultado desse exemplo é mostrado a seguir:

	(No column name)
1	2013-09-30

Para avançar 1 mês, adicione o valor 1 no parâmetro **adicionar\_mês**;

	(No column name)
1	2013-10-31

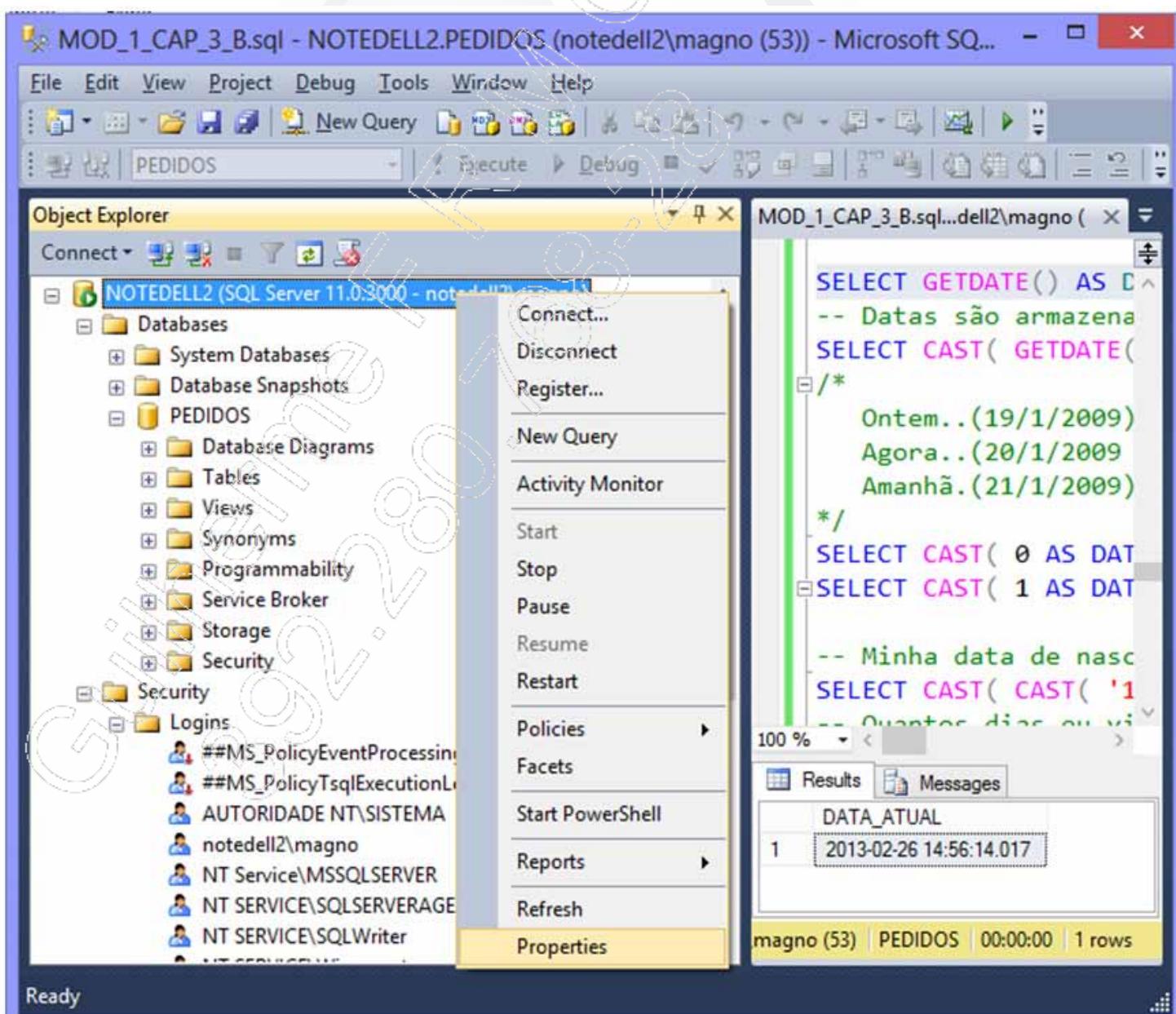
Para voltar 1 mês adicione o valor -1 no parâmetro **adicionar\_mês**;

	(No column name)
1	2013-08-31

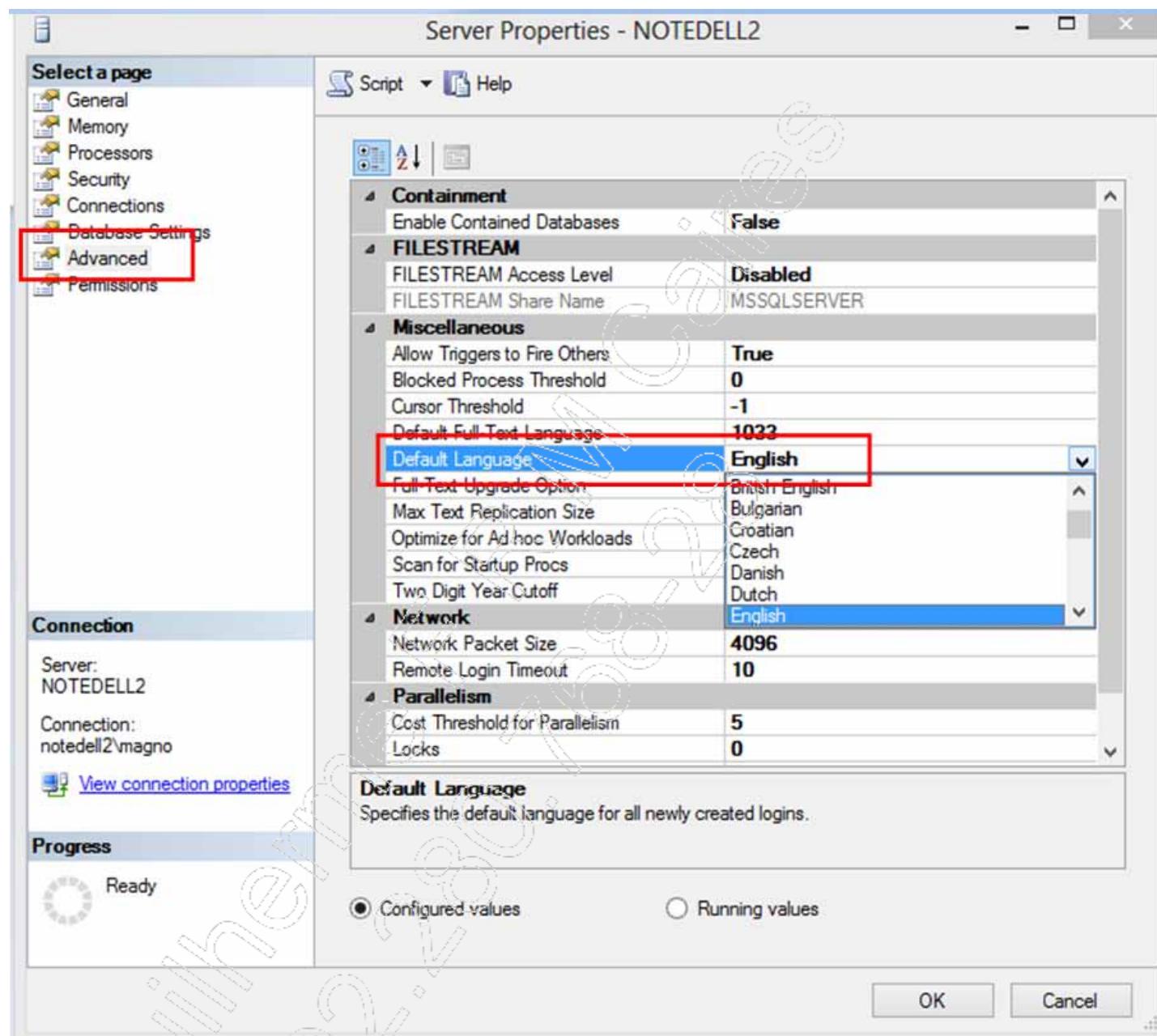
## 4.13. Alterando a configuração de idioma a partir do SSMS

O resultado retornado pela função **DATENAME()** dependerá do idioma do servidor SQL. Então, para que essa função retorne o nome do mês ou do dia da semana da maneira desejada, pode ser necessário alterar o idioma do SQL Server. Isso pode ser feito a partir do SQL Server Management Studio, como descrito no passo-a-passo a seguir:

1. Clique com o botão direito do mouse no nome do servidor, selecionando **Properties** no menu de contexto, como ilustrado a seguir:



2. Na janela seguinte, selecione a opção **Advanced**, procure o item **Default Language** e altere-o para o idioma que desejar:



O formato da data pode variar de acordo com o idioma escolhido:

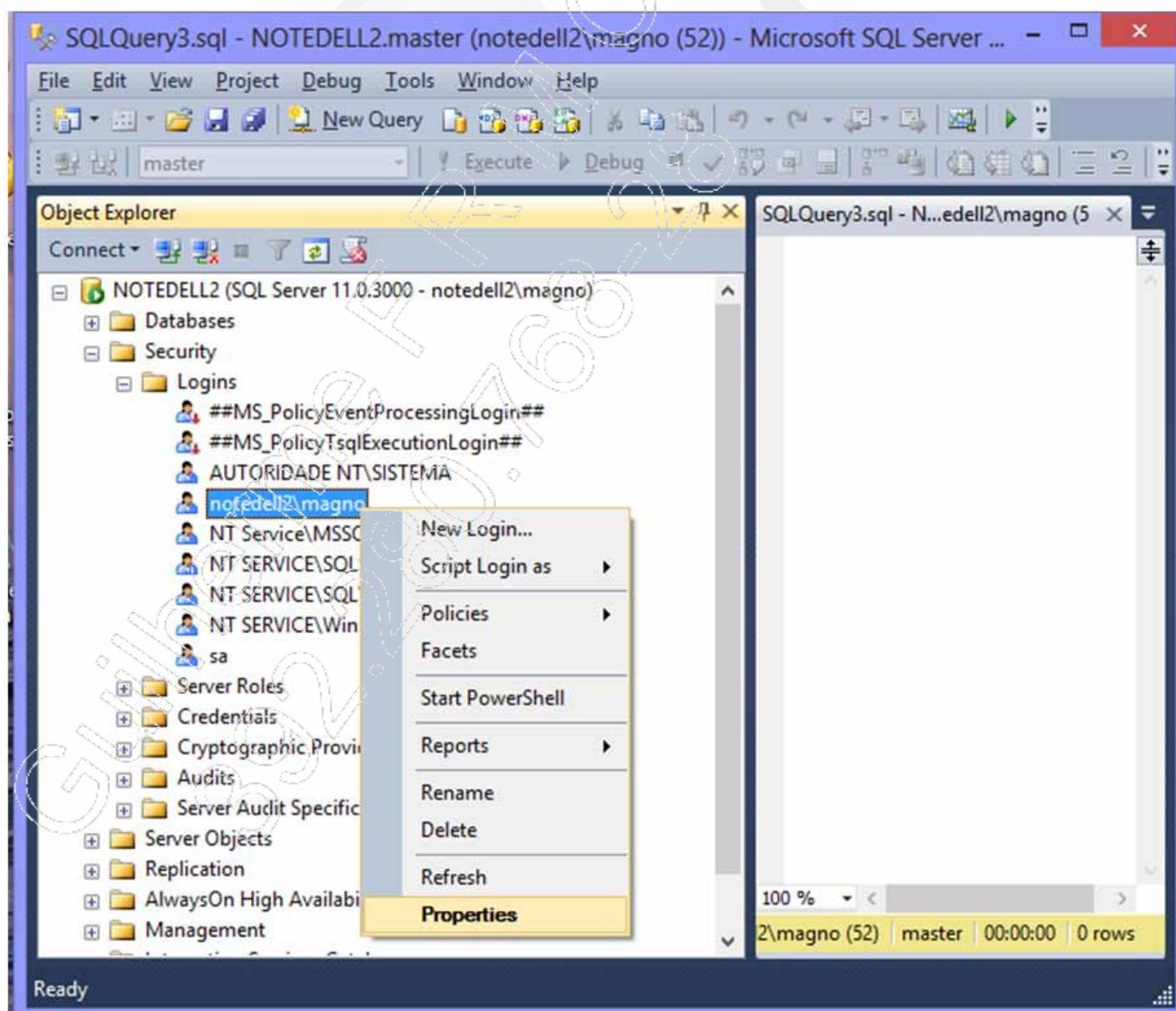
- **Brazilian:** Pode utilizar os formatos dd/mm/yy ou dd/mm/yyyy;
- **English:** Pode utilizar os formatos mm/dd/yy, mm/dd/yyyy ou yyyy.mm.dd.

# SQL 2014 - Módulo I

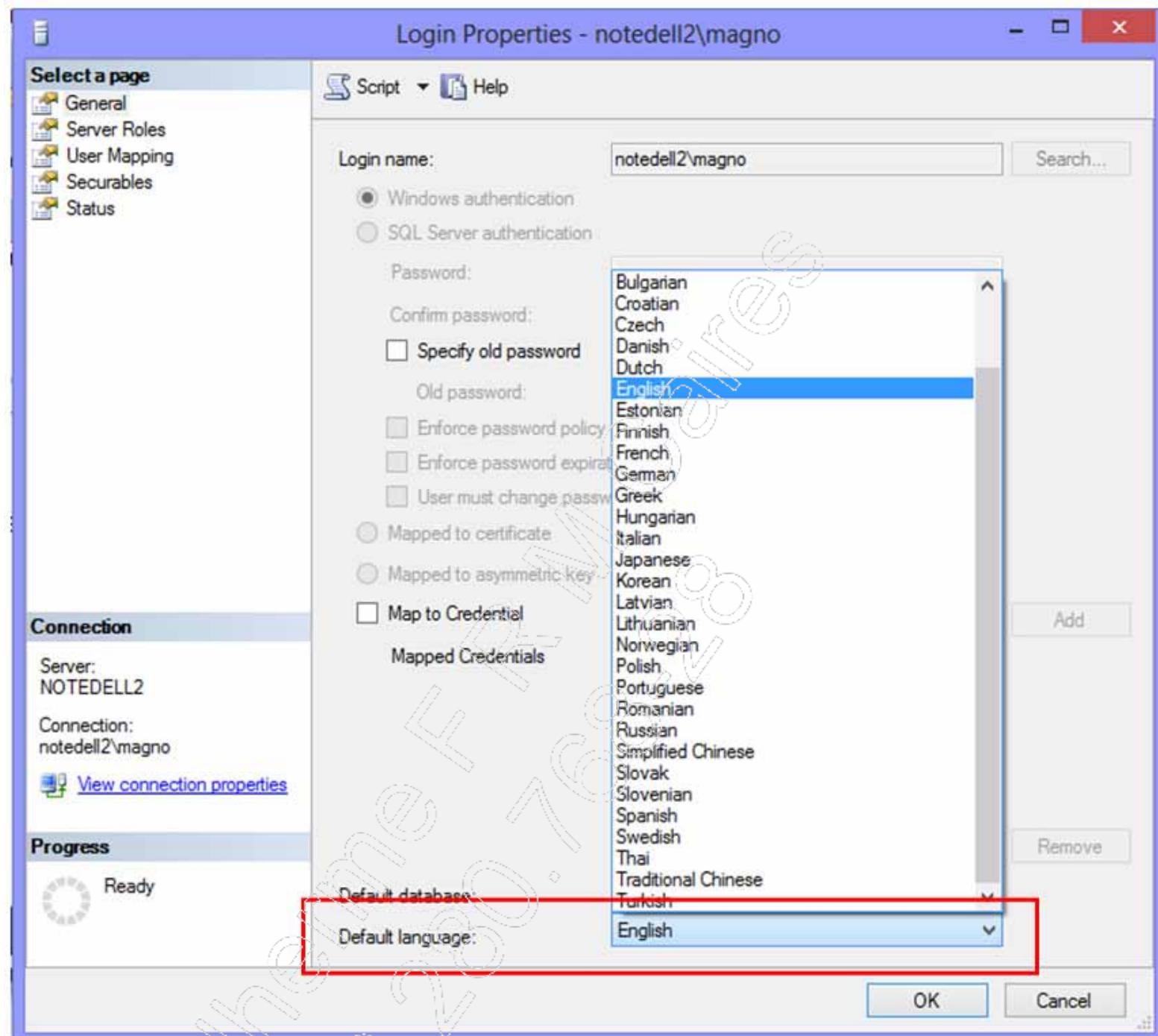
A configuração escolhida será aplicada a todos os logins criados posteriormente. Logins já existentes deverão ser configurados novamente para que possuam o novo formato.

Para alterar a configuração de logins existentes, siga os passos adiante:

1. No navegador do SQL Server Management Studio, abra a pasta **Security** e, em seguida, **Logins**;
2. Clique com o botão direito no login a ser alterado e selecione **Properties** no menu de contexto, conforme ilustrado a seguir:



Será exibida a seguinte janela:



3. Na caixa de seleção **Default Language**, escolha o idioma desejado e clique em **OK**.

Alterar a configuração de idioma do servidor SQL afetará não apenas o valor retornado por DATENAME(), mas todos os aspectos associados ao idioma, como mensagens de erro exibidas no SQL Server Management Studio e o formato de digitação de datas.

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Na linguagem SQL, o principal comando utilizado para a realização de consultas é o **SELECT**. Pertencente à categoria DML (Data Manipulation Language), esse comando é utilizado para consultar todos os dados de uma fonte de dados ou apenas uma parte específica deles;
- Às vezes, é necessário que o resultado da consulta de dados seja fornecido em uma ordem específica, de acordo com um determinado critério. Para isso, contamos com opções e cláusulas. Uma delas é a cláusula **ORDER BY**, que considera certa ordem para retornar dados de consulta;
- A cláusula **WHERE** é utilizada para definir critérios com o objetivo de filtrar o resultado de uma consulta. As condições definidas nessa cláusula podem ter diferentes propósitos, tais como a comparação de dados na fonte de dados, a verificação de dados de determinadas colunas e o teste de colunas nulas ou valores nulos;
- O tipo de dado **datetime** é utilizado para definir valores de data e hora. Esse tipo é baseado no padrão norte-americano, ou seja, os valores devem atender ao modelo **mm/dd/aa** (mês, dia e ano, respectivamente). Dispomos de diversas funções para retornar dados desse tipo, tais como **GETDATE()**, **DAY()**, **MONTH()** e **YEAR()**.

4

# Consultando dados

## Teste seus conhecimentos

Guilherme Caires  
392.280.168-28  
ERP -



**IMPACTA**  
EDITORA

## 1. Verifique o comando a seguir e indique qual afirmação está errada:

```
SELECT CODFUN AS Código,  
       NOME AS Nome,  
       SALARIO AS Salário,  
       SALARIO * 1.10 [Salário com 10% de aumento]  
  FROM EMPREGADOS
```

- a) O comando acima altera a tabela EMPREGADOS devido à coluna calculada.
- b) A instrução apresenta as informações do empregado e uma coluna calculada com aumento de 10%.
- c) O cabeçalho da coluna CODFUN será alterado para CODIGO.
- d) Serão apresentadas apenas 4 colunas da tabela empregados.
- e) Na última coluna será calculado o valor atual do empregado vezes 10%.

2. O comando a seguir apresenta uma consulta na tabela de empregados. Para ordenarmos essa consulta, podemos utilizar algumas das cláusulas apresentadas, porém uma delas está errada. Qual?

```
SELECT CODFUN AS Código,  
       NOME AS Nome,  
       SALARIO AS Salário,  
       SALARIO * 1.10 [Salário com 10% de aumento]  
  FROM EMPREGADOS
```

- a) ORDER BY 4
- b) ORDER BY [Salário com 10% de aumento]
- c) ORDER BY 4 DESC
- d) ORDER BY 2, 4 DESC.
- e) ORDER BY SALARIO DECRESCENT

## 3. Como podemos restringir a quantidade de linhas de um comando SELECT?

- a) Cláusula OUTPUT
- b) SELECT com FROM
- c) SELECT com ORDER BY
- d) SELECT com TOP
- e) Nenhuma das alternativas anteriores está correta.

## 4. A cláusula WHERE permite filtrar o resultado do comando SELECT. Qual comando a seguir seria uma instrução válida para apresentar um cliente de código 10 (Campo CODCLI do tipo INT)?

- a) SELECT \* FROM CLIENTES WHERE CODCLI 10
- b) SELECT \* FROM CLIENTES WHERE CODCLI LIKE '%10%'
- c) SELECT \* FROM CLIENTES WHERE CODCLI
- d) SELECT \* FROM CLIENTES WHERE ISNULL(CODCLI , 0) >10
- e) SELECT \* FROM CLIENTES WHERE CODCLI = 10

## 5. Ao utilizarmos a cláusula TOP com ORDER BY, podemos:

- a) Gerar uma consulta do tipo ranking.
- b) Somente restringir a quantidade de linhas apresentadas na consulta.
- c) Não podemos utilizar a cláusula ORDER BY com TOP.
- d) Ela não altera o resultado do comando.
- e) É obrigatória a utilização da cláusula WITH TIES.

**6. Verifique o comando a seguir e selecione a melhor afirmação:**

```
SELECT * FROM EMPREGADOS  
WHERE SALARIO < 3000 AND SALARIO > 5000  
ORDER BY SALARIO;
```

- a) A consulta retorna todos os empregados que possuem salário menor que 3000 e maior que 5000.
- b) A consulta retorna um erro de sintaxe.
- c) A consulta apresenta os empregados ordenados pelo salário de forma crescente.
- d) A consulta não retornará nenhum valor, pois não é possível atender aos dois critérios.
- e) Nenhuma das alternativas anteriores está correta.

**7. Qual afirmação está errada?**

- a) A função ISNULL retorna um valor não nulo quando a expressão for nula.
- b) A função COALESCE retorna um valor não nulo de uma lista de valores testados.
- c) Um valor nulo não é considerado zero.
- d) Um valor nulo não é considerado um texto em branco.
- e) Não devemos tratar um valor nulo, pois em cálculos são considerados como zero.

**8. Com relação ao uso de um filtro por ano em uma consulta, qual opção está errada?**

- a) Função YEAR.
- b) Função DATEPARTE.
- c) Expressão {campo} >= {1º dia do ano} and {campo} <= {último dia do ano}.
- d) Função DATEPART.
- e) Cláusula WHERE.



4

# Consultando dados

Mãos à obra!

Guilherme Caires  
392.280.1668  
FR-2020

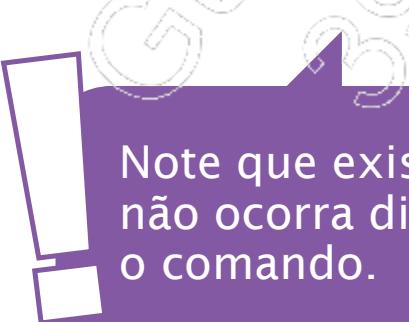


**IMPACTA**  
EDITORA

## Laboratório 1

A – Utilizando o banco de dados **PEDIDOS** e listando suas tabelas com base em diferentes critérios

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste a tabela **PRODUTOS**, mostrando as colunas **COD\_PRODUTO**, **DESCRICAO**, **PRECO\_CUSTO**, **PRECO\_VENDA** e calculando o lucro unitário (**PRECO\_VENDA - PRECO\_CUSTO**);
3. Liste a tabela **PRODUTOS**, mostrando os campos **COD\_PRODUTO**, **DESCRICAO** e calculando o valor total investido no estoque daquele produto (**QTD\_REAL \* PRECO\_CUSTO**);
4. Liste a tabela **ITENSPEDEIDO**, mostrando as colunas **NUM\_PEDIDO**, **NUM\_ITEM**, **COD\_PRODUTO**, **PR\_UNITARIO**, **QUANTIDADE**, **DESCONTO** e calculando o valor de cada item (**PR\_UNITARIO \* QUANTIDADE \* (1-DESCONTO/100)**);
5. Liste a tabela **PRODUTOS**, mostrando as colunas **COD\_PRODUTO**, **DESCRICAO**, **PRECO\_CUSTO**, **PRECO\_VENDA** e calculando lucro estimado em reais (**QTD\_REAL \* (PRECO\_VENDA - PRECO\_CUSTO)**);
6. Liste a tabela **PRODUTOS**, mostrando os campos **COD\_PRODUTO**, **DESCRICAO**, **PRECO\_CUSTO**, **PRECO\_VENDA**, calculando o lucro unitário em reais (**PRECO\_VENDA - PRECO\_CUSTO**) e o lucro unitário percentual (**(100 \* (PRECO\_VENDA - PRECO\_CUSTO) / PRECO\_CUSTO)**).



Note que existe uma divisão na instrução. Deve-se garantir que não ocorra divisão por zero, pois isso provoca erro ao executar o comando.

## Laboratório 2

A – Utilizando o banco de dados **PEDIDOS** e listando suas tabelas com base em novos critérios

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste tabela **PRODUTOS**, criando campo calculado (**QTD\_REAL - QTD\_MINIMA**), e filtre os registros resultantes, mostrando somente aqueles que tiverem a quantidade real abaixo da quantidade mínima;  


Neste caso, o exercício não cita as colunas que devem ser exibidas. Sendo assim, basta utilizar o símbolo asterisco (\*) ou, então, colocar as colunas que julgar importantes.
3. Liste a tabela **PRODUTOS**, mostrando os registros que tenham quantidade real acima de 5000;
4. Liste produtos com preço de venda inferior a R\$0,50;
5. Liste a tabela **PEDIDOS** com valor total (**VLR\_TOTAL**) acima de R\$15.000,00;
6. Liste produtos com **QTD\_REAL** entre 500 e 1000 unidades;
7. Liste pedidos com valor total entre R\$15.000,00 e R\$25.000,00;
8. Liste produtos com quantidade real acima de 5000 e código do tipo igual a 6;
9. Liste produtos com quantidade real acima de 5000 ou código do tipo igual a 6;
10. Liste pedidos com valor total inferior a R\$100,00 ou acima de R\$100.000,00;
11. Liste produtos com **QTD\_REAL** menor que 500 ou maior que 1000.

## Laboratório 3

### A - Utilizando o banco de dados PEDIDOS

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste clientes do estado de São Paulo (SP);
3. Liste clientes dos estados de Minas Gerais e Rio de Janeiro (MG, RJ);
4. Liste clientes dos estados de São Paulo, Minas Gerais e Rio de Janeiro (SP, MG, RJ);
5. Liste os vendedores com o nome **LEIA**;
6. Liste todos os clientes que tenham **NOME** começando com **BRINDES**;
7. Liste todos os clientes que tenham **NOME** terminando com **BRINDES**;
8. Liste todos os clientes que tenham **NOME** contendo **BRINDES**;
9. Liste todos os produtos com **DESCRICAO** começando por **CANETA**;
10. Liste todos os produtos com **DESCRICAO** contendo **SPECIAL**;
11. Liste todos os produtos com **DESCRICAO** terminando por **GOLD**;
12. Liste todos os clientes que tenham a letra **A** como segundo caractere do nome;
13. Liste todos os produtos que tenham **0** (ZERO) como segundo caractere do campo **COD\_PRODUTO**;
14. Liste todos os produtos que tenham a letra **A** como terceiro caractere do campo **COD\_PRODUTO**.

## Laboratório 4

A – Utilizando novamente o banco de dados **PEDIDOS** e listando suas tabelas com base em outros critérios

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste todos os pedidos com data de emissão anterior a Jan/2006;
3. Liste todos os pedidos com data de emissão no primeiro semestre de 2006;
4. Liste todos os pedidos com data de emissão em janeiro e junho de 2006;
5. Liste todos os pedidos do Vendedor Código 1 em Jan/2006;
6. Liste os pedidos emitidos em Jan/2006 em uma sexta-feira.



# Atualizando e excluindo dados

5

- ✓ UPDATE;
- ✓ DELETE;
- ✓ OUTPUT para DELETE e UPDATE;
- ✓ Transações.



**IMPACTA**  
EDITORA

## 5.1. Introdução

Os comandos da categoria Data Manipulation Language, ou DML, são utilizados não apenas para consultar (**SELECT**) e inserir (**INSERT**) dados, mas também para realizar alterações e exclusões de dados presentes em registros. Os comandos responsáveis por alterações e exclusões são, respectivamente, **UPDATE** e **DELETE**.

Para executarmos diferentes comandos ao mesmo tempo, podemos escrevê-los em um só script. Porém, esse procedimento dificulta a correção de eventuais erros na sintaxe. Portanto, recomenda-se executar cada um dos comandos separadamente.

É importante lembrar que os apóstrofos (‘) devem ser utilizados entre as strings de caracteres, com exceção dos dados numéricos. Os valores de cada coluna, por sua vez, devem ser separados por vírgulas.

## 5.2. UPDATE

Os dados pertencentes a múltiplas linhas de uma tabela podem ser alterados por meio do comando **UPDATE**.

Quando utilizamos o comando **UPDATE**, é necessário especificar algumas informações, como o nome da tabela que será atualizada e as colunas cujo conteúdo será alterado. Também, devemos incluir uma expressão ou um determinado valor que realizará essa atualização, bem como algumas condições para determinar quais linhas serão editadas.

A sintaxe de **UPDATE** é a seguinte:

```
UPDATE tabela  
SET nome_coluna = expressao [, nome_coluna = expressao, ...]  
[WHERE condicao]
```

Em que:

- **tabela**: Define a tabela em que dados de uma linha ou grupo de linhas serão alterados;
- **nome\_coluna**: Trata-se do nome da coluna a ser alterada;
- **expressao**: Trata-se da expressão cujo resultado será gravado em **nome\_coluna**. Também pode ser uma constante;
- **condicao**: É a condição de filtragem utilizada para definir as linhas de tabela a serem alteradas.

Este capítulo aborda a utilização simples de **UPDATE** sem as cláusulas **FROM** e **JOIN** que foram omitidas da sintaxe do comando. Em um capítulo posterior, veremos como utilizar esse comando com as cláusulas **FROM/JOIN**.

# SQL 2014 - Módulo I

Nas expressões especificadas com o comando **UPDATE**, costumamos utilizar operadores aritméticos, os quais realizam operações matemáticas, e o operador de atribuição **=**. A tabela a seguir descreve esses operadores:

Operador	Descrição
<b>+</b>	Adição
<b>-</b>	Subtração
<b>*</b>	Multiplicação
<b>/</b>	Divisão
<b>%</b>	Retorna o resto inteiro de uma divisão
<b>=</b>	Atribuição

Tais operadores podem ser combinados, como descreve a próxima tabela:

Operadores	Descrição
<b>+=</b>	Soma e atribui
<b>-=</b>	Subtrai e atribui
<b>*=</b>	Multiplica e atribui
<b>/=</b>	Divide e atribui
<b>%=</b>	Obtém o resto da divisão e atribui

Veja alguns exemplos da utilização desses operadores:

```
-- Operadores
DECLARE @A INT = 10;
SET @A += 5;    -- O mesmo que SET @A = @A + 5;
PRINT @A;
SET @A -= 2;    -- O mesmo que SET @A = @A - 2;
PRINT @A;
SET @A *= 4;    -- O mesmo que SET @A = @A * 4;
PRINT @A;
SET @A /= 2;    -- O mesmo que SET @A = @A / 2;
PRINT @A;
GO
```

## 5.2.1. Alterando dados de uma coluna

O exemplo a seguir descreve como alterar dados de uma coluna:

```
-- Alterar dados de uma coluna
USE PEDIDOS;

-- Aumentar o salário de todos os funcionários em 20%
UPDATE EMPREGADOS
SET SALARIO = SALARIO * 1.2;
-- OU
UPDATE EMPREGADOS
SET SALARIO *= 1.2;

-- Somar 2 na quantidade de dependentes do funcionário de
-- código 5
UPDATE EMPREGADOS
SET NUM_DEPEND = NUM_DEPEND + 2
WHERE CODFUN = 5;
-- OU
UPDATE EMPREGADOS
SET NUM_DEPEND += 2
WHERE CODFUN = 5;
```

## 5.2.2. Alterando dados de diversas colunas

O exemplo a seguir descreve como alterar dados de várias colunas. Será corrigido o endereço do cliente de código 5:

```
SELECT * FROM CLIENTES WHERE CODCLI = 5;

-- Alterar os dados do cliente de código 5
UPDATE CLIENTES
SET ENDERECO = 'AV. PAULISTA, 1009 - 10 AND',
    BAIRRO   = 'CERQUEIRA CESAR',
    CIDADE   = 'SÃO PAULO'
WHERE CODCLI = 5;

-- Conferir o resultado da alteração
SELECT * FROM CLIENTES WHERE CODCLI = 5;
```

No exemplo a seguir, serão corrigidos dados de um grupo de produtos:

```
SELECT * FROM PRODUTOS
WHERE COD_TIPO = 5;

-- Alterar os dados do grupo de produtos
UPDATE PRODUTOS SET QTD_ESTIMADA = QTD_REAL,
                     CLAS_FISC = '96082000',
                     IPI = 8
WHERE COD_TIPO = 5;

-- A linha a seguir confere o resultado da alteração
SELECT * FROM PRODUTOS
WHERE COD_TIPO = 5;
```

## 5.2.3. Utilizando TOP em uma instrução UPDATE

Utilizada em uma instrução **UPDATE**, a cláusula **TOP** define uma quantidade ou porcentagem de linhas que serão atualizadas, conforme o exemplo a seguir, o qual multiplica por 10 o valor de salário de 15 registros da tabela **EMP\_TEMP**:

```
-- Consultar
SELECT * FROM EMP_TEMP;
-- Multiplicar por 10 o valor do SALARIO de 15 registros da tabela
UPDATE TOP(15) EMP_TEMP SET SALARIO = 10*SALARIO;
-- Consultar
SELECT * FROM EMP_TEMP;
```

## 5.3. DELETE

O comando **DELETE** deve ser utilizado quando desejamos excluir os dados de uma tabela. Sua sintaxe é a seguinte:

```
DELETE [FROM] tabela  
[WHERE condicao]
```

Em que:

- **tabela**: É a tabela cuja linha ou grupo de linhas será excluído;
- **condicao**: É a condição de filtragem utilizada para definir as linhas de tabela a serem excluídas.

Uma alternativa ao comando **DELETE**, sem especificação da cláusula **WHERE**, é o uso de **TRUNCATE TABLE**, que também exclui todas as linhas de uma tabela. No entanto, este último apresenta as seguintes vantagens:

- Não realiza o log da exclusão de cada uma das linhas, o que acaba por consumir pouco espaço no log de transações;
- A tabela não fica com páginas de dados vazias;
- Os valores originais da tabela, quando ela foi criada, são restabelecidos, caso haja uma coluna de identidade.

A sintaxe dessa instrução é a seguinte:

```
TRUNCATE TABLE <nome_tabela>
```

## 5.3.1. Excluindo todas as linhas de uma tabela

Podemos excluir todas as linhas de uma tabela com o comando **DELETE**. Nesse caso, não utilizamos a cláusula **WHERE**.

1. Primeiramente, gere uma cópia da tabela **EMPREGADOS**:

```
SELECT * INTO EMPREGADOS_TMP FROM EMPREGADOS;
```

2. Agora, exclua os empregados que ganham mais do que 5000:

```
SELECT * FROM EMPREGADOS_TMP WHERE SALARIO > 5000;  
--  
DELETE FROM EMPREGADOS_TMP WHERE SALARIO > 5000;  
--
```

A linha a seguir verifica se os empregados que ganham mais do que 5000 realmente foram excluídos:

```
SELECT * FROM EMPREGADOS_TMP WHERE SALARIO > 5000;
```

3. A seguir, exclua empregados de código 3, 5 e 7:

```
SELECT * FROM EMPREGADOS_TMP WHERE CODFUN IN (3,5,7);  
--  
DELETE FROM EMPREGADOS_TMP WHERE CODFUN IN (3,5,7);
```

A linha a seguir verifica se os empregados dos referidos códigos realmente foram excluídos:

```
SELECT * FROM EMPREGADOS_TMP WHERE CODFUN IN (3,5,7);
```

4. Já com o código adiante, elimine todos os registros da tabela **EMPREGADOS\_TMP**:

```
DELETE FROM EMPREGADOS_TMP;  
-- OU  
TRUNCATE TABLE EMPREGADOS_TMP;  
--
```

A linha a seguir verifica se todos os registros da referida tabela foram eliminados:

```
SELECT * FROM EMPREGADOS_TMP;
```

### 5.3.2. Utilizando TOP em uma instrução DELETE

Em uma instrução **DELETE**, a cláusula **TOP** define uma quantidade ou porcentagem de linhas a serem removidas de uma tabela, como demonstrado no exemplo adiante, que exclui 10 linhas da tabela **CLIENTES\_MG**:

```
-- Colocar o banco de dados PEDIDOS em uso
USE PEDIDOS;
-- Criar a tabela a partir do comando SELECT INTO
SELECT * INTO CLIENTE_MG FROM CLIENTES;
-- Consultar CLIENTE_MG
SELECT * FROM CLIENTE_MG;

DELETE TOP(10) FROM CLIENTE_MG;
-- Consultar
SELECT * FROM CLIENTE_MG;
```

O resultado do código anterior é o seguinte:

	CODIGO	NOME	ENDERECO	BAIRRO	CIDADE	FONE
1	109	ENFOR LTDA	R.SANTOS,1931	JARDIN AMERICA	BELO HORIZ...	0313732252
2	112	EUDELCIO ALVES FRANCO	AV.TEREZINA,2056	UMUARAMA	UBERLANDIA	0342323152
3	126	GRAFICA ELDORADO LTDA	R.JOAQUIM PEREGRINO,33	NOSSA SENHOR...	PARA DE MI...	0372314577
4	131	HANNAS PERSONALIZACAO	R. JUCA FLAVIA,101	INCONFIDENTES	CONTAGEM	031 3623087
5	165	JOSE ADRIANO MARTINS MA...	R.GERALDO GONCALVES FERREIRA,31	NULL	TEOFILO OT...	0335216983
6	167	JOSE DA LUZ PERIERA ME	PRACA DR. MARCOS FROTA,220	NULL	VARGINHA	0352215115
7	170	LOURIVAL MATOS ASSUNCAO	R.AVES E SILVA,49 SALA 04	NULL	VARGINHA	NULL
8	180	MR SILK SCREEN LTDA	R.CEL JOSE CUSTODIO,48-B CX.POSTAL 59	CENTRO	CAMPEESTRE	035743 1554
9	202	EIDER PERPETUO	R.RIO PARAOPEBA,1364	RIACHO DAS PED...	CONTAGEM	031 3515683
10	227	LORIVAL MATOS ASSUNCAO	R.RIO DE JANEIRO,419	NULL	VARGINHA	035 2222157

Consulta executada com êxito.

SOMA5\SQLEXPRESS2008 (10.0 ... SOMA5\CARLOS MAGNO SOU... PEDIDOS 00:00:00 10 linhas

## 5.4. OUTPUT para DELETE e UPDATE

A sintaxe é a seguinte:

```
DELETE [FROM] <tabela> [OUTPUT deleted.<nomeCampo>|* [, . . . ]]
WHERE <condição>

UPDATE <tabela> SET <campo1> = <expr1> [, . . . ]
[OUTPUT deleted|inserted.<nomeCampo> [, . . . ]]
[WHERE <condição>]
```

Veja alguns exemplos de **OUTPUT**:

```
-- Colocar o banco PEDIDOS em uso
USE PEDIDOS;

-- 1. Gerar uma cópia da tabela EMPREGADOS chamada EMP_TEMP
CREATE TABLE EMP_TEMP
( CODFUN      INT PRIMARY KEY,
  NOME        VARCHAR(30),
  COD_DEPTO   INT,
  COD_CARGO   INT,
  SALARIO     NUMERIC(10,2) )
GO

-- 2. Inserir dados e exibir os registros inseridos
INSERT INTO EMP_TEMP OUTPUT INSERTED.*
SELECT CODFUN, NOME, COD_DEPTO, COD_CARGO, SALARIO
FROM EMPREGADOS;
GO

-- 4. Deletar registros e mostrar os registros deletados
DELETE FROM EMP_TEMP OUTPUT DELETED.*
WHERE COD_DEPTO = 2
GO

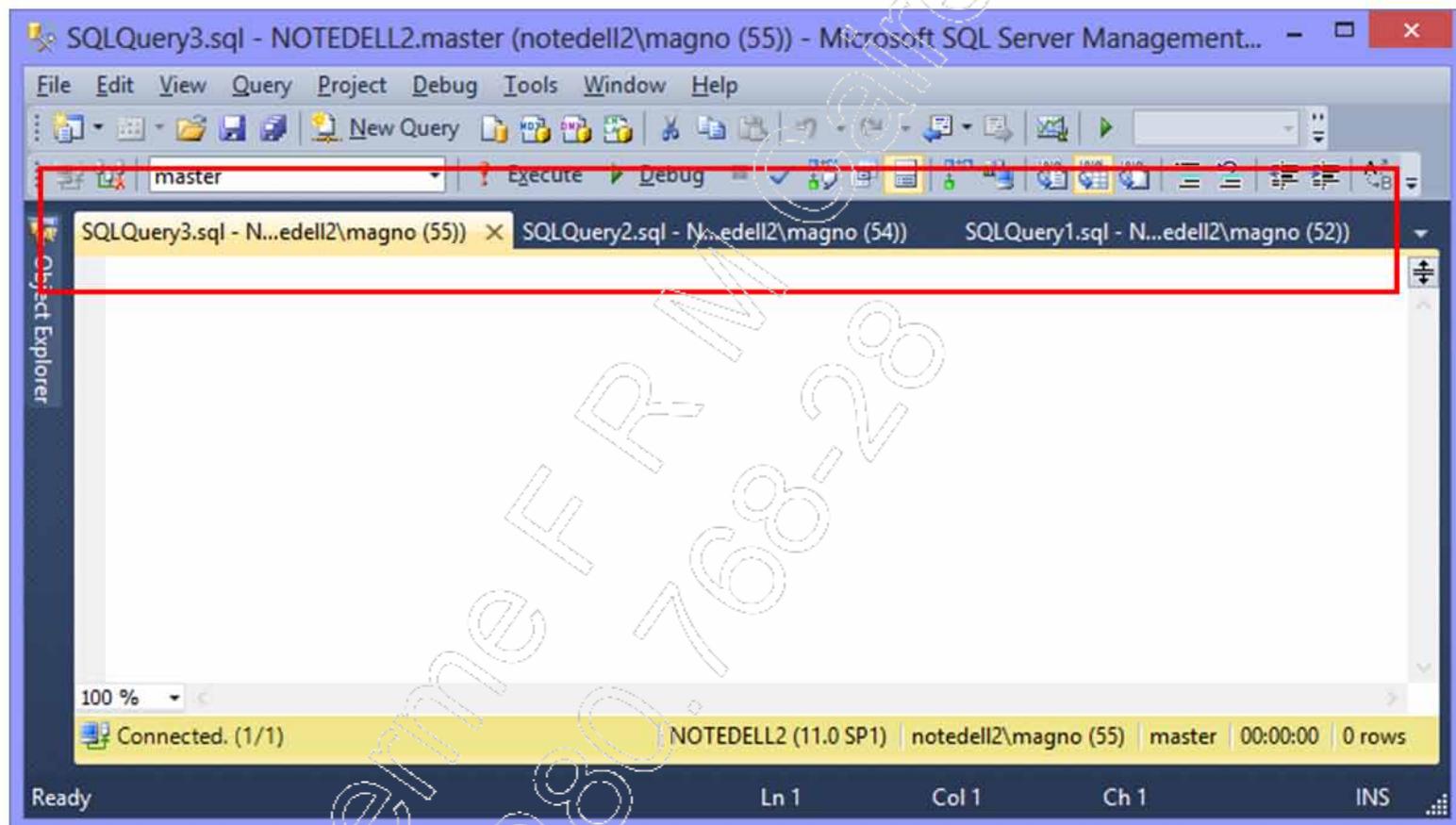
-- 6. Alterar registros e mostrar os dados antes e depois da
-- alteração
UPDATE EMP_TEMP SET SALARIO *= 1.5
OUTPUT
    INSERTED.CODFUN, INSERTED.NOME, INSERTED.COD_DEPTO,
    DELETED.SALARIO AS SALARIO_ANTIGO,
    INSERTED.SALARIO AS SALARIO_NOVO
WHERE COD_DEPTO = 3;
GO
```

Observe que podemos conferir se a alteração foi feita corretamente porque é possível verificar se a condição está correta (**COD\_DEPTO = 3**), e verificar o campo **SALARIO** antes e depois da alteração.

## 5.5. Transações

Quando uma conexão ocorre no MS-SQL, ela recebe um número de sessão. Mesmo que a conexão ocorra a partir da mesma máquina e mesmo login, cada conexão é identificada por um número único de sessão.

Observe a figura a seguir, em que temos três conexões identificadas pelas sessões 52, 54 e 55:



Sobre as transações, é importante considerar as seguintes informações:

- Um processo de transação é aberto por uma sessão e deve também ser fechado pela mesma sessão;
- Durante o processo, as alterações feitas no banco de dados poderão ser efetivadas ou revertidas quando a transação for finalizada;
- Os comandos **DELETE**, **INSERT** e **UPDATE** abrem uma transação de forma automática. Se o comando não provocar erro, ele confirma as alterações no final, caso contrário, descarta todas as alterações.

## 5.5.1. Transações explícitas

As transações explícitas são aquelas em que seu início e seu término são determinados de forma explícita. Para definir este tipo de transação, os scripts Transact-SQL utilizam os seguintes comandos:

- **BEGIN TRANSACTION ou BEGIN TRAN**  
Inicia um processo de transação para a sessão atual.
- **COMMIT TRANSACTION, COMMIT WORK ou simplesmente COMMIT**  
Finaliza o processo de transação atual, confirmando todas as alterações efetuadas desde o início do processo.
- **ROLLBACK TRANSACTION, ROLLBACK WORK ou ROLLBACK**  
Finaliza o processo de transação atual, descartando todas as alterações efetuadas desde o início do processo.

Sobre as transações explícitas, é importante considerar as seguintes informações:

- Se uma conexão for fechada com uma transação aberta, um **ROLLBACK** será executado;
- As operações feitas por um processo de transação ficam armazenadas em um arquivo de log (**.ldf**), que todo banco de dados possui;
- Durante um processo de transação, as linhas das tabelas que foram alteradas ficam bloqueadas para outras sessões.

Veja exemplos de transação:

```
-- Alterar os salários dos funcionários com COD_CARGO = 5
-- para R$950,00
-- Abrir processo de transação
BEGIN TRANSACTION;
-- Verificar se existe processo de transação aberto
SELECT @@TRANCOUNT;
-- Alterar os salários do COD_CARGO = 5
UPDATE Empregados SET SALARIO = 950
OUTPUT inserted.CODFUN, inserted.NOME,
deleted.salario as Salario_Anterior,
inserted.salario as Salario_Atualizado
WHERE COD_CARGO = 5
-- Conferir os resultados na listagem gerada pela cláusula
-- OUTPUT
-- Se estiver tudo OK...
COMMIT TRANSACTION
-- caso contrário
ROLLBACK TRANSACTION
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Os dados pertencentes a múltiplas linhas de uma tabela podem ser alterados por meio do comando **UPDATE**;
- O comando **DELETE** deve ser utilizado quando desejamos excluir os dados de uma tabela;
- 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 executa 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 executada com êxito. No entanto, caso a transação não seja finalizada com êxito por conta de erros, são excluídas quaisquer alterações feitas sobre os dados.

# **Atualizando e excluindo dados**

## **Teste seus conhecimentos**

**5**

Guilherme  
392.280.  
Páginas  
Caires



**IMPACTA**  
EDITORA

## 1. Qual comando deve ser utilizado para alterar um registro?

- a) DELETE
- b) SELECT
- c) TRUNCATE
- d) UPDATE
- e) INSERT

## 2. O que o comando a seguir realizará?

```
UPDATE EMPREGADOS  
SET SALARIO *= 1.2;
```

- a) Definirá que todos os empregados ficarão com o salário de R\$1,20.
- b) Os empregados terão um aumento de 120%.
- c) A tabela de empregado sofrerá um aumento no campo de salário de 1.2.
- d) Os empregados terão um aumento de 20% no salário.
- e) A sintaxe está errada e o SQL emitirá um erro.

### 3. Verifique o comando a seguir e selecione a afirmação correta:

```
UPDATE TOP(15) EMP_TEMP SET SALARIO = 10*SALARIO;
```

- a) A sintaxe está errada.
- b) A cláusula TOP somente pode ser utilizada no comando SELECT.
- c) O comando UPDATE realizará a atualização de todos os registros da tabela EMP\_TEMP.
- d) Serão alterados 15 registros da tabela EMP\_TEMP.
- e) A sintaxe está incompleta.

### 4. Qual comando devemos utilizar para apagar apenas um registro?

- a) DELETE
- b) DELETE ou TRUNCATE
- c) UPDATE
- d) SELECT
- e) INSERT

## 5. Quando utilizamos a cláusula OUTPUT para um comando UPDATE:

- a) Podemos somente verificar o estado do registro depois da alteração.
- b) Podemos somente verificar o estado do registro antes da alteração.
- c) Esse recurso não agrega valor ao comando.
- d) Ao utilizarmos esse comando, podemos comparar a alteração antes e depois da alteração.
- e) Não é possível utilizar esse comando, pois é um recurso antigo.

5

# Atualizando e excluindo dados

## Mãos à obra!

Guilherme  
Caires  
392.280.  
F000  
2020



**IMPACTA**  
EDITORA

## Laboratório 1

### A - Atualizando e excluindo dados

1. Coloque o banco de dados **PEDIDOS** em uso;
2. Aumente o preço de custo de todos os produtos do tipo 2 em 15%;



Utilize transação e a cláusula **OUTPUT** para conferir os resultados.

3. Faça com que os preços de venda dos produtos do tipo 2 fiquem 30% acima do preço de custo;
4. Altere o campo **IPI** de todos os produtos com **COD\_TIPO = 3** para 5%;
5. Reduza em 10% o campo **QTD\_MINIMA** de todos os produtos (multiplique **QTD\_MINIMA** por 0.9);
6. Altere os seguintes campos do cliente de código 11:
  - **ENDERECO**: AV. CELSO GARCIA, 1234;
  - **BAIRRO**: TATUAPE;
  - **CIDADE**: SAO PAULO;
  - **ESTADO**: SP;
  - **CEP**: 03407080.
7. Copie **ENDERECO**, **BAIRRO**, **CIDADE**, **ESTADO** e **CEP** do cliente de código 13 para os campos **END\_COB**, **BAI\_COB**, **CID\_COB**, **EST\_COB** e **CEP\_COB** do mesmo cliente;

8. Altere a tabela **CLIENTES**, mudando o conteúdo do campo **ICMS** de clientes dos estados **RJ, RO, AC, RR, MG, PR, SC, RS, MS e MT** para 12;
9. Altere os campos **ICMS** de todos os clientes de **SP** para 18;
10. Altere o campo **ICMS** para 7 para clientes que não sejam dos estados **RJ, RO, AC, RR, MG, PR, SC, RS, MS, MT** e **SP**;
11. Altere para 7 o campo **DESCONTO** da tabela **ITENSPEIDO**, mas somente dos itens do produto com **ID\_PRODUTO = 8**, com data de entrega em janeiro de 2007 e com **QUANTIDADE** acima de 1000;
12. Zere o campo **DESCONTO** de todos os itens de pedido com quantidade abaixo de 1000, com data de entrega posterior a **1-Junho-2007** e que tenham desconto acima de zero;
13. Usando **SELECT INTO**, gera uma cópia da tabela **VENDEDORES** com o nome de **VENDEDORES\_TMP**;
14. Exclua de **VENDEDORES\_TMP** os registros com **CODVEN** acima de 5;
15. Utilizando o comando **SELECT...INTO**, faça uma cópia da tabela **PEDIDOS** chamada **COPIA\_PEDIDOS**;
16. Exclua os registros da tabela **COPIA\_PEDIDOS** que sejam do vendedor código 2;
17. Exclua os registros da tabela **COPIA\_PEDIDOS** que sejam do primeiro semestre de 2007;
18. Exclua todos os registros restantes da tabela **COPIA\_PEDIDOS**;
19. Exclua a tabela **COPIA\_PEDIDOS** do banco de dados.



# Associando tabelas

# 6

- ✓ INNER JOIN;
- ✓ OUTER JOIN;
- ✓ CROSS JOIN.



**IMPACTA**  
EDITORA

## 6.1. Introdução

A associação de tabelas, ou simplesmente **JOIN** entre tabelas, tem como principal objetivo trazer, em uma única consulta (um único **SELECT**), dados contidos em mais de uma tabela.

Normalmente, essa associação é feita por meio da chave estrangeira de uma tabela com a chave primária da outra. Mas isso não é um pré-requisito para o **JOIN**, de forma que qualquer informação comum entre duas tabelas servirá para associá-las.

Diferentes tipos de associação podem ser escritos com a ajuda das cláusulas **JOIN** e **WHERE**. Por exemplo, podemos obter apenas os dados relacionados entre duas tabelas associadas. Também podemos combinar duas tabelas de forma que seus dados relacionados e não relacionados sejam obtidos.

Basicamente, existem três tipos de **JOIN** que serão vistos neste capítulo: **INNER JOIN**, **OUTER JOIN** e **CROSS JOIN**.

## 6.2. INNER JOIN

A cláusula **INNER JOIN** compara os valores de colunas provenientes de tabelas associadas, utilizando, para isso, operadores de comparação. Por meio desta cláusula, os registros de duas tabelas são utilizados para que sejam gerados os dados relacionados de ambas.

A sintaxe de **INNER JOIN** é a seguinte:

```
SELECT <lista_de_campos>
  FROM <nome_primeira_tabela> [INNER] JOIN <nome_segunda_tabela>
    [ON (condicao)]
```

Em que:

- **condicao**: Define um critério que relaciona as duas tabelas;
- **INNER**: É opcional, se colocarmos apenas JOIN, o INNER já é subentendido.

EMPREGADO				TABELADEP			
CODFUN	NOME	NUM_DEPE...	DATA_NASCIMENTO	COD_DEP...	COD_DEP...	DEPTO	
1	OLAVO TRINDADE	1	1950-06-06 00:00:00.000	4	1	PESSOAL	
2	JOSE REIS	6	1952-10-09 00:00:00.000	2	2	C.P.D.	
3	MARCELO SOARES	1	1950-06-06 00:00:00.000	5	3	CONTROLE DE ESTOQUE	
4	PAULO CESAR JUNIOR	2	1952-03-19 00:00:00.000	8	4	COMPRAS	
5	JOAO LIMA MACHADO DA SILVA	2	1955-10-30 00:00:00.000	4	5	PRODUCAO	
6	CARLOS ALBERTO SILVA	0	1961-07-06 00:00:00.000	11	6	DIRETORIA	
7	ELIANE PEREIRA	0	1955-01-14 00:00:00.000	6	7	TELEMARKETING	
8	RUDGE RAMOS SANTANA DA PENHA	3	1961-07-22 00:00:00.000	2	8	FINANCEIRO	
9	MARIA CARMEM	0	1954-03-14 00:00:00.000	5	9	RECURSOS HUMANOS	
10					10	TREINAMENTO	
					11	PRESIDENCIA	
					12	PORTARIA	
					13	CONTROLADORIA	
					14	P.C.P.	

Observando as duas tabelas, é possível concluir que a funcionária de **CODFUN = 5** trabalha no departamento de **COMPRAS**, cujo código é 4.

A especificação desse tipo de associação pode ocorrer por meio das cláusulas **WHERE** ou **FROM**. Veja os exemplos a seguir:

```
SELECT EMPREGADOS.CODFUN, EMPREGADOS.NOME, TABELADEP.DEPTO
FROM EMPREGADOS JOIN TABELADEP
    ON EMPREGADOS.COD_DEPTO = TABELADEP.COD_DEPTO;
```

```
SELECT EMPREGADOS.CODFUN, EMPREGADOS.NOME, TABELADEP.DEPTO
FROM EMPREGADOS, TABELADEP
WHERE EMPREGADOS.COD_DEPTO = TABELADEP.COD_DEPTO;
```

A respeito do **INNER JOIN**, é importante considerar as seguintes informações:

- A principal característica do **INNER JOIN** é que somente trará registros que encontrem correspondência nas duas tabelas, ou seja, se existir um empregado com **COD\_DEPTO** igual a 99 e na tabela de departamentos não existir um **COD\_DEPTO** de mesmo número, esse empregado não aparecerá no resultado final;
- O **SELECT** apontará um erro de sintaxe se existirem campos de mesmo nome nas duas tabelas e não indicarmos de qual tabela vem cada campo.

Neste treinamento, faremos o **JOIN** sempre na cláusula **FROM** usando a palavra **JOIN** e o critério com **ON**. Usar a cláusula **WHERE** para fazer o **JOIN** pode tornar o comando confuso e mais vulnerável a erros de resultado.

Veja os seguintes exemplos:

- **Exemplo 1**

```
SELECT CODFUN, NOME, DEPTO  
FROM EMPREGADOS JOIN TABELADEP  
    ON EMPREGADOS.COD_DEPTO = TABELADEP.COD_DEPTO;
```

Este exemplo está correto, porque não há duplicidade nos campos **CODFUN**, **NOME** e **DEPTO**.

- Exemplo 2

```
SELECT CODFUN, NOME, DEPTO  
FROM EMPREGADOS JOIN TABELADEP  
    ON COD_DEPTO = COD_DEPTO;
```

Este exemplo está errado, porque o campo **COD\_DEPTO** existe nas duas tabelas. É obrigatório indicar de qual tabela vamos pegar os campos.

 Sempre use o nome da tabela antes do nome do campo, mesmo que ele exista em apenas uma das tabelas.

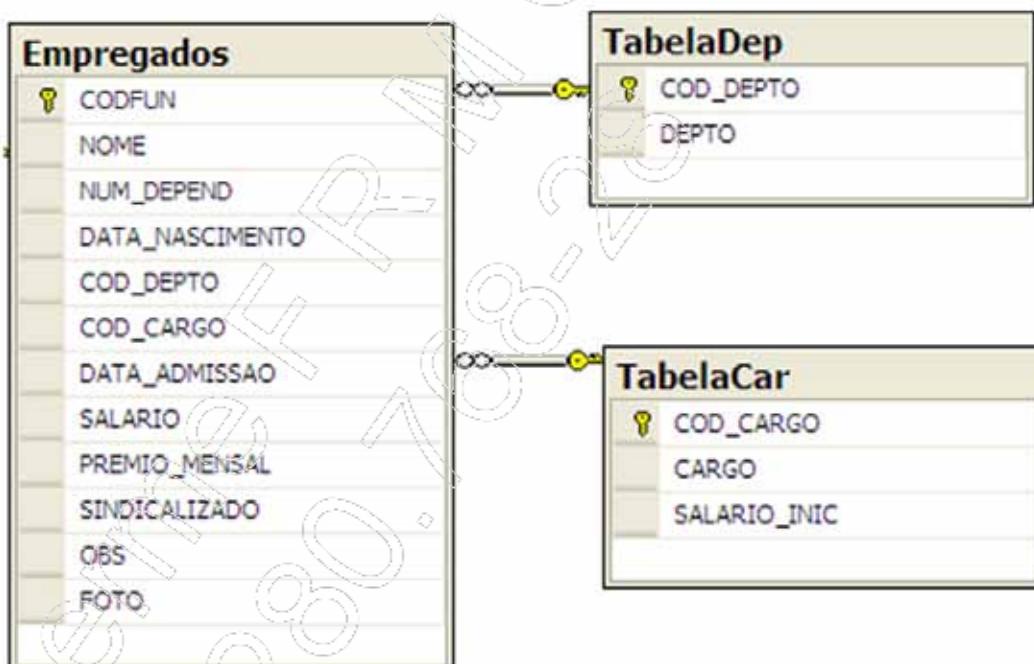
Quando tivermos nomes de tabelas muito extensos, podemos simplificar a escrita, dando apelidos às tabelas. Veja os exemplos:

```
SELECT E.CODFUN, E.NOME, D.DEPTO  
FROM EMPREGADOS AS E JOIN TABELADEP AS D  
    ON E.COD_DEPTO = D.COD_DEPTO;  
  
-- OU  
SELECT E.CODFUN, E.NOME, D.DEPTO  
FROM EMPREGADOS E JOIN TABELADEP D  
    ON E.COD_DEPTO = D.COD_DEPTO;
```

O que acabamos de demonstrar é um **JOIN**, ou associação. Quando relacionamos duas tabelas, é preciso informar qual campo permite essa ligação. Normalmente, o relacionamento se dá entre a chave estrangeira de uma tabela e a chave primária da outra, mas isso não é uma regra.

Na figura a seguir, você pode notar um ícone de chave na posição horizontal localizado na linha que liga as tabelas **EMPREGADOS** e **TABELADEP**. Essa chave está ao lado da tabela **TABELADEP**, o que indica que é a chave primária dessa tabela (**COD\_DEPTO**), e se relaciona com a tabela **EMPREGADOS**, que também possui um campo **COD\_DEPTO**, que é a chave estrangeira.

Há, também, um **JOIN** entre as tabelas **EMPREGADOS** e **TABELACAR**. Podemos perceber a existência de uma chave horizontalmente posicionada na linha que liga essas tabelas, e ela está ao lado de **TABELACAR**. Então, é a chave primária de **TABELACAR** (**COD\_CARGO**) que se relaciona com a tabela **EMPREGADOS**. Em **EMPREGADOS**, também temos um campo **COD\_CARGO**.



Normalmente, os campos que relacionam duas tabelas possuem o mesmo nome nas duas tabelas. Porém, isso não é uma condição necessária para que o **JOIN** funcione.

Quando executamos um **SELECT**, a primeira cláusula lida é **FROM**, antes de qualquer coisa. Depois é que as outras cláusulas são processadas. No código a seguir, temos dois erros: **E.CODIGO\_DEPTO** (que não existe) e **FROM EMPREGADS** (erro de digitação):

```
SELECT
    E.CODFUN, E.NOME, E.CODIGO_DEPTO, E.COD_CARGO, D.DEPTO
FROM EMPREGADS E
JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO;
```

Executado o código anterior, a seguinte mensagem de erro será exibida:

```
Mensagem 208, Nível 16, Estado 1, Linha 1
Invalid object name 'EMPREGADS'
```

O primeiro erro acusado na execução do código foi na cláusula **FROM**, o que prova que ela é processada antes.

A seguir, temos outro exemplo de **JOIN**:

```
-- EMPREGADOS e TABELACAR (cargos)
SELECT E.CODFUN, E.NOME, C.CARGO
FROM EMPREGADOS E JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;
-- OU
SELECT E.CODFUN, E.NOME, C.CARGO
FROM TABELACAR C JOIN EMPREGADOS E ON E.COD_CARGO = C.COD_CARGO;
```

# SQL 2014 - Módulo I

No próximo exemplo, temos o uso de **JOIN** para consultar três tabelas:

```
-- Consultar 3 tabelas
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO
FROM EMPREGADOS E
    JOIN TABELAdep D ON E.COD_DEPTO = D.COD_DEPTO
    JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;
-- OU
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO
FROM TABELAdep D
    JOIN EMPREGADOS E ON E.COD_DEPTO = D.COD_DEPTO
    JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;
-- OU
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO
FROM TABELACAR C
    JOIN EMPREGADOS E ON E.COD_CARGO = C.COD_CARGO
    JOIN TABELAdep D ON E.COD_DEPTO = D.COD_DEPTO;
```

Na consulta a seguir, temos dois erros. O primeiro é que não há **JOIN** entre **TABELAdep** e **TABELACAR**, como é mostrado no diagrama de tabelas do SSMS. O segundo é que não podemos fazer referência a uma tabela (**E.COD\_CARGO**), antes de abri-la:

```
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO
FROM TABELACAR C
    JOIN TABELAdep D ON E.COD_DEPTO = D.COD_DEPTO      --<< (1)
    JOIN EMPREGADOS E ON E.COD_CARGO = C.COD_CARGO;
```

A seguir, temos mais um exemplo da utilização de **JOIN**, desta vez com seis tabelas:

```
/*
Join com 6 tabelas. Vai exibir:
ITENSPEDIDO.NUM_PEDIDO
ITENSPEDIDO.NUM_ITEM
ITENSPEDIDO.COD_PRODUTO
PRODUTOS.DESCRICAO
ITENSPEDIDO.QUANTIDADE
ITENSPEDIDO.PR_UNITARIO
TIPOPTODUTO.TIPO
UNIDADES.UNIDADE
TABCOR.COR
PEDIDOS.DATA_EMISSAO

Filtrando pedidos emitidos em Janeiro de 2007
*/
SELECT
    I.NUM_PEDIDO, I.NUM_ITEM, I.COD_PRODUTO, PR.DESCRICAO,
    I.QUANTIDADE, I.PR_UNITARIO, T.TIPO, U.UNIDADE, CR.COR,
    PE.DATA_EMISSAO
FROM ITENSPEDIDO I
    JOIN PRODUTOS PR    ON I.ID_PRODUTO      = PR.ID_PRODUTO
    JOIN TABCOR CR      ON I.CODCOR        = CR.CODCOR
    JOIN TIPOPTODUTO T  ON PR.COD_TIPO     = T.COD_TIPO
    JOIN UNIDADES U     ON PR.COD_UNIDADE = U.COD_UNIDADE
    JOIN PEDIDOS PE    ON I.NUM_PEDIDO    = PE.NUM_PEDIDO
WHERE PE.DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31';
```

É possível, também, associar valores em duas colunas não idênticas. Nessa operação, utilizamos os mesmos operadores e predicados utilizados em qualquer **INNER JOIN**. A associação de colunas só é funcional quando associamos uma tabela a ela mesma, o que é conhecido como autoassociação ou self-join.

Em uma autoassociação, utilizamos a mesma tabela duas vezes na consulta, porém, especificamos cada instância da tabela por meio de aliases, que são utilizados para especificar os nomes das colunas durante a consulta.

Observe que, na tabela **EMPREGADOS**, temos o campo **CODFUN** (código do funcionário) e temos também o campo **COD\_SUPERVISOR**, que corresponde ao código do funcionário que é supervisor de cada empregado. Portanto, se quisermos consultar o nome do funcionário e do seu supervisor, precisaremos fazer um **JOIN**, da seguinte forma:

```
SELECT E.CODFUN, E.NOME AS FUNCIONARIO, S.NOME AS SUPERVISOR  
FROM EMPREGADOS E JOIN EMPREGADOS S  
ON E.COD_SUPERVISOR = S.CODFUN;
```

## 6.3. OUTER JOIN

A cláusula **INNER JOIN**, vista anteriormente, tem como característica retornar apenas as linhas em que o campo de relacionamento existe em ambas as tabelas. Se o conteúdo do campo chave de relacionamento existe em uma tabela, mas não na outra, essa linha não será retornada pelo **SELECT**. Vejamos um exemplo de **INNER JOIN**:

```
-- INNER JOIN  
SELECT * FROM EMPREGADOS; -- retorna 61 linhas  
  
--  
SELECT -- retorna 58 linhas  
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO  
FROM EMPREGADOS E  
    INNER JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;  
-- OU (a palavra INNER é opcional)  
SELECT -- retorna 58 linhas  
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO  
FROM EMPREGADOS E  
    JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;  
  
/*  
    Existem 61 linhas na tabela EMPREGADOS, mas quando fazemos  
    INNER JOIN com TABELACAR, retorna apenas com 58 linhas.  
    A explicação para isso é que existem 3 linhas em EMPREGADOS  
    com COD_CARGO inválido, inexistente em TABELADEP.  
*/
```

Uma cláusula **OUTER JOIN** retorna todas as linhas de uma das tabelas presentes em uma cláusula **FROM**. Dependendo da tabela (ou tabelas) cujos dados são retornados, podemos definir alguns tipos de **OUTER JOIN**, como veremos a seguir.

- **LEFT JOIN**

A cláusula **LEFT JOIN** ou **LEFT OUTER JOIN** permite obter não apenas os dados relacionados de duas tabelas, mas também os dados não relacionados encontrados na tabela à esquerda da cláusula **JOIN**. Ou seja, a tabela à esquerda sempre terá todos os seus dados retornados em uma cláusula **LEFT JOIN**. Caso não existam dados relacionados entre as tabelas à esquerda e à direita de **JOIN**, os valores resultantes de todas as colunas de lista de seleção da tabela à direita serão nulos.

Veja exemplos da utilização de **LEFT JOIN**:

```
/*
    OUTER JOIN: Exibe também as linhas que não tenham correspondência.
    No exemplo a seguir, mostramos TODAS as linhas da tabela
    que está à esquerda da palavra JOIN (EMPREGADOS)
*/
-- 
SELECT -- retorna 61 linhas
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM EMPREGADOS E
    LEFT OUTER JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;

-- OU (a palavra OUTER é opcional)
SELECT -- retorna 61 linhas
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM EMPREGADOS E
    LEFT JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO;
-- Observe o resultado e veja que existem 3 empregados
-- com CARGO = NULL porque o campo COD_CARGO não foi preenchido
```

O **SELECT** a seguir verifica, na tabela **EMPREGADOS**, os empregados que não possuem um código de departamento (**COD\_DEPTO**) válido:

```
-- Filtrar somente os registros não correspondentes
SELECT -- retorna 3 linhas
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM EMPREGADOS E
    LEFT JOIN TABELACAR C ON E.COD_CARGO = C.COD_CARGO
WHERE C.COD_CARGO IS NULL;
```

- **RIGHT JOIN**

Ao contrário da **LEFT OUTER JOIN**, a cláusula **RIGHT JOIN** ou **RIGHT OUTER 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.

Veja o seguinte uso de **RIGHT JOIN**. Da mesma forma que existem empregados que não possuem um **COD\_DEPTO** válido, podemos verificar se existe algum departamento sem nenhum empregado cadastrado. Nesse caso, deveremos exibir todos os registros da tabela que está à direita (**RIGHT**) da palavra **JOIN**, ou seja, da tabela **TABELADEP**:

```
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO
FROM EMPREGADOS E RIGHT JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO;
-- O resultado terá 2 departamentos que
-- não retornaram dados de empregados.

-- Filtrar somente os registros não correspondentes
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO
FROM EMPREGADOS E RIGHT JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO
WHERE E.COD_DEPTO IS NULL;
```

- **FULL JOIN**

Todas as linhas da tabela à esquerda de **JOIN** e da tabela à direita serão retornadas pela cláusula **FULL JOIN** ou **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.

A seguir, é exemplificada a utilização de **FULL JOIN**:

```
-- FULL JOIN une o LEFT e o RIGHT JOIN
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM EMPREGADOS E FULL JOIN TABELACAR C ON E.COD_CARGO = C.COD_
CARGO;
-- Observe o resultado e veja que existem 2 departamentos que
-- não retornaram dados de empregados.

-- Filtrar somente os registros não correspondentes
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM EMPREGADOS E FULL JOIN TABELACAR C ON E.COD_CARGO = C.COD_
CARGO
WHERE E.COD_CARGO IS NULL OR C.COD_CARGO IS NULL;
```

## 6.4. CROSS JOIN

Todos os dados da tabela à esquerda de **JOIN** são cruzados com os dados da tabela à direita de **JOIN**, ao utilizarmos **CROSS JOIN**. As possíveis combinações de linhas em todas as tabelas são conhecidas como produto cartesiano. O tamanho do produto cartesiano será definido pelo número de linhas na primeira tabela multiplicado pelo número de linhas na segunda tabela. É possível cruzar informações de duas ou mais tabelas.

Quando **CROSS JOIN** não possui uma cláusula **WHERE**, gera um produto cartesiano das tabelas envolvidas. Se adicionarmos uma cláusula **WHERE**, **CROSS JOIN** se comportará como uma **INNER JOIN**.

A seguir, temos um exemplo da utilização de **CROSS JOIN**:

```
-- CROSS JOIN  
SELECT -- retorna 854 linhas  
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO  
FROM EMPREGADOS E CROSS JOIN TABELADEP D;
```

**I**A **CROSS JOIN** deve ser utilizada apenas quando for realmente necessário um produto cartesiano, já que o resultado gerado pode ser muito grande.

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A associação de tabelas pode ser realizada, por exemplo, para converter em informação os dados encontrados em duas ou mais tabelas. As tabelas podem ser combinadas por meio de uma condição ou um grupo de condições de junção;
- 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;
- **JOIN** é uma cláusula que permite a associação entre várias tabelas, com base na relação existente entre elas. Por meio dessa cláusula, os dados de uma tabela são utilizados para selecionar dados pertencentes à outra tabela;
- Há diversos tipos de **JOIN**: **INNER JOIN**, **OUTER JOIN (LEFT JOIN, RIGHT JOIN e FULL JOIN)** e **CROSS JOIN**.



6

# Associando tabelas

## Teste seus conhecimentos

Guilherme Ribeiro  
392.280.168-28  
Caires



**IMPACTA**  
EDITORA

## 1. Qual cláusula não é um tipo de JOIN?

- a) INNER JOIN
- b) OUTER JOIN
- c) CROSS JOIN
- d) FULL JOIN
- e) ALTER JOIN

## 2. Analise o comando e verifique qual afirmação é a correta:

```
SELECT E.CODFUN, E.NOME, C.CARGO  
FROM EMPREGADOS E JOIN TABELACAR C ON E.COD_CARGO =  
E.COD_CARGO;
```

- a) O comando gera um erro.
- b) Apresenta uma lista com o código, nome e cargo do funcionário.
- c) Não apresenta nenhuma informação.
- d) Realiza um SELF-JOIN (relação com a mesma tabela) da tabela Empregado.
- e) Não é possível realizar esse JOIN.

### 3. Analise o comando e verifique qual afirmação é a correta:

```
SELECT
    I.NUM_PEDIDO, I.NUM_ITEM, I.COD_PRODUTO, PR.DESCRICAO,
    I.QUANTIDADE, I.PR_UNITARIO, T.TIPO, U.UNIDADE,
    CR.COR,
    PE.DATA_EMISSAO
FROM ITENSPEDIDO I
    JOIN PRODUTOS PR    ON I.ID_PRODUTO      = PR.ID_PRODUTO
    JOIN TABCOR CR       ON I.CODCOR        = CR.CODCOR
    JOIN TIOPRODUTO T   ON PR.COD_TIPO      = T.COD_TIPO
    JOIN UNIDADES U     ON PR.COD_UNIDADE   = U.COD_UNIDADE
    JOIN PEDIDOS PE     ON I.NUM_PEDIDO     = PE.NUM_PEDIDO
WHERE PE.DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31';
```

- a) A sintaxe está errada.
- b) Existem muitas tabelas e o SQL não consegue resolver a consulta.
- c) O comando executa um JOIN com várias tabelas e retorna as informações.
- d) Não é necessário utilizar a tabela PEDIDOS, pois não é utilizado nenhum campo dela.
- e) O comando executa um JOIN com várias tabelas, porém não retorna as informações.

4. A cláusula WHERE permite filtrar o resultado do comando SELECT. Qual comando a seguir seria uma instrução válida para apresentar um cliente de código 10 (Campo CODCLI do tipo INT)?

- a) FROM TABELA\_A LEFT OUTER JOIN TABELA\_B ON ....
- b) FROM TABELA\_A INNER JOIN TABELA\_B ON ....
- c) FROM TABELA\_A JOIN TABELA\_B ON ....
- d) FROM TABELA\_A RIGHT JOIN TABELA\_B ON ....
- e) FROM TABELA\_A FULL JOIN TABELA\_B ON ....

5. Um JOIN do tipo CROSS realiza:

- a) Relação entre duas tabelas mostrando todos os dados das duas tabelas.
- b) Um produto cartesiano, que é a combinação de todos os registros de uma tabela com todos da outra.
- c) É um recurso disponível somente da versão SQL 2008 em diante.
- d) Devemos utilizar sempre e filtrarmos a consulta com WHERE.
- e) Não existe este tipo de JOIN.

6

# Associando tabelas

Mãos à obra!

Guilherme FR  
392.280.668-28  
Caires

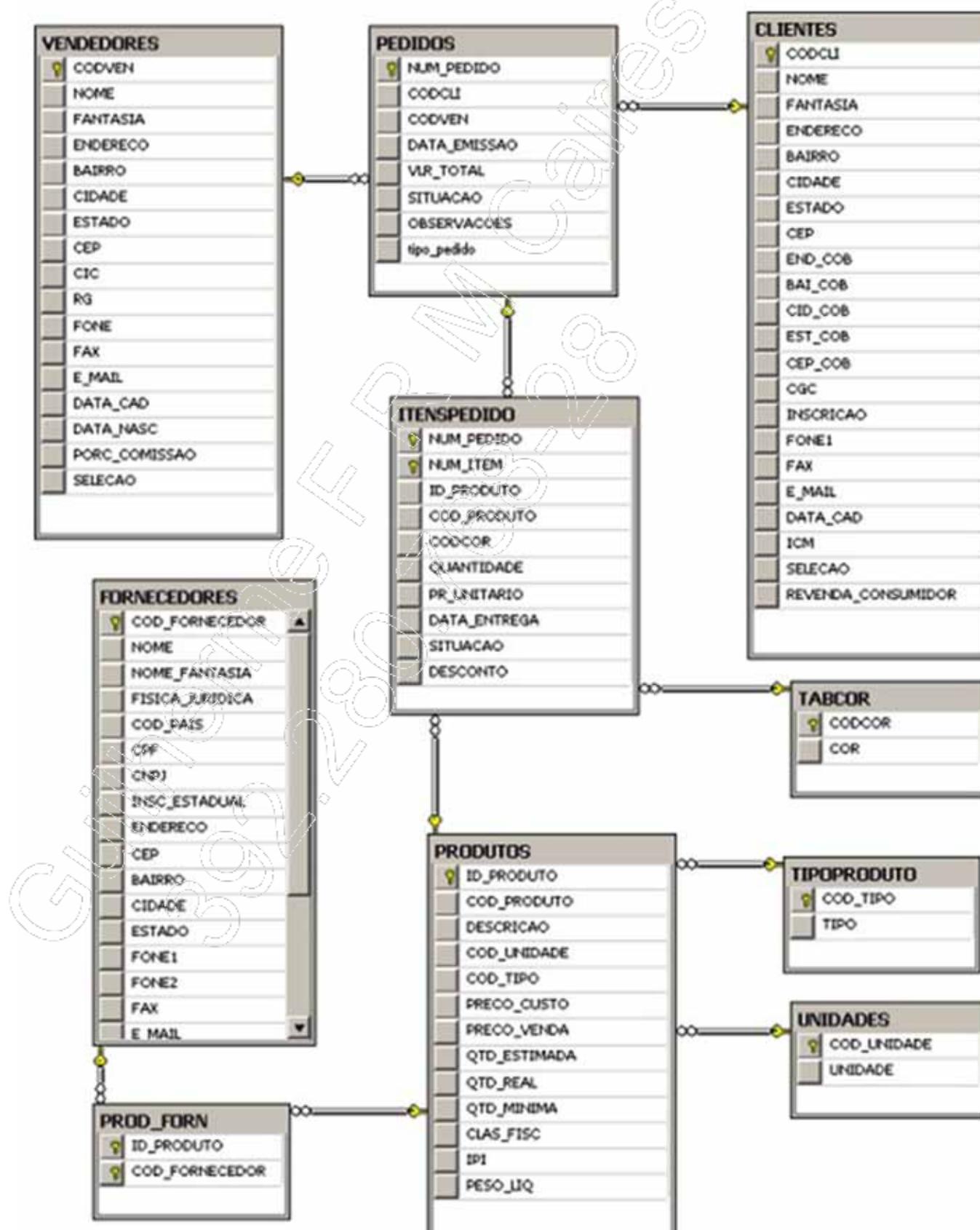


**IMPACTA**  
EDITORA

## Laboratório 1

### A – Utilizando o comando JOIN para associar tabelas

Considere o seguinte diagrama relacional de banco de dados para associar tabelas:



1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste os campos **NUM\_PEDIDO**, **DATA\_EMISSAO** e **VLR\_TOTAL** de **PEDIDOS**, seguidos de **NOME** do vendedor;
3. Liste os campos **NUM\_PEDIDO**, **DATA\_EMISSAO** e **VLR\_TOTAL** de **PEDIDOS**, seguidos de **NOME** do cliente;
4. Liste os pedidos com o nome do vendedor e o nome do cliente;
5. Liste os itens de pedido (**ITENSPEEDIDO**) com o nome do produto (**PRODUTOS.DESCRIAO**);
6. Liste os campos **COD\_PRODUTO** e **DESCRICAQ** da tabela **PRODUTOS**, seguidos da descrição do tipo de produto (**TIPOPRODUTO.TIPO**);
7. Liste os campos **COD\_PRODUTO** e **DESCRICAQ** da tabela **PRODUTOS**, seguidos da descrição do tipo de produto (**TIPOPRODUTO.TIPO**) e do nome da unidade de medida (**UNIDADES.UNIDADE**);
8. Liste os campos **NUM\_PEDIDO**, **NUM\_ITEM**, **COD\_PRODUTO**, **QUANTIDADE** e **PR\_UNITARIO** da tabela **ITENSPEEDIDO**, e os campos **COD\_PRODUTO** e **DESCRICAQ** da tabela **PRODUTOS**, seguidos da descrição do tipo de produto (**TIPOPRODUTO.TIPO**) e do nome da unidade de medida (**UNIDADES.UNIDADE**);
9. Liste os campos **NUM\_PEDIDO**, **NUM\_ITEM**, **COD\_PRODUTO**, **QUANTIDADE** e **PR\_UNITARIO** da tabela **ITENSPEEDIDO**, e os campos **COD\_PRODUTO** e **DESCRICAQ** da tabela **PRODUTOS**, seguidos da descrição do tipo de produto (**TIPOPRODUTO.TIPO**), do nome da unidade de medida (**UNIDADES.UNIDADE**) e do nome da cor (**TABCOR.COR**);
10. Liste todos os pedidos (**PEDIDOS**) do vendedor **MARCELO** em Jan/2007;



Este exercício não especifica quais campos devem ser exibidos.  
Escolha você os campos que devem ser mostrados.

## SQL 2014 - Módulo I

---

11. Liste os nomes dos clientes (**CLIENTES.NOME**) que efetuaram compras em janeiro de 2007;
12. Liste os nomes de produtos (**PRODUTOS.DESCRICAO**) que foram vendidos em janeiro de 2007;
13. Liste **NUM\_PEDIDO**, **VLR\_TOTAL** (**PEDIDOS**) e **NOME** (**CLIENTE**). Mostre apenas pedidos de janeiro de 2007 e clientes que tenham **NOME** iniciado com **MARCIO**;
14. Liste **NUM\_PEDIDO**, **QUANTIDADE** vendida e **PR\_UNITARIO** (de **ITENSPEDIDO**), **DESCRICAO** (de **PRODUTOS**), **NOME** do vendedor que vendeu cada item de pedido (de **VENDEDORES**);
15. Liste todos os itens de pedido com desconto superior a 7%. Mostre **NUM\_PEDIDO**, **DESCRICAO** do produto, **NOME** do cliente, **NOME** do vendedor e **QUANTIDADE** vendida;
16. Liste os itens de pedido com o nome do produto, a descrição do tipo, a descrição da unidade e o nome da cor, mas apenas os itens vendidos em janeiro de 2007 na cor **LARANJA**;
17. Liste **NOME** e **FONE1** dos fornecedores que venderam o produto **CANETA STAR I**;
18. Liste a **DESCRICAO** dos **PRODUTOS** comprados do fornecedor cujo **NOME** começa com **LINCE**;
19. Liste **NOME** e **FONE1** dos fornecedores, bem como **DESCRICAO** dos produtos com **QTD\_REAL** abaixo de **QTD\_MINIMA**;
20. Liste todos os **PRODUTOS** comprados do fornecedor cujo nome inicia-se com **FESTO**.

# Consultas com subqueries

7

- ✓ Principais características das subqueries;
- ✓ Subqueries introduzidas com IN e NOT IN;
- ✓ Subqueries introduzidas com sinal de igualdade (=);
- ✓ Subqueries correlacionadas;
- ✓ Diferenças entre subqueries e associações;
- ✓ Diferenças entre subqueries e tabelas temporárias.



**IMPACTA**  
EDITORA

### 7.1. Introdução

Uma consulta aninhada em uma instrução **SELECT, INSERT, DELETE ou UPDATE** é chamada de subquery (subconsulta). Isso ocorre quando usamos um **SELECT** dentro de um **SELECT, INSERT, UPDATE ou DELETE**. Também é comum chamar uma subquery de query interna. Já a instrução em que está inserida a subquery pode ser chamada de query externa.

O limite máximo de aninhamento de uma subquery é de 32 níveis, limite este que varia de acordo com a complexidade das outras instruções que compõem a consulta e com a quantidade de memória disponível.

### 7.2. Principais características das subqueries

A seguir, temos a descrição das principais características das subqueries:

- As subqueries podem ser escalares (retornam apenas uma linha) ou tabulares (retornam linhas e colunas);
- É possível obter apenas uma coluna por subquery;
- Uma subquery, que pode ser incluída dentro de outra subquery, deve estar entre parênteses, o que a diferenciará da consulta principal;
- Em instruções **SELECT, UPDATE, INSERT e DELETE**, uma subquery é utilizada nos mesmos locais em que poderiam ser utilizadas expressões;
- Pelo fato de podermos trabalhar com consultas estruturadas, as subqueries permitem que partes de um comando sejam separadas das demais partes;
- Se uma instrução é permitida em um local, este local aceita a utilização de uma subquery;

- Alguns tipos de dados não podem ser utilizados na lista de seleção de uma subquery. São eles: **nvarchar(max)**, **varchar(max)** e **varbinary(max)**;
- Um único valor será retornado ao utilizarmos o sinal de igualdade (=) no início da subquery;
- As palavras-chave **ALL**, **ANY** e **SOME** podem ser utilizadas para modificar operadores de comparação que introduzem uma subquery. Podemos fazer as seguintes considerações a respeito delas:
  - **ALL** realiza uma comparação entre um valor escalar e um conjunto de valores de uma coluna. Então, retornará **TRUE** nos casos em que a comparação for verdadeira para todos os pares;
  - **SOME** (padrão ISO que equivale a **ANY**) e **ANY** realizam uma comparação entre um valor escalar e um conjunto de valores de uma coluna. Então, retornarão **TRUE** nos casos em que a comparação for verdadeira para qualquer um dos pares.
- Quando utilizamos um operador de comparação (=, < >, >, > =, <, ! >, ! < ou < =) para introduzir uma subquery, sua lista de seleção poderá incluir apenas um nome de coluna ou expressão, a não ser que utilizemos **IN** na lista ou **EXISTS** no **SELECT**;
- As cláusulas **GROUP BY** e **HAVING** não podem ser utilizadas em subqueries introduzidas por um operador de comparação que não seja seguido pelas palavras-chave **ANY** ou **ALL**;
- A utilização da cláusula **ORDER BY** só é possível caso a cláusula **TOP** seja especificada;
- Alternativamente, é possível formular muitas instruções do Transact-SQL com subqueries como associações;

- O nome de coluna que, ocasionalmente, estiver presente na cláusula **WHERE** de uma query externa deve ser associável com a coluna da lista de seleção da subquery;
- A qualificação dos nomes de colunas de uma instrução é feita pela tabela referenciada na cláusula **FROM**;
- Subqueries que incluem **GROUP BY** não aceitam a utilização de **DISTINCT**;
- Uma view não pode ser atualizada caso ela tenha sido criada com uma subquery;
- Uma subquery aninhada na instrução **SELECT** externa é formada por uma cláusula **FROM** regular com um ou mais nomes de view ou tabela, por uma consulta **SELECT** regular junto dos componentes da lista de seleção regular e pelas cláusulas opcionais **WHERE**, **HAVING** e **GROUP BY**;
- Subqueries podem ser utilizadas para realizar testes de existência de linhas. Nesse caso, é adotado o operador **EXISTS**;
- Por oferecer diversas formas de obter resultados, as subqueries eliminam a necessidade de utilização das cláusulas **JOIN** e **UNION** de maior complexidade;
- Uma instrução que possui uma subquery não apresenta muitas diferenças de performance em relação a uma versão semanticamente semelhante que não possui a subquery. No entanto, uma **JOIN** apresenta melhor desempenho nas situações em que é necessário realizar testes de existência;
- As colunas de uma tabela não poderão ser incluídas na saída, ou seja, na lista de seleção da query externa, caso essa tabela apareça apenas em uma subquery e não na query externa.

A seguir, temos os formatos normalmente apresentados pelas instruções que possuem uma subquery:

```
WHERE expressao [NOT] IN (subquery);  
WHERE expressao operador_comparacao [ANY | ALL] (subquery);  
WHERE [NOT] EXISTS (subquery).
```

Veja um exemplo de subquery que verifica a existência de clientes que compraram no mês de janeiro de 2007:

```
SELECT * FROM CLIENTES  
WHERE EXISTS (SELECT * FROM PEDIDOS  
    WHERE CODCLI = CLIENTES.CODCLI AND  
        DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31');  
-- OU  
SELECT * FROM CLIENTES  
WHERE CODCLI IN (SELECT CODCLI FROM PEDIDOS  
    WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31');
```

No próximo exemplo, é verificada a existência de clientes que não compraram em janeiro de 2007:

```
SELECT * FROM CLIENTES  
WHERE NOT EXISTS (SELECT * FROM PEDIDOS  
    WHERE CODCLI = CLIENTES.CODCLI AND  
        DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31');  
-- OU  
SELECT * FROM CLIENTES  
WHERE CODCLI NOT IN (SELECT CODCLI FROM PEDIDOS  
    WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31');
```

## 7.3. Subqueries introduzidas com IN e NOT IN

Uma subquery terá como resultado uma lista de zero ou mais valores caso tenha sido introduzida com a utilização de **IN** ou **NOT IN**. O resultado, então, será utilizado pela query externa.

Os exemplos adiante demonstram subqueries introduzidas com **IN** e **NOT IN**:

- **Exemplo 1**

```
-- Lista de empregados cujo cargo tenha salário inicial
-- inferior a 5000
SELECT * FROM EMPREGADOS
WHERE COD_CARGO IN (SELECT COD_CARGO FROM TABELACAR
                      WHERE SALARIO_INIC < 5000);
```

- **Exemplo 2**

```
-- Lista de departamentos em que não existe nenhum
-- funcionário cadastrado
SELECT * FROM TABELADEP
WHERE COD_DEPTO NOT IN
      (SELECT DISTINCT COD_DEPTO FROM EMPREGADOS
       WHERE COD_DEPTO IS NOT NULL)
-- O mesmo que
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.COD_DEPTO,
    D.DEPTO
  FROM EMPREGADOS E RIGHT JOIN TABELADEP D ON E.COD_DEPTO = D.COD_
DEPTO
  WHERE E.COD_DEPTO IS NULL
```

- **Exemplo 3**

```
-- Lista de cargos em que não existe nenhum
-- funcionário cadastrado
SELECT * FROM TABELACAR
WHERE COD_CARGO NOT IN
      (SELECT DISTINCT COD_CARGO FROM EMPREGADOS
       WHERE COD_CARGO IS NOT NULL)
-- O mesmo que
SELECT
    C.COD_CARGO, C.CARGO
  FROM EMPREGADOS E RIGHT JOIN TABELACAR C ON E.COD_CARGO = C.COD_
CARGO
  WHERE E.COD_CARGO IS NULL
```

## 7.4. Subqueries introduzidas com sinal de igualdade (=)

Veja exemplos de como utilizar o sinal de igualdade (=) para inserir subqueries:

- **Exemplo 1**

```
-- Funcionário(s) que ganha(m) menos  
SELECT * FROM EMPREGADOS  
WHERE SALARIO = (SELECT MIN(SALARIO) FROM Empregados)  
-- o mesmo que  
SELECT TOP 1 WITH TIES * FROM EMPREGADOS  
WHERE SALARIO IS NOT NULL  
ORDER BY SALARIO
```

- **Exemplo 2**

```
-- Funcionário mais novo na empresa  
SELECT * FROM EMPREGADOS  
WHERE DATA_ADMISSAO = (SELECT MAX(DATA_ADMISSAO) FROM EMPREGA-  
DOS);  
  
-- O mesmo que  
SELECT TOP 1 WITH TIES * FROM EMPREGADOS  
ORDER BY DATA_ADMISSAO DESC;
```

## 7.5. Subqueries correlacionadas

Quando uma subquery possui referência a uma ou mais colunas da query externa, ela é chamada de subquery correlacionada. É uma subquery repetitiva, pois é executada uma vez para cada linha da query externa. Assim, os valores das subqueries correlacionadas dependem da query externa, o que significa que, para construir uma subquery desse tipo, será necessário criar tanto a query interna como a externa.

Também é possível que subqueries correlacionadas incluam, na cláusula **FROM**, funções definidas pelo usuário, as quais retornam valores de tipo de dado **table**. Para isso, basta que colunas de uma tabela na query externa sejam referenciadas como argumento de uma função desse tipo. Então, será feita a avaliação dessa função de acordo com a subquery para cada linha da query externa. Veja o exemplo a seguir, que grava, no campo **SALARIO** de cada funcionário, o valor de salário inicial contido na tabela de cargos:

```
UPDATE EMPREGADOS SET SALARIO = (SELECT SALARIO_INIC  
FROM TABELACAR  
WHERE COD_CARGO = EMPREGADOS.COD_CARGO);
```

## 7.5.1. Subqueries correlacionadas com EXISTS

Subqueries correlacionadas introduzidas com a cláusula **EXISTS** não retornam dados, mas apenas **TRUE** ou **FALSE**. Sua função é executar um teste de existência de linhas, portanto, se houver qualquer linha em uma subquery, será retornado **TRUE**.

Sobre **EXISTS**, é importante atentarmos para os seguintes aspectos:

- Antes de **EXISTS** não deve haver nome de coluna, constante ou expressão;
- Quando **EXISTS** introduz uma subquery, sua lista de seleção será, normalmente, um asterisco.

Por meio do código a seguir, é possível saber se temos clientes que não realizaram compra no mês de janeiro de 2007:

```
SELECT * FROM CLIENTES  
WHERE NOT EXISTS (SELECT * FROM PEDIDOS  
WHERE CODCLI = CLIENTES.CODCLI AND  
DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31');
```

## 7.6. Diferenças entre subqueries e associações

Ao comparar subqueries e associações (**JOINS**), é possível constatar que as associações são mais indicadas para verificação de existência, pois apresentam desempenho melhor nesses casos. Também podemos verificar que, ao contrário das subqueries, as associações não atuam em listas com um operador de comparação modificado por **ANY** ou **ALL**, ou em listas que tenham sido introduzidas com **IN** ou **EXISTS**.

Em alguns casos, pode ser que lidemos com questões muito complexas para serem respondidas com associações, então, será mais indicado usar subqueries. Isso porque a visualização do aninhamento e da organização da query é mais simples em uma subquery, enquanto que, em uma consulta com diversas associações, a visualização pode ser complicada. Além disso, nem sempre as associações podem reproduzir os efeitos de uma subquery.

O código a seguir utiliza **JOIN** para calcular o total vendido por cada vendedor no período de janeiro de 2007 e a porcentagem de vendas em relação ao total de vendas realizadas no mesmo mês:

```
SELECT P.CODVEN, V.NOME,
       SUM(P.VLR_TOTAL) AS TOT_VENDIDO,
       100 * SUM(P.VLR_TOTAL) / (SELECT SUM(VLR_TOTAL)
                                FROM PEDIDOS
                                WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31') AS
                                PORCENTAGEM
   FROM PEDIDOS P JOIN VENDEDORES V ON P.CODVEN = V.CODVEN
  WHERE P.DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31'
 GROUP BY P.CODVEN, V.NOME;
```

Já o código a seguir utiliza subqueries para calcular, para cada departamento, o total de salários dos funcionários sindicalizados e o total de salários dos não sindicalizados:

```
SELECT COD_DEPTO,
       (SELECT SUM(E.SALARIO) FROM Empregados E
        WHERE E.SINDICALIZADO = 'S' AND
              E.COD_DEPTO = Empregados.COD_DEPTO) AS TOT_SALARIO_SIND,
       (SELECT SUM(E.SALARIO) FROM Empregados E
        WHERE E.SINDICALIZADO = 'N' AND
              E.COD_DEPTO = Empregados.COD_DEPTO) AS TOT_SALARIO_NAO_
        SIND
   FROM EMPREGADOS
  GROUP BY COD_DEPTO;
```

### 7.7. Diferenças entre subqueries e tabelas temporárias

Embora as tabelas temporárias sejam parecidas com as permanentes, elas são armazenadas em **tempdb** e excluídas automaticamente após terem sido utilizadas.

As tabelas temporárias locais apresentam, antes do nome, o símbolo # e são visíveis somente durante a conexão atual. Quando o usuário desconecta-se da instância do SQL Server, ela é excluída.

Já as tabelas temporárias globais apresentam, antes do nome, dois símbolos ## e são visíveis para todos os usuários. Uma tabela desse tipo será excluída apenas quando todos os usuários que a referenciam se desconectarem da instância do SQL Server.

A escolha da utilização de tabelas temporárias ou de subqueries dependerá de cada situação e de aspectos como desempenho do sistema e até mesmo das preferências pessoais de cada usuário. O fato é que, por conta das diferenças existentes entre elas, o uso de uma, para uma situação específica, acaba sendo mais indicado do que o emprego de outra.

Assim, quando temos bastante RAM, as subqueries são preferíveis, pois ocorrem na memória. Já as tabelas temporárias, como necessitam dos recursos disponibilizados pelo disco rígido para serem executadas, são indicadas nas situações em que o(s) servidor(es) do banco de dados apresenta(m) bastante espaço no disco rígido.

Há, ainda, uma importante diferença entre tabela temporária e subquery: normalmente, esta última é mais fácil de manter. No entanto, se a subquery for muito complexa, a melhor medida a ser tomada pode ser fragmentá-la em diversas tabelas temporárias, criando, assim, blocos de dados de tamanho menor.

Veja o exemplo a seguir, que utiliza subqueries para retornar os pedidos da vendedora **LEIA** para clientes de **SP** que não compraram em janeiro de 2007, mas compraram em dezembro de 2006:

```
SELECT * FROM PEDIDOS
WHERE CODVEN IN (SELECT CODVEN FROM VENDEDORES WHERE NOME =
'LEIA')
AND CODCLI IN (
    SELECT CODCLI FROM CLIENTES
    WHERE CODCLI NOT IN (SELECT CODCLI FROM PEDIDOS
        WHERE DATA_EMISSAO BETWEEN
        '2007.1.1' AND '2007.1.31')
    AND
        CODCLI IN (SELECT CODCLI FROM PEDIDOS
            WHERE DATA_EMISSAO BETWEEN
            '2006.12.1'
            AND '2006.12.31')
    AND ESTADO = 'SP' );
```

Já no próximo código, em vez de subqueries, utilizamos tabelas temporárias para obter o mesmo resultado do código anterior:

```
-- Tabela temporária 1 - Código da vendedora LEIA
SELECT CODVEN INTO #VEND_LEIA FROM VENDEDORES WHERE NOME =
'LEIA';

-- Tabela temporária 2 -- Clientes que compraram em Jan/2007
SELECT CODCLI INTO #CLI_COM_PED_JAN_2007 FROM PEDIDOS
WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31';

-- Tabela temporária 3 - Clientes que compraram em Dez/2006
SELECT CODCLI INTO #CLI_COM_PED_DEZ_2006 FROM PEDIDOS
WHERE DATA_EMISSAO BETWEEN '2006.12.1' AND '2006.12.31';

-- Tabela temporária 4 - Clientes de SP que compraram
-- em Dez/2006, mas não compraram em Jan de 2007
SELECT CODCLI INTO #CLI_FINAL FROM CLIENTES
WHERE CODCLI NOT IN (SELECT CODCLI FROM #CLI_COM_PED_JAN_2007)
    AND
        CODCLI IN (SELECT CODCLI FROM #CLI_COM_PED_DEZ_2006)
    AND
        ESTADO = 'SP';

-- SELECT de PEDIDOS
SELECT * FROM PEDIDOS
WHERE CODVEN IN (SELECT CODVEN FROM #VEND_LEIA)
    AND
        CODCLI IN (SELECT CODCLI FROM #CLI_FINAL);
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Uma consulta aninhada em uma instrução **SELECT**, **INSERT**, **DELETE** ou **UPDATE** é denominada subquery (subconsulta). As subqueries são também referidas como queries internas. Já a instrução em que está inserida a subquery pode ser chamada de query externa;
- Vejamos algumas das diversas características das subqueries: podem ser escalares (retornam apenas uma linha) ou tabulares (retornam linhas e colunas). Elas, que podem ser incluídas dentro de outras subqueries, devem estar entre parênteses, o que as diferenciará da consulta principal;
- Uma subquery retornará uma lista de zero ou mais valores caso tenha sido introduzida com a utilização de **IN** ou **NOT IN**. O resultado, então, será utilizado pela query externa;
- O sinal de igualdade (=) pode ser utilizado para inserir subqueries;
- Quando uma subquery possui referência a uma ou mais colunas da query externa, ela é chamada de subquery correlacionada. Trata-se de uma subquery repetitiva, pois é executada uma vez para cada linha da query externa. Desta forma, os valores das subqueries correlacionadas dependem da query externa, o que significa que, para construir uma subquery desse tipo, será necessário criar tanto a query interna como a externa;
- Ao comparar subqueries e associações (**JOINS**), é possível constatar que as associações são mais indicadas para verificação de existência, pois apresentam desempenho melhor nesses casos;

- A visualização do aninhamento e da organização da query é mais simples em uma subquery, enquanto que, em uma consulta com diversas associações, a visualização pode ser complicada;
- Tabelas temporárias são armazenadas no **tempdb** e excluídas automaticamente após terem sido utilizadas;
- As tabelas temporárias locais apresentam, antes do nome, o símbolo # e são visíveis somente durante a conexão atual. Quando o usuário desconecta-se da instância do SQL Server, ela é excluída. Já as tabelas temporárias globais apresentam, antes do nome, dois símbolos ## e são visíveis para todos os usuários. Uma tabela desse tipo será excluída apenas quando todos os usuários que a referenciam se desconectarem da instância do SQL Server;
- A escolha da utilização de tabelas temporárias ou subqueries dependerá de cada situação e de aspectos como desempenho do sistema e até mesmo das preferências pessoais de cada usuário.



# **Consultas com subqueries**

## **Teste seus conhecimentos**

**7**

Guilherme  
Caires  
392.280.128



**IMPACTA**  
EDITORA

## 1. Qual não é uma característica da SUBQUERY?

- a) A utilização da cláusula ORDER BY só é possível caso a cláusula TOP seja especificada OUTER JOIN.
- b) Subqueries podem ser utilizadas para realizar testes de existência de linhas. Nesse caso, é adotado o operador EXISTS.
- c) Subqueries são recursos que não devem ser utilizados.
- d) Em instruções SELECT, UPDATE, INSERT e DELETE, uma subquery é utilizada nos mesmos locais em que poderiam ser utilizadas expressões.
- e) Se uma instrução é permitida em um local, este local aceita a utilização de uma subquery.

**2. Analise o comando e verifique qual afirmação é a correta:**

```
SELECT * FROM CLIENTES  
WHERE CODCLI IN (SELECT CODCLI FROM PEDIDOS  
                   WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND  
                   '2007.1.31');
```

- a) Apresenta os clientes que compraram em Janeiro de 2007.
- b) Apresenta os clientes que não compraram em Janeiro de 2007.
- c) Não apresenta nenhuma informação.
- d) Comando errado, pois, no lugar do operador IN, deve ser utilizado o igual.
- e) Apresenta os pedidos dos clientes de Janeiro de 2007.

### 3. Analise o comando e verifique qual afirmação é a correta:

```
SELECT * FROM EMPREGADOS  
WHERE COD_CARGO = (SELECT COD_CARGO FROM TABELACAR  
                     WHERE SALARIO_INIC < 5000);
```

- a) A sintaxe está errada.
- b) O comando retornará todos os empregados que possuem cargo com salário inicial menor que 5000.
- c) O comando não retorna nenhum registro.
- d) Não existe diferença entre o operador IN e igual.
- e) Ocorrerá um erro quando a sub consulta retornar mais de um registro.

## 4. Qual afirmação está errada, com relação a subqueries correlacionadas?

- a) Antes de EXISTS não deve haver nome de coluna, constante ou expressão.
- b) Quando EXISTS introduz uma subquery, sua lista de seleção será, normalmente, um asterisco.
- c) Subqueries são complexas e não devem ser utilizadas.
- d) A cláusula EXISTS não retorna dados, mas apenas TRUE ou FALSE.
- e) Todas as alternativas anteriores estão corretas.

## 5. Qual afirmação está errada, com relação a Tabelas Temporárias?

- a) Quando declararmos com sinal #, a tabela é global.
- b) Quando declararmos com sinal #, a tabela é Local.
- c) Ao encerrarmos a conexão, a tabela temporária local é excluída automaticamente.
- d) Tabelas temporárias globais são recursos que devem ser evitados.
- e) Tabelas temporárias ficam armazenadas no tempdb.



# Consultas com subqueries

## Mãos à obra!

7

Guilherme  
Caires  
392.280.  
1000

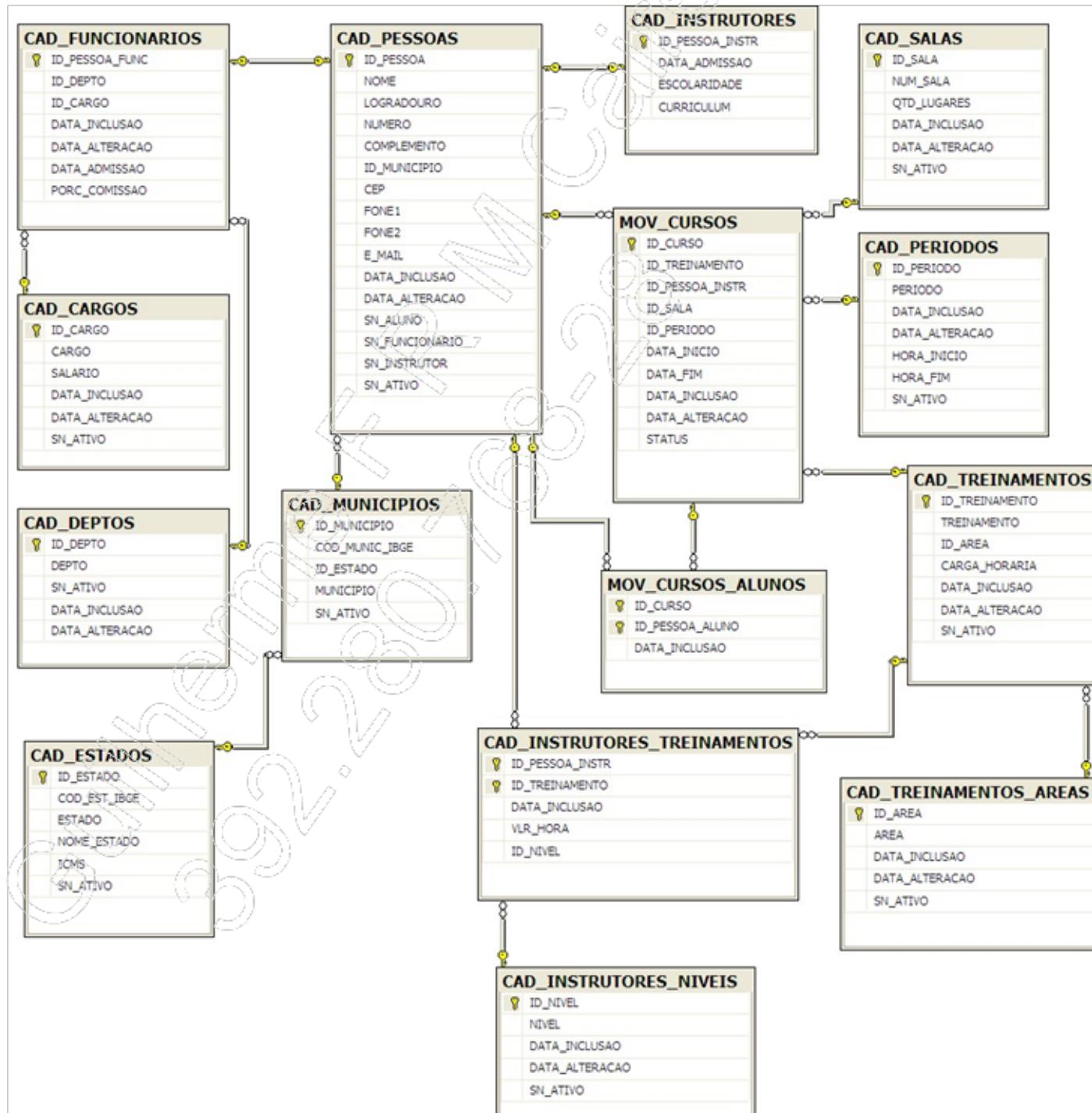


**IMPACTA**  
EDITORA

# Laboratório 1

## A – Trabalhando com JOIN e subquery

Para auxiliar na execução do laboratório, utilize o diagrama a seguir:



1. Execute o script: **Cap07\_Lab01\_CriaBanco.sql**;
2. Apresente todas as salas de aula para as quais não há nenhum curso marcado;
3. Apresente todos os treinamentos para os quais não há instrutor;
4. Apresente os alunos (**CAD\_PESSOAS**) que não têm e nem tiveram cursos agendados;
5. Apresente os departamentos que não possuem funcionários cadastrados;
6. Apresente os cargos para os quais não existem funcionários cadastrados;
7. Apresente as pessoas que sejam de estados cujo ICMS seja menor que 7;
8. Apresente os dados do instrutor que possui o maior valor hora (**VLR\_HORA**);
9. Apresente os dados do instrutor que possui o menor valor hora (**VLR\_HORA**).



# Atualizando e excluindo dados em associações e subqueries

8

- ✓ UPDATE com subqueries;
- ✓ DELETE com subqueries;
- ✓ UPDATE com JOIN;
- ✓ DELETE com JOIN.



**IMPACTA**  
EDITORA

### 8.1. UPDATE com subqueries

Ao utilizarmos a instrução **UPDATE** em uma subquery, será possível atualizar linhas de uma tabela com informações provenientes de outra tabela. Para isso, na cláusula **WHERE** da instrução **UPDATE**, em vez de usar como critério para a operação de atualização a origem explícita da tabela, basta utilizar uma subquery.

Veja os dois códigos a seguir. Ambos utilizam **UPDATE** com subquery. No primeiro código, o comando é para aumentar em 10% os salários dos empregados do departamento **CPD**. No segundo, o preço de venda é atualizado para que fique 20% acima do preço de custo de todos os produtos do tipo **REGUA**:

```
UPDATE EMPREGADOS  
SET SALARIO = SALARIO * 1.10  
WHERE COD_DEPTO = (SELECT COD_DEPTO FROM TABELADEP  
WHERE DEPTO = 'CPD');  
  
UPDATE PRODUTOS SET PRECO_VENDA = PRECO_CUSTO * 1.2  
WHERE COD_TIPO = (SELECT COD_TIPO FROM TIPOPRODUTO  
WHERE TIPO = 'REGUA');
```

### 8.2. DELETE com subqueries

Podemos utilizar subqueries para remover dados de uma tabela. Basta definir a cláusula **WHERE** da instrução **DELETE** como uma subquery. Isso irá excluir linhas de uma tabela base conforme os dados armazenados em outra tabela. Veja o próximo exemplo, que elimina os pedidos do vendedor **MARCELO** que foram emitidos na primeira quinzena de dezembro de 2006:

```
DELETE FROM PEDIDOS  
WHERE DATA_EMISSAO BETWEEN '2006.12.1' AND '2006.12.15' AND  
CODVEN = (SELECT CODVEN FROM VENDEDORES WHERE NOME = 'MAR-  
CELO');
```

## 8.3. UPDATE com JOIN

Podemos usar uma associação em tabelas para determinar quais colunas serão atualizadas por meio de **UPDATE**. No exemplo a seguir, aumentaremos em 10% os salários dos empregados do departamento **CPD**:

```
UPDATE Empregados  
SET SALARIO *= 1.10  
FROM EMPREGADOS E JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO  
WHERE D.DEPTO = 'CPD';
```

Já o próximo código atualiza o preço de venda para 20% acima do preço de custo de todos os produtos do tipo **REGUA**:

```
UPDATE PRODUTOS SET PRECO_VENDA = PRECO_CUSTO * 1.2  
FROM PRODUTOS P JOIN TIPOPRODUTO T ON P.COD_TIPO = T.COD_TIPO  
WHERE T.TIPO = 'REGUA';
```

## 8.4. DELETE com JOIN

Os dados provenientes de tabelas associadas podem ser eliminados por meio da cláusula **JOIN** junto ao comando **DELETE**. Veja o seguinte exemplo, que exclui os pedidos do vendedor **MARCELO** emitidos na primeira quinzena de dezembro de 2006:

```
DELETE FROM PEDIDOS  
FROM PEDIDOS P JOIN VENDEDORES V ON P.CODVEN = V.CODVEN  
WHERE P.DATA_EMISSAO BETWEEN '2006.12.1' AND '2006.12.15' AND  
V.NOME = 'MARCELO';
```

## Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- O uso da instrução **UPDATE** em uma subquery permite atualizar linhas de uma tabela com informações provenientes de outra tabela;
- Subqueries podem ser utilizadas com o intuito de remover dados de uma tabela. Basta definirmos a cláusula **WHERE** da instrução **DELETE** como uma subquery para excluir linhas de uma tabela base conforme os dados armazenados em uma outra tabela;
- Podemos utilizar uma associação (**JOIN**) em tabelas para determinar quais colunas serão atualizadas por meio de **UPDATE**;
- Os dados de tabelas associadas podem ser eliminados por meio da cláusula **JOIN** com o comando **DELETE**.

# Atualizando e excluindo dados em associações e subqueries

## Teste seus conhecimentos

8

Guilherme  
Caires  
2026  
39  
39



**IMPACTA**  
EDITORA

## 1. Analise o comando e verifique qual afirmação é a correta:

```
UPDATE EMPREGADOS  
SET SALARIO = SALARIO * 1.10  
WHERE COD_DEPTO = (SELECT COD_DEPTO FROM TABELADEP  
                     WHERE DEPTO = 'CPD');
```

- a) Atualiza o campo salário de todos os empregados.
- b) A sintaxe está errada, pois deve ser utilizado o operador IN em vez do igual.
- c) Nenhum salario será alterado devido à cláusula errada.
- d) Os empregados do departamento 'CPD' terão um aumento de 10% no salário.
- e) Para o comando UPDATE não podemos utilizar subqueries.

## 2. Analise o comando e verifique qual afirmação é a correta:

```
DELETE FROM PEDIDOS  
WHERE DATA_EMISSAO BETWEEN '2006.12.1' AND '2006.12.15'  
AND  
    CODVEN = (SELECT CODVEN FROM VENDEDORES WHERE NOME  
= 'MARCELO');
```

- a) Não podemos utilizar subqueries com DELETE.
- b) A sintaxe está errada.
- c) Serão apagados todos os pedidos do vendedor Marcelo.
- d) Nenhum pedido será apagado.
- e) Serão apagados todos os pedidos do vendedor Marcelo da primeira quinzena de dezembro de 2006.

### 3. Analise o comando e verifique qual afirmação é a correta:

```
UPDATE Empregados  
SET SALARIO *= 1.10  
FROM EMPREGADOS E JOIN TABELADEP D ON E.COD_DEPTO =  
D.COD_DEPTO  
WHERE D.DEPTO = 'CPD';
```

- a) Os empregados do CPD terão uma aumento de 10%.
- b) Os empregados do CPD terão uma aumento de 110%.
- c) Somente os empregado do CPD terão aumento, inclusive os que possuírem COD\_DEPTO nulo.
- d) Nenhum empregado terá aumento.
- e) Deve seu utilizado uma subquery em vez de JOIN.

## 4. Analise o comando e verifique qual afirmação é a correta:

```
TRUNCATE FROM PEDIDOS  
FROM PEDIDOS P JOIN VENDEDORES V ON P.CODVEN = V.CODVEN  
WHERE P.DATA_EMISSAO BETWEEN '2006.12.1' AND '2006.12.15'  
AND  
V.NOME = 'MARCELO';
```

- a) Serão apagados todos os pedidos.
- b) A sintaxe está errada.
- c) Somente os pedidos relacionados com a tabela de vendedores serão excluídos.
- d) Nenhum pedido será excluído.
- e) Serão excluídos os pedidos do vendedor Marcelo da primeira quinzena de dezembro de 2006.

**5. Qual das afirmações a seguir está errada, com relação à atualização de uma tabela com informações de outra?**

- a) Para atualização de tabelas podemos utilizar subqueries ou JOIN.
- b) Somente podemos atualizar uma tabela temporária.
- c) As subqueries correlacionam tabelas e podemos utilizar as informações para atualização de tabelas.
- d) As subqueries podem simplificar o código de atualização.
- e) Utiliza-se a instrução UPDATE para atualizar uma tabela com informações de outra tabela.

# Atualizando excluindo dados em associações e subqueries

Mãos à obra!

8



**IMPACTA**  
EDITORA

## Laboratório 1

### A – Atualizando tabelas com associações e subqueries

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Altere a tabela **TABELACAR**, mudando o salário inicial do cargo **OFFICE BOY** para 600,00;
3. Altere a tabela de cargos, estipulando 10% de aumento para o campo **SALARIO\_INIC** de todos os cargos;
4. Transfira para o campo **SALARIO** da tabela **EMPREGADOS** o salário inicial cadastrado no cargo correspondente da **TABELACAR**;
5. Reajuste os preços de venda de todos os produtos de modo que fiquem 30% acima do preço de custo ( $\text{PRECO\_VENDA} = \text{PRECO\_CUSTO} * 1.3$ );
6. Reajuste os preços de venda dos produtos com **COD\_TIPO** = 5, de modo que fiquem 20% acima do preço de custo;
7. Reajuste os preços de venda dos produtos com descrição do tipo igual à **REGUA**, de modo que fiquem 40% acima do preço de custo. Para isso, considere as seguintes informações:
  - $\text{PRECO\_VENDA} = \text{PRECO\_CUSTO} * 1.4$ ;
  - Para produtos com **TIPOPRODUTO.TIPO** = ‘**REGUA**’;
  - É necessário fazer um **JOIN** de **PRODUTOS** com **TIPOPRODUTO** \*/.
8. Altere a tabela **ITENSPEDIDO** de modo que todos os itens com produto indicado como **VERMELHO** passem a ser **LARANJA**. Considere somente os pedidos com data de entrega em outubro de 2007;

9. Altere o campo **ICMS** tabela **CLIENTES** para 12. Considere apenas clientes dos estados: RJ, RO, AC, RR, MG, PR, SC, RS, MS e MT;

10. Altere o campo **ICMS** para 18, apenas para clientes de SP;

11. Altere o campo **ICMS** da tabela **CLIENTES** para 7. Considere apenas clientes que não sejam dos estados: RJ, RO, AC, RR, MG, PR, SC, RS, MS, MT e SP;

12. Crie a tabela **ESTADOS** com os respectivos campos:

- **COD\_ESTADO**: Inteiro, autonumeração e chave primária;
- **SIGLA**: Char(2);
- **ICMS**: Numérico, tamanho 4 com 2 decimais.

13. Copie os dados coletados do seguinte comando **SELECT** para a tabela **ESTADOS** utilizando um comando **INSERT**:

```
SELECT DISTINCT ESTADO, ICMS FROM CLIENTES  
WHERE ESTADO IS NOT NULL
```

 O **SELECT** deve retornar 21 linhas e não repetir o Estado. Se o resultado for diferente, é porque os **UPDATES** de **ICMS** estão incorretos.

14. Crie o campo **COD\_ESTADO** na tabela **CLIENTES**;

15. Copie para **CLIENTES.COD\_ESTADO** o código do Estado gerado na tabela **ESTADOS**.



# Agrupando dados

9

- ✓ Funções de agregação;
- ✓ GROUP BY.

Guilherme R M Caires  
392.280.768-28



**IMPACTA**  
EDITORA

## 9.1. Introdução

Neste capítulo, você aprenderá como a cláusula **GROUP BY** pode ser utilizada para agrupar vários dados, tornando mais prática sua summarização. Também verá como utilizar funções de agregação para summarizar dados e como a cláusula **GROUP BY** pode ser usada com a cláusula **HAVING** e com os operadores **ALL**, **WITH ROLLUP** e **CUBE**.

## 9.2. Funções de agregação

As funções de agregação fornecidas pelo SQL Server permitem summarizar dados. Por meio delas, podemos somar valores, calcular médias e contar a quantidade de linhas summarizadas. Os cálculos realizados pelas funções de agregação são feitos com base em um conjunto ou grupo de valores, mas retornam um único valor.

Para obter os valores sobre os quais poderão realizar os cálculos, as funções de agregação geralmente são utilizadas com a cláusula **GROUP BY**. Quando não há uma cláusula **GROUP BY**, os grupos de valores podem ser obtidos de uma tabela inteira filtrada pela cláusula **WHERE**.

Ao utilizar funções de agregação, é preciso prestar atenção a valores **NULL**. A maioria das funções ignora esses valores, o que pode gerar resultados inesperados.



Além de utilizar as funções de agregação fornecidas pelo SQL Server, há a possibilidade de criar funções personalizadas.

## 9.2.1. Tipos de função de agregação

A seguir, são descritas as principais funções de agregação fornecidas pelo SQL Server:

- **AVG ( [ ALL | DISTINCT ] expressão )**

Esta função calcula o valor médio do parâmetro **expressão** em determinado grupo, ignorando valores **NULL**. Os parâmetros opcionais **ALL** e **DISTINCT** são utilizados para especificar se a agregação será executada em todos os valores do campo (**ALL**) ou aplicada apenas sobre valores distintos (**DISTINCT**).

Veja um exemplo:

```
USE PEDIDOS;
-- neste caso, o grupo corresponde a toda a tabela EMPREGADOS
SELECT AVG(SALARIO) AS SALARIO_MEDIO
FROM EMPREGADOS;
-- neste caso, o grupo corresponde aos empregados com
-- COD_DEPTO = 2
SELECT AVG(SALARIO) AS SALARIO_MEDIO FROM EMPREGADOS
WHERE COD_DEPTO = 2;
```

- **COUNT ( { [ ALL | DISTINCT ] expressão | \* } )**

Esta função é utilizada para retornar a quantidade de registros existentes não nulos em um grupo. Ao especificar o parâmetro **ALL**, a função não retornará valores nulos. Os parâmetros de **COUNT** têm a mesma função dos parâmetros de **AVG**.

Veja um exemplo:

```
-- neste caso, o grupo corresponde a toda a tabela EMPREGADOS  
SELECT COUNT(*) AS QTD_EMPREGADOS  
FROM EMPREGADOS;  
-- neste caso, o grupo corresponde aos empregados com  
-- COD_DEPTO = 2  
SELECT COUNT(COD_DEPTO) AS QTD_EMPREGADOS FROM EMPREGADOS  
WHERE COD_DEPTO = 2;
```

**! Se colocarmos o nome de um campo como argumento da função COUNT, não serão contados os registros em que o conteúdo desse campo seja NULL.**

- **MIN ( [ ALL | DISTINCT ] expressão)**

Esta função retorna o menor valor não nulo de **expressão** existente em um grupo. Os parâmetros de **MIN** têm a mesma função dos parâmetros de **AVG**.

Veja um exemplo:

```
-- neste caso, o grupo corresponde a toda a tabela EMPREGADOS  
SELECT MIN(SALARIO) AS MENOR_SALARIO FROM EMPREGADOS;  
-- neste caso, o grupo corresponde aos empregados com  
-- COD_DEPTO = 2  
SELECT MIN(SALARIO) AS MENOR_SALARIO FROM EMPREGADOS  
WHERE COD_DEPTO = 2
```

- **MAX ( [ ALL | DISTINCT ] expressão)**

Esta função retorna o maior valor não nulo de **expressão** existente em um grupo. Os parâmetros de **MAX** têm a mesma função dos parâmetros de **AVG**.

Veja um exemplo:

```
-- neste caso, o grupo corresponde a toda a tabela EMPREGADOS  
SELECT MAX(SALARIO) AS MAIOR_SALARIO  
FROM EMPREGADOS;  
-- neste caso, o grupo corresponde aos empregados com  
-- COD_DEPTO = 2  
SELECT MAX(SALARIO) AS MAIOR_SALARIO FROM EMPREGADOS  
WHERE COD_DEPTO = 2
```

- **SUM ( [ ALL | DISTINCT ] expressão)**

Esta função realiza a soma de todos os valores não nulos na **expressão** em um determinado grupo. Os parâmetros de **SUM** têm a mesma função dos parâmetros de **AVG**.

Veja um exemplo:

```
-- neste caso, o grupo corresponde a toda a tabela EMPREGADOS  
SELECT SUM(SALARIO) AS SOMA_SALARIOS  
FROM EMPREGADOS;  
-- neste caso, o grupo corresponde aos empregados com  
-- COD_DEPTO = 2  
SELECT SUM(SALARIO) AS SOMA_SALARIOS FROM EMPREGADOS  
WHERE COD_DEPTO = 2
```

## 9.3. GROUP BY

Utilizando a cláusula **GROUP BY**, é possível agrupar diversos registros com base em uma ou mais colunas da tabela.

Esta cláusula é responsável por determinar em quais grupos devem ser colocadas as linhas de saída. Caso a cláusula **SELECT** contenha funções de agregação, a cláusula **GROUP BY** realiza um cálculo a fim de chegar ao valor sumário para cada um dos grupos.

Quando especificar 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**.

Ao utilizar uma cláusula **GROUP BY**, todas as colunas na lista **SELECT** que não são parte de uma expressão agregada serão usadas para agrupar os resultados obtidos. Para não agrupar os resultados em uma coluna, não se deve colocá-los na lista **SELECT**. Valores **NULL** são agrupados todos em uma mesma coluna, já que são considerados iguais.

Quando utilizamos a cláusula **GROUP BY**, mas não empregamos a cláusula **ORDER BY**, o resultado obtido são os grupos em ordem aleatória, visto que é essencial o uso de **ORDER BY** para determinar a ordem de apresentação dos dados.

Observe, a seguir, a sintaxe da cláusula **GROUP BY**:

```
[ GROUP BY [ ALL ] expressao_group_by [ ,...n ]
[ HAVING <condicaoFiltroGrupo> ] ]
```

Em que:

- **ALL**: É a palavra 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 **ALL** é especificada;
- **expressao\_group\_by**: Também conhecida como coluna agrupada, é uma expressão na qual o agrupamento é realizado. 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 como um alias de coluna determinado na lista de seleção. Além disso, não podemos utilizar em uma **expressao\_group\_by** as colunas de um dos seguintes tipos: **image**, **text** e **ntext**;
- **[HAVING <condicaoFiltroGrupo>]**: Determina uma condição de busca para um grupo ou um conjunto de registros. Essa condição é especificada em **<condicaoFiltroGrupo>**.

# SQL 2014 - Módulo I

Observe o resultado da instrução:

```
SELECT COD_DEPTO, SALARIO FROM EMPREGADOS ORDER BY COD_DEPTO;
```

	COD_DEPTO	SALARIO
4	1	890.00
5	1	4500.00
6	1	800.00
7	1	4500.00
8	1	890.00
9	1	4500.00
10	1	3300.00
11	1	8300.00
12	1	5000.00
13	1	3300.00
14	1	1200.00
15	1	8300.00
16	2	8300.00
17	2	800.00
18	2	600.00
19	2	800.00
20	2	4500.00
21	2	3330.00
22	2	600.00
23	2	500.00

Veja que o campo **COD\_DEPTO** se repete e, portanto, forma grupos. Em uma situação dessas, podemos gerar totalizações para cada um dos grupos utilizando a cláusula **GROUP BY**.

A seguir, veja exemplos da utilização de **GROUP BY**:

- **Exemplo 1**

```
-- Total de salário de cada departamento  
SELECT COD_DEPTO, SUM( SALARIO ) AS TOT_SAL  
FROM EMPREGADOS  
GROUP BY COD_DEPTO  
ORDER BY TOT_SAL;
```

- **Exemplo 2**

```
-- GROUP BY + JOIN  
SELECT E.COD_DEPTO, D.DEPTO, SUM( E.SALARIO ) AS TOT_SAL  
FROM EMPREGADOS E  
    JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO  
GROUP BY E.COD_DEPTO, D.DEPTO  
ORDER BY TOT_SAL;
```

- **Exemplo 3**

```
-- Consulta do tipo RANKING utilizando TOP n + ORDER BY  
-- Os 5 departamentos que mais gastam com salários  
SELECT TOP 5 E.COD_DEPTO, D.DEPTO, SUM( E.SALARIO ) AS TOT_SAL  
FROM EMPREGADOS E  
    JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO  
GROUP BY E.COD_DEPTO, D.DEPTO  
ORDER BY TOT_SAL DESC;
```

- **Exemplo 4**

```
-- Os 10 clientes que mais compraram em Janeiro de 2007  
SELECT TOP 10 C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRADO  
FROM PEDIDOS P JOIN CLIENTES C ON P.CODCLI = C.CODCLI  
WHERE P.DATA_EMISAO BETWEEN '2007.1.1' AND '2007.1.31'  
GROUP BY C.CODCLI, C.NOME  
ORDER BY TOT_COMPRADO DESC;
```

## 9.3.1. Utilizando ALL

ALL inclui todos os grupos e conjuntos de resultados. A seguir, veja um exemplo da utilização de ALL:

```
-- Clientes que compraram em janeiro de 2007. Veremos que  
-- todas as linhas do resultado terão um total não nulo.  
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRADO  
FROM PEDIDOS P JOIN CLIENTES C ON P.CODCLI = C.CODCLI  
WHERE P.DATA_EMISAO BETWEEN '2007.1.1' AND '2007.1.31'  
GROUP BY C.CODCLI, C.NOME;  
  
-- Neste caso, aparecerão também os clientes que não  
-- compraram. Totais estarão nulos.  
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRADO  
FROM PEDIDOS P JOIN CLIENTES C ON P.CODCLI = C.CODCLI  
WHERE P.DATA_EMISAO BETWEEN '2007.1.1' AND '2007.1.31'  
GROUP BY ALL C.CODCLI, C.NOME;
```

## 9.3.2. Utilizando HAVING

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. Ela é utilizada para estreitar um conjunto de resultados por meio de critérios e valores agregados e para filtrar linhas após o agrupamento ter sido feito e antes dos resultados serem retornados ao cliente.

É importante lembrar que essa cláusula só pode ser utilizada em parceria com **GROUP BY**. Se uma consulta é feita sem **GROUP BY**, a cláusula **HAVING** pode ser usada como cláusula **WHERE**.

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.

O código do exemplo a seguir utiliza a cláusula **HAVING** para consultar os departamentos que totalizam mais de R\$100.000,00 em salários:

```
SELECT E.COD_DEPTO, D.DEPTO, SUM( E.SALARIO ) AS TOT_SAL
FROM EMPREGADOS E
    JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO
GROUP BY E.COD_DEPTO, D.DEPTO HAVING SUM(E.SALARIO) > 100000
ORDER BY TOT_SAL;
```

O próximo código consulta os clientes que compraram mais de R\$5.000,00 em janeiro de 2007:

```
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRA
FROM PEDIDOS P JOIN CLIENTES C ON P.CODCLI = C.CODCLI
WHERE P.DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31'
GROUP BY C.CODCLI, C.NOME HAVING SUM(P.VLR_TOTAL) > 5.000
ORDER BY TOT_COMPRA;
```

Já o próximo código consulta os clientes que não realizaram compras em janeiro de 2007:

```
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRA
FROM PEDIDOS P JOIN CLIENTES C ON P.CODCLI = C.CODCLI
WHERE P.DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31'
GROUP BY ALL C.CODCLI, C.NOME HAVING SUM(P.VLR_TOTAL) IS NULL;
```

## 9.3.3. Utilizando WITH ROLLUP

Esta cláusula determina que, além das linhas normalmente retornadas 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 essa ordem seja alterada, a quantidade de linhas produzidas pode ser afetada.

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

Suponha que esteja trabalhando em um banco de dados com as seguintes características:

- O cadastro de produtos está organizado em categorias (tipos);
- Há vários produtos que pertencem a uma mesma categoria;
- As categorias de produtos são armazenadas na tabela **TIPOPRODUTO**.

O **SELECT** a seguir mostra as vendas de cada categoria de produto (**TIPOPRODUTO**) que os vendedores (tabela **VENDEDORES**) realizaram para cada cliente (tabela **CLIENTES**) no primeiro semestre de 2006:

```
SELECT
    V.NOME AS VENDEDOR, C.NOME AS CLIENTE,
    T.TIPO AS TIPO_PRODUTO, SUM( I.QUANTIDADE ) AS QTD_TOT
FROM
    PEDIDOS Pe JOIN CLIENTES C ON Pe.CODCLI = C.CODCLI
        JOIN VENDEDORES V ON Pe.CODVEN = V.CODVEN
        JOIN ITENSPEIDO I ON Pe.NUM_PEDIDO = I.NUM_PEDIDO
        JOIN PRODUTOS Pr ON I.ID_PRODUTO = Pr.ID_PRODUTO
        JOIN TIPOPRODUTO T ON Pr.COD_TIPO = T.COD_TIPO
WHERE Pe.DATA_EMISSAO BETWEEN '2006.1.1' AND '2006.6.30'
GROUP BY V.NOME , C.NOME, T.TIPO;
```

Suponha que o resultado retornado seja o seguinte:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
1	CELSOM MARTINS	3R (ARISTEU,ADALTON)	ACES.CHAVEIRO	111
2	CELSOM MARTINS	3R (ARISTEU,ADALTON)	ACESSORIOS P/CANETA	170
3	CELSOM MARTINS	3R (ARISTEU,ADALTON)	CANETA	600
4	CELSOM MARTINS	3R (ARISTEU,ADALTON)	CHAVEIRO	513
5	CELSOM MARTINS	3R (ARISTEU,ADALTON)	MATL DIVERSOS	982
6	CELSOM MARTINS	3R (ARISTEU,ADALTON)	PORTA LAPIS	1352
7	CELSOM MARTINS	3R (ARISTEU,ADALTON)	REGUA	19
8	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	CANETA	153
9	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	CHAVEIRO	295
10	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	MATL DIVERSOS	211
11	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	PORTA LAPIS	162

Note que um dos vendedores é **CELSOM MARTINS** e que alguns de seus clientes são **3R (ARISTEU,ADALTON)** e **ALLAN HEBERT RELOGIOS E PRESENTES**. Também é possível perceber, por exemplo, que o cliente **3R (ARISTEU,ADALTON)** comprou 111 unidades de produtos do tipo **ACES.CHAVEIRO** do vendedor **CELSOM MARTINS**, enquanto **ALLAN HEBERT RELOGIOS E PRESENTES** comprou 153 produtos do tipo **CANETA** do mesmo vendedor.

Terminados os registros de vendas do **CELSOM MARTINS**, iniciam-se os registros de venda do próximo vendedor, e assim por diante, como você pode ver a seguir:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
168	CELSOM MARTINS	VILSON CORDEIRO DO ROSARIO	REGUA	143
169	CELSOM MARTINS	VILSON CORDEIRO DO ROSARIO	YO-YO	516
170	DORINHA	3R (ARISTEU,ADALTON)	CANETA	750
171	DORINHA	3R (ARISTEU,ADALTON)	PORTA LAPIS	30

# SQL 2014 - Módulo I

Agora, acrescente a cláusula **WITH ROLLUP** após a linha de **GROUP BY**. O código anterior ficará assim:

```
SELECT
    V.NOME AS VENDEDOR, C.NOME AS CLIENTE,
    T.TIPO AS TIPO_PRODUTO, SUM( I.QUANTIDADE ) AS QTD_TOT
FROM
    PEDIDOS Pe JOIN CLIENTES C ON Pe.CODCLI = C.CODCLI
        JOIN VENDEDORES V ON Pe.CODVEN = V.CODVEN
        JOIN ITENSPEIDO I ON Pe.NUM_PEDIDO = I.NUM_PEDIDO
        JOIN PRODUTOS Pr ON I.ID_PRODUTO = Pr.ID_PRODUTO
        JOIN TIPOPRODUTO T ON Pr.COD_TIPO = T.COD_TIPO
WHERE Pe.DATA_EMISSAO BETWEEN '2006.1.1' AND '2006.6.30'
GROUP BY V.NOME , C.NOME, T.TIPO
WITH ROLLUP;
```

O resultado será o seguinte:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
1	CELSOM MARTINS	3R (ARISTEU,ADALTON)	ACES.CHAVEIRO	111
2	CELSOM MARTINS	3R (ARISTEU,ADALTON)	ACESSORIOS P/CANETA	170
3	CELSOM MARTINS	3R (ARISTEU,ADALTON)	CANETA	600
4	CELSOM MARTINS	3R (ARISTEU,ADALTON)	CHAVEIRO	513
5	CELSOM MARTINS	3R (ARISTEU,ADALTON)	MATL DIVERSOS	982
6	CELSOM MARTINS	3R (ARISTEU,ADALTON)	PORTA LAPIS	1352
7	CELSOM MARTINS	3R (ARISTEU,ADALTON)	REGUA	19
8	CELSOM MARTINS	3R (ARISTEU,ADALTON)	NULL	3747
9	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	CANETA	153
10	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	CHAVEIRO	295
11	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	MATL DIVERSOS	211
12	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	PORTA LAPIS	162
13	CELSOM MARTINS	ALLAN HEBERT RELOGIOS E PRESENTES	NULL	821

Observe na figura anterior que, após a última linha do vendedor **CELSOM MARTINS**, existe um **NULL** na coluna **TIPO\_PRODUTO**, o que significa que o valor apresentado na coluna **QTD\_TOT** corresponde ao total vendido por esse vendedor para o cliente **3R (ARISTEU.ADALTON)**, ou seja, a coluna **TIPO\_PRODUTO (NULL)** não foi considerada para a totalização. Isso se repetirá até o último cliente que comprou de **CELSOM MARTINS**.

Antes de iniciar as totalizações do vendedor seguinte, existe uma linha na qual apenas o nome do vendedor não é **NULL**, o que significa que o total apresentado na coluna **QTD\_TOT** representa o total vendido pelo vendedor **CELSOM MARTINS**, ou seja, **55912** produtos, independentemente do cliente e do tipo de produto:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
213	CELSOM MARTINS	VILSON CORDEIRO DO ROSARIO	YO-YO	516
214	CELSOM MARTINS	VILSON CORDEIRO DO ROSARIO	NULL	3266
215	CELSOM MARTINS	NULL	NULL	55912
216	DORINHA	3R (ARISTEU.ADALTON)	CANETA	750
217	DORINHA	3R (ARISTEU.ADALTON)	POR TA LAPIS	30

Na última linha do resultado, temos **NULL** nas 3 primeiras colunas. O total corresponde ao total vendido (**1022534**) no período mencionado, independentemente do vendedor, do cliente ou do tipo de produto:

1931	MARCELO	WALEU IND E COM DE PLASTICOS LTDA	POR TA LAPIS	110
1932	MARCELO	WALEU IND E COM DE PLASTICOS LTDA	NULL	805
1933	MARCELO	NULL	NULL	216732
1934	NULL	NULL	NULL	1022534

## 9.3.4. Utilizando WITH CUBE

A cláusula **WITH 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 de grupos e de subgrupos no conjunto de resultados.

Visto que a cláusula **WITH CUBE** é responsável por retornar todas as combinações possíveis de grupos e de 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 apresentada.

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 com **GROUP BY**, gera totais e subtotais, apresentando vários agrupamentos de acordo com as colunas definidas com **GROUP BY**.

Para explicar o que faz **WITH CUBE**, considere o exemplo utilizado para **WITH ROLLUP**. No lugar desta última cláusula, utilize **WITH CUBE**. O código ficará assim:

```
SELECT
    V.NOME AS VENDEDOR, C.NOME AS CLIENTE,
    T.TIPO AS TIPO_PRODUTO, SUM( I.QUANTIDADE ) AS QTD_TOT
FROM
    PEDIDOS Pe JOIN CLIENTES C ON Pe.CODCLI = C.CODCLI
        JOIN VENDEDORES V ON Pe.CODVEN = V.CODVEN
        JOIN ITENSPEIDO I ON Pe.NUM_PEDIDO = I.NUM_PEDIDO
        JOIN PRODUTOS Pr ON I.ID_PRODUTO = Pr.ID_PRODUTO
        JOIN TIPOPRODUTO T ON Pr.COD_TIPO = T.COD_TIPO
WHERE Pe.DATA_EMISSAO BETWEEN '2006.1.1' AND '2006.6.30'
GROUP BY V.NOME , C.NOME, T.TIPO
WITH CUBE;
```

O resultado é o seguinte:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
1	DORINHA	ABILIO MARTINS PINTO JR-ME	ABRIDOR	10
2	LEIA	ABILIO MARTINS PINTO JR-ME	ABRIDOR	10
3	NULL	ABILIO MARTINS PINTO JR-ME	ABRIDOR	20
4	LEIA	ALAMBRINDES GRAFICA EDITORA LTDA.	ABRIDOR	2200
5	NULL	ALAMBRINDES GRAFICA EDITORA LTDA.	ABRIDOR	2200

Esse tipo de resultado não existia com a opção **WITH ROLLUP**. Neste caso, a coluna **VENDEDOR** é **NULL** e o total corresponde ao total de produtos do tipo **ABRIDOR** comprado pelo cliente **ABILIO** (20 produtos). Já **ALAMBRINDES GRAFICA EDITORA LTDA** comprou 2200 produtos do tipo **ABRIDOR**. **WITH CUBE** inclui todas as outras subtotalizações possíveis no resultado.

Neste outro trecho do mesmo resultado retornado, as colunas **VENDEDOR** e **CLIENTE** são nulas e o total corresponde ao total vendido de produtos do tipo **CANETA**, ou seja, **526543**:

1211	NULL	ZINHO'S COM DE BRINDES LTDA ME	CANETA	8450
1212	NULL	NULL	CANETA	526543

Seguindo a mesma regra, se apenas o **CLIENTE** não é nulo, o total corresponde ao total comprado por este cliente, independentemente de vendedor ou de produto:

3531	LEIA	VITOR BRIGIO	NULL	1705
3532	NULL	VITOR BRIGIO	NULL	1705

### Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- As funções de agregação fornecidas pelo SQL Server permitem summarizar dados. Por meio delas, é possível somar valores, calcular média e contar resultados. Os cálculos feitos pelas funções de agregação são feitos com base em um conjunto ou grupo de valores, porém retornam um único valor. Para obter os valores sobre os quais você poderá realizar os cálculos, geralmente são utilizadas as funções de agregação com a cláusula **GROUP BY**;
- Utilizando a cláusula **GROUP BY**, é possível agrupar diversos registros com base em uma ou mais colunas da tabela.

9

# Agrupando dados

Teste seus conhecimentos

Guilherme F. Caires  
392.280.168-28



**IMPACTA**  
EDITORA

## 1. Qual destas funções não é uma função de agregação?

- a) AVG
- b) COUNT
- c) MAX
- d) MIN
- e) STR

## 2. O que está faltando no comando adiante?

```
SELECT TOP 5 E.COD_DEPTO, D.DEPTO, SUM( E.SALARIO ) AS  
TOT_SAL  
FROM EMPREGADOS E  
JOIN TABELADEP D ON E.COD_DEPTO = D.COD_DEPTO
```

- a) ORDER BY
- b) HAVING
- c) GROUP BY
- d) O comando está correto.
- e) WHERE

**3. Qual é a diferença entre GROUP BY e GROUP BY ALL?**

- a) Não há diferença entre a cláusula GROUP BY e a GROUP BY ALL.
- b) GROUP BY apresenta todas as linhas, mesmo não agrupadas.
- c) GROUP BY ALL deve ser utilizada com funções de agrupamento.
- d) Enquanto GROUP BY apresenta somente os grupos selecionados, GROUP BY ALL apresenta todos os grupos mesmo que não estejam na cláusula WHERE.
- e) GROUP BY ALL apresenta apenas os grupos selecionados.

**4. Escolha a resposta que apresenta os departamentos que gastam mais do que R\$15.000,00:**

- a) WHERE SALARIO>15000
- b) WHERE SALARIO>15000 GROUP BY SALARIO>15000
- c) GROUP BY SALARIO>15000
- d) GROUP BY DEPARTAMENTO HAVING SUM(SALARIO)>15000
- e) WHERE SUM(SALARIO)>15000

**5. Com GROUP BY também podemos exibir totais.  
Para isso, qual comando é o correto?**

- a) GROUP BY ... WITH ROLLUP
- b) GROUP BY ... CUBE
- c) GROUP BY ... ROLLUP
- d) GROUP BY ... WITH ROLUP
- e) Todas as alternativas anteriores  
estão corretas.

9

# Agrupando dados

Mãos à obra!

Guilherme  
392.280.700-28  
Fernando  
Caires



**IMPACTA**  
EDITORA

## Laboratório 1

### A – Realizando consultas e ordenando dados

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Calcule a média de preço de venda (**PRECO\_VENDA**) do cadastro de **PRODUTOS**;
3. Calcule a quantidade de pedidos cadastrados em janeiro de 2007 (o maior e o menor valor total, **VLR\_TOTAL**);
4. Calcule o valor total vendido (soma de **PEDIDOS.VLR\_TOTAL**) em janeiro de 2007;
5. Calcule o valor total vendido pelo vendedor de código 1 em janeiro de 2007;
6. Calcule o valor total vendido pela vendedora **LEIA** em janeiro de 2007;
7. Calcule o valor total vendido pelo vendedor **MARCELO** em janeiro de 2007;
8. Calcule o valor da comissão (soma de **PEDIDOS.VLR\_TOTAL \* VENDEDORES.PORC\_COMISSAO/100**) que a vendedora **LEIA** recebeu em janeiro de 2007;
9. Calcule o valor da comissão que o vendedor **MARCELO** recebeu em janeiro de 2007;
10. Liste os totais vendidos por cada vendedor (mostrar **VENDEDOR.NOME** e a soma de **PEDIDOS.VLR\_TOTAL**) em janeiro de 2007. Deve exibir o nome do vendedor;

11. Liste o total comprado por cada cliente em janeiro de 2007. Deve mostrar o nome do cliente;
12. Liste o valor e a quantidade total vendida de cada produto em janeiro de 2007;
13. Liste os totais vendidos por cada vendedor em janeiro de 2007. Deve exibir o nome do vendedor e mostrar apenas os vendedores que venderam mais de R\$80.000,00;
14. Liste o total comprado por cada cliente em janeiro de 2007. Deve mostrar o nome do cliente e somente os clientes que compraram mais de R\$6.000,00;
15. Liste o total vendido de cada produto em janeiro de 2007. Deve mostrar apenas os produtos que venderam mais de R\$16.000,00;
16. Liste o total comprado por cada cliente em janeiro de 2007. Deve mostrar o nome do cliente e somente os 10 primeiros do ranking;
17. Liste o total vendido de cada produto em janeiro de 2007. Deve mostrar os 10 produtos que mais venderam;
18. Liste o total vendido em cada um dos meses de 2006.



# Comandos Adicionais

# 10

- ✓ Funções de cadeia de caracteres;
- ✓ Função CASE;
- ✓ UNION;
- ✓ EXCEPT e INTERSECT.



**IMPACTA**  
EDITORA

## 10.1. Funções de cadeia de caracteres

Estas funções executam operações em um valor de entrada de cadeia de caracteres e retornam o mesmo dado trabalhado. Por exemplo, podemos concatenar, replicar ou inverter os dados de entrada. A seguir, vamos apresentar as funções de cadeia de caracteres principais fornecidas pelo SQL Server:

- **LEN (expressão\_string)**

Esta função retorna o número de caracteres especificado no parâmetro **expressão\_string**.

Veja um exemplo:

```
SELECT LEN ('Brasil');
```

(No column name)	1
	6

- **REPLICATE (expressão\_string, mult)**

Esta função repete os caracteres do parâmetro **expressão\_string** pelo número de vezes especificado no parâmetro **mult**.

Veja um exemplo:

```
SELECT REPLICATE ('Teste', 4);
```

(No column name)	1
	TesteTesteTesteTeste

- **REVERSE (expressão\_string)**

Esta função retorna a ordem inversa do parâmetro **expressão\_string**.

Veja um exemplo:

```
SELECT REVERSE ('amina');
```

Results	
	(No column name)
1	anima

- **STR (número [, tamanho [, decimal]] )**

Esta função retorna dados do tipo **string** a partir de dados numéricos.

Veja um exemplo:

```
SELECT STR (213);
```

Results	
	(No column name)
1	213

- **SUBSTRING (expressão, início, tamanho)**

Esta função retorna uma parte dos caracteres do parâmetro **expressão** a partir dos valores de **início** e **tamanho**.

Veja um exemplo:

```
SELECT SUBSTRING ('Paralelepípedo', 3, 7);
```

Results	
	(No column name)
1	ralelep

- **CONCAT (expr1, expr2 [, exprN])**

Esta função concatena as expressões retornando um string. As expressões podem ser de qualquer tipo.

Veja um exemplo:

```
SELECT CONCAT ('SQL ', 'módulo ', 'I');
```

Results		Messages
(No column name)		
1	SQL módulo I	

- **CHARINDEX (expr1, expr2 [, exprN])**

Esta função pesquisa uma string dentro de outra, retornando a posição encontrada. Caso não encontre o valor pesquisado, o retorno será zero.

Veja um exemplo:

```
SELECT CHARINDEX ('A', 'CASA')
```

Results		Messages
(No column name)		
1	2	

- **FORMAT (expressão, formato)**

Formata uma **expressão** numérica ou date/time no formato definido por um string de formatação.

A seguir, temos alguns exemplos de caracteres usados na formatação de strings:

- Caracteres para formatação de números:

0 (zero)	Define uma posição numérica. Se não existir número na posição, aparece o zero.
#	Define uma posição numérica. Se não existir número na posição, fica vazio.
. (ponto)	Separador de decimal.
, (vírgula)	Separador de milhar.
%	Mostra o sinal e o número multiplicado por 100.

Qualquer outro caractere inserido na máscara de formatação será exibido normalmente na posição em que foi colocado.

- Caracteres para formatação de data:

<b>d</b>	Dia com 1 ou 2 dígitos.
<b>dd</b>	Dia com 2 dígitos.
<b>ddd</b>	Abreviação do dia da semana.
<b>dddd</b>	Nome do dia da semana.
<b>M</b>	Mês com 1 ou 2 dígitos.
<b>MM</b>	Mês com 2 dígitos.
<b>MMM</b>	Abreviação do nome do mês.
<b>MMMM</b>	Nome do mês.
<b>yy</b>	Ano com 2 dígitos.
<b>yyyy</b>	Ano com 4 dígitos.
<b>hh</b>	Hora de 1 a 12.
<b>HH</b>	Hora de 0 a 23.
<b>mm</b>	Minutos.
<b>ss</b>	Segundos.
<b>fff</b>	Milésimos de segundo.

Veja um exemplo:

```
SELECT FORMAT (GETDATE(), 'dd/MM/yyyy');
```



## 10.2. Função CASE

Os valores pertencentes a uma coluna podem ser testados por meio da cláusula **CASE** em conjunto com o comando **SELECT**. Dessa maneira, é possível aplicar diversas condições de validação em uma consulta.

No exemplo a seguir, **CASE** é utilizado para verificar se os funcionários da tabela **EMPREGADOS** são ou não sindicalizados:

```
SELECT NOME, SALARIO, CASE SINDICALIZADO
    WHEN 'S' THEN 'Sim'
    WHEN 'N' THEN 'Não'
    ELSE 'N/C'
END AS [Sindicato?],
DATA ADMISSAO
FROM EMPREGADOS;
```

Já no próximo exemplo, verificamos em qual dia da semana os empregados foram admitidos:

```
SELECT NOME, SALARIO, DATA ADMISSAO,
CASE DATEPART(WEEKDAY, DATA ADMISSAO)
    WHEN 1 THEN 'Domingo'
    WHEN 2 THEN 'Segunda-Feira'
    WHEN 3 THEN 'Terça-Feira'
    WHEN 4 THEN 'Quarta-Feira'
    WHEN 5 THEN 'Quinta-Feira'
    WHEN 6 THEN 'Sexta-Feira'
    WHEN 7 THEN 'Sábado'
END AS DIA SEMANA
FROM EMPREGADOS;
```

## 10.3. UNION

A cláusula **UNION** combina resultados de duas ou mais queries em um conjunto de resultados simples, incluindo todas as linhas de todas as queries combinadas. Ela é utilizada quando é preciso recuperar todos os dados de duas tabelas, sem fazer associação entre elas.

Para utilizar **UNION**, é necessário que o número e a ordem das colunas nas queries sejam iguais, bem como os tipos de dados sejam compatíveis. Se os tipos de dados forem diferentes em precisão, escala ou extensão, as regras para determinar o resultado serão as mesmas das expressões de combinação.

O operador **UNION**, por padrão, elimina linhas duplicadas do conjunto de resultados.

Veja o seguinte exemplo:

```
SELECT NOME, FONE1 FROM CLIENTES
UNION
SELECT NOME, FONE1 FROM CLIENTES ORDER BY NOME;
```

### 10.3.1. Utilizando UNION ALL

A **UNION ALL** é a cláusula responsável por unir informações obtidas a partir de diversos comandos **SELECT**. Para obter esses dados, não há necessidade de que as tabelas que os possuem estejam relacionadas.

Para utilizar a cláusula **UNION ALL**, é necessário considerar as seguintes regras:

- 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 os mesmos tipos de dados em sequência. Por exemplo, uma vez que a segunda coluna do primeiro **SELECT** baseia-se no tipo de dado decimal, é preciso que as segundas colunas dos outros **SELECT** também apresentem um tipo de dado 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 a exibição única de dados repetidos em mais de uma tabela.

Enquanto **UNION**, por padrão, elimina linhas duplicadas do conjunto de resultados, **UNION ALL** inclui todas as linhas nos resultados e não remove as linhas duplicadas. A seguir, veja um exemplo da utilização de **UNION ALL**:

```
SELECT NOME, FONE1 FROM CLIENTES
UNION ALL
SELECT NOME, FONE1 FROM CLIENTES ORDER BY NOME;
```

### 10.4. EXCEPT e INTERSECT

Os resultados de duas instruções **SELECT** podem ser comparados por meio dos operadores **EXCEPT** e **INTERSECT**, resultando, assim, em novos valores.

O operador **INTERSECT** retorna os valores encontrados nas duas consultas, tanto a que está à esquerda quanto a que está à direita do operador na sintaxe a seguir:

```
<instrução_select_1>
INTERSECT
<instrução_select_2>
```

O operador **EXCEPT** retorna os valores da consulta à esquerda que não se encontram também na consulta à direita:

```
<instrução_select_1>
EXCEPT
<instrução_select_2>
```

Os operadores **EXCEPT** e **INTERSECT** podem ser utilizados juntamente com outros operadores em uma expressão. Neste caso, a avaliação da expressão segue uma ordem específica. Em primeiro lugar, as expressões em parênteses, em seguida, o operador **INTERSECT** e, por fim, o operador **EXCEPT**, avaliado da esquerda para a direita, de acordo com sua posição na expressão.

Também podemos utilizar **EXCEPT** e **INTERSECT** para comparar mais de duas queries. Quando for assim, a conversão dos tipos de dados é feita pela comparação de duas consultas ao mesmo tempo, de acordo com a ordem de avaliação que apresentamos.

 Para utilizar **EXCEPT** e **INTERSECT**, é necessário que as colunas estejam em mesmo número e ordem em todas as consultas, e que os tipos de dados sejam compatíveis.

Primeiramente, vamos escolher o banco de dados a ser utilizado:

```
USE PEDIDOS;
```

A seguir, temos exemplos que demonstram a utilização dos operadores **EXCEPT** e **INTERSECT**:

- **Exemplo 1**

A instrução a seguir lista os códigos de departamento que possuem funcionários que ganham mais de \$5.000,00:

```
SELECT COD_DEPTO FROM TABELADEP  
INTERSECT  
SELECT COD_DEPTO FROM EMPREGADOS  
WHERE SALARIO > 5000
```

O resultado do código anterior é exibido a seguir:

COD_DEPTO
1
2

- **Exemplo 2**

O exemplo a seguir lista os departamentos sem um funcionário sequer cadastrado:

```
SELECT COD_DEPTO FROM TABELADEP  
EXCEPT  
SELECT COD_DEPTO FROM EMPREGADOS
```

O resultado do código anterior é exibido a seguir:

COD_DEPTO
10
13

- **Exemplo 3**

O exemplo a seguir lista os cargos que não possuem um funcionário sequer cadastrado:

```
SELECT COD_CARGO FROM TABELACAR  
EXCEPT  
SELECT COD_CARGO FROM EMPREGADOS
```

O resultado do código anterior é exibido a seguir:

COD_CARGO
1
2
3

- **Exemplo 4**

O código a seguir consulta os clientes que compraram em janeiro de 2007:

```
SELECT CODCLI FROM CLIENTES  
INTERSECT  
SELECT CODCLI FROM PEDIDOS  
WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31'
```

O resultado do código anterior é exibido a seguir:

CODCLI
1
2
3
4
5
6

245 linhas

- **Exemplo 5**

O código a seguir consulta os clientes que não compraram em janeiro de 2007:

```
SELECT CODCLI FROM CLIENTES
EXCEPT
SELECT CODCLI FROM PEDIDOS
WHERE DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31'
```

O resultado do código anterior é exibido a seguir:

CODCLI
1 6
2 33
3 37
4 66
5 70
6 240

## Pontos principais

**Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.**

- Existem várias funções que auxiliam com valores de entrada de cadeia de caracteres como: LEN, REVERSE, CONCAT etc.;
- A função CASE permite que selecionemos valores conforme uma ou mais cláusulas específicas;
- UNION permite a união de duas ou mais consultas em uma única;
- Uma forma de podermos comparar consultas é a utilização dos comandos INTERSECT e EXCEPT.

# **Comandos Adicionais**

## **Teste seus conhecimentos**

**10**

Guilherme  
392.280.1628  
Caires



**IMPACTA**  
EDITORA

## 1. Qual destas funções não é uma função de texto?

- a) LEN
- b) STR
- c) COUNT
- d) CHARINDEX
- e) REPLICATE

## 2. O que está faltando no comando a seguir?

```
SELECT NOME, SALARIO, CASE SINDICALIZADO  
    WHEN 'S' THEN 'Sim'  
    WHEN 'N' THEN 'Não'  
    ELSE 'N/C'  
    AS [Sindicato?] ,  
    DATA ADMISSAO  
FROM EMPREGADOS;
```

- a) ORDER BY
- b) END
- c) GROUP BY
- d) O comando está correto.
- e) WHERE

**3. Qual afirmação está errada em relação à cláusula UNION?**

- a) A cláusula UNION é responsável pela união dos resultados de duas ou mais consultas.
- b) UNION e UNION ALL possuem funções idênticas.
- c) Ao utilizamos UNION ALL, as consultas devem possuir a mesma quantidade de colunas.
- d) Ao utilizarmos UNION, o SQL elimina as linhas duplicadas.
- e) UNION ALL apresenta todas as linhas, mesmo duplicadas.

**4. Qual comando é utilizado para comparação de consultas?**

- a) WHERE
- b) INTERSECT
- c) SELECT
- d) SELECT com subqueries.
- e) UNION

**5. Analise o comando a seguir e assinale a resposta correta:**

```
SELECT COD_CARGO FROM TABELACAR  
EXCEPT  
SELECT COD_CARGO FROM EMPREGADOS
```

- a) A sintaxe está errada.
- b) A consulta não retorna nenhuma informação.
- c) A consulta retornará todos os cargos que possuem empregados.
- d) A consulta retornará todos os cargos que não possuem empregados.
- e) A consulta retornará todos os cargos, possuindo empregados ou não.

# Comandos Adicionais

## Mãos à obra!

10

Guilherme  
392.280.7000  
Caires



**IMPACTA**  
EDITORA

## Laboratório 1

### A – Realizando consultas e ordenando dados

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Apresente uma listagem com o nome, salário, data de admissão e o mês da data de admissão do empregado;
3. Apresente o nome do cliente, concatenando endereço para que fique em apenas uma única coluna (exemplo: AV. PRES. VARGAS, 364 – CENTRO - GARIBALDI/RS);
4. Selecione os empregados apresentando o 1º nome e o dia e mês (dd/mm) de aniversário;
5. Execute o script **Cap10\_CriaTabela.sql**;
6. Compare as tabelas Produtos e TBPROD, apresentando os registros que são iguais;
7. Compare as tabelas PRODUTOS e TBPROD, apresentando os registros que aparecem somente em PRODUTOS;
8. Compare as tabelas PRODUTOS e TBPROD, apresentando os registros que aparecem somente em TBPROD.