

IMPACTA



Editora
IMPACTA





SQL 2016 - Criando Sistemas de Banco de Dados (online)





Créditos

Copyright © Monte Everest Participações e Empreendimentos 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 Monte Everest Participações e Empreendimentos 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 2016 - Criando Sistemas de Banco de Dados (online)

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Supervisão de Desenvolvimento Digital

Alexandre Hideki Chicaoka

Produção, Gravação, Edição de Vídeo e Finalização

Xandros Luiz de Oliveira Almeida
Rebecca Labre Rodrigues dos Santos

Roteirização

Hélio de Almeida Monteiro Júnior

Curso ministrado por

Hélio de Almeida Monteiro Júnior

Edição e Revisão final

Fernanda Monteiro Laneri

Diagramação

Bruno de Oliveira Santos

Edição nº 1 | 1827_0_EAD

Abril/ 2017

Sobre o instrutor do curso:

Hélio de Almeida é consultor em soluções de desenvolvimento de aplicações .NET, VBA e banco de dados SQL Server, e soluções em análises de estratégia de negócios (BI). Formado em gestão da tecnologia da informação, Hélio é instrutor de informática desde 1994 e ministra cursos de VBA, C# e SQL na Impacta Tecnologia desde 2005.

Este material é uma nova obra derivada da seguinte obra original, produzida por Monte Everest Participações e Empreendimentos Ltda., em Ago/2016:
SQL 2016 – Módulo I
Autoria: Daniel Paulo Tamarosi Salvador
Nº registro BN: 735276

Sumário

Apresentação	08
Aulas 1 a 5 - Introdução ao SQL Server 2016	09
1.1. Banco de dados relacional	10
1.2. Design do banco de dados	10
1.2.1. Modelo descritivo	10
1.2.2. Modelo conceitual	12
1.2.3. Modelo lógico	13
1.2.4. Modelo físico	13
1.2.5. Dicionário de dados	15
1.3. Normalização de dados	16
1.3.1. Regras de normalização	16
1.4. Arquitetura cliente / servidor	20
1.5. As linguagens SQL e T-SQL	22
1.6. SQL Server	23
1.6.1. Componentes	23
1.6.2. Objetos de banco de dados	23
1.6.2.1. Tabelas	23
1.6.2.2. Índices	24
1.6.2.3. CONSTRAINT	24
1.6.2.4. VIEW (Visualização)	24
1.6.2.5. PROCEDURE (Procedimento armazenado)	24
1.6.2.6. FUNCTION (Função)	24
1.6.2.7. TRIGGER (Gatilho)	24
1.7. Ferramentas de gerenciamento	25
1.8. SQL Server Management Studio (SSMS)	26
1.8.1. Inicializando o SSMS	26
1.8.2. Interface	28
1.8.3. Executando um comando	32
1.8.4. Comentários	34
1.8.5. Opções	34
1.8.6. Salvando scripts	36
1.8.7. Soluções e Projetos	37
Pontos principais	39
Teste seus conhecimentos	41
Mãos à obra!	43
Aulas 6 e 7 - Criando um banco de dados	47
1.1. Introdução	48
1.2. CREATE DATABASE	48
1.3. CREATE TABLE	50
1.4. Tipos de dados	53
1.4.1. Numéricos exatos	53
1.4.2. Numéricos aproximados	54
1.4.3. Data e hora	55
1.4.4. Strings de caracteres ANSI	55
1.4.5. Strings de caracteres Unicode	56
1.4.6. Strings binárias	57
1.4.7. Outros tipos de dados	57
1.5. Campo de autonumeração (IDENTITY)	58
1.6. Constraints	58
1.6.1. Nulabilidade	59
1.6.2. Tipos de constraints	59
1.6.2.1. PRIMARY KEY (chave primária)	59
1.6.2.2. UNIQUE	60
1.6.2.3. CHECK	61
1.6.2.4. DEFAULT	61
1.6.2.5. FOREIGN KEY (chave estrangeira)	62

SQL 2016 - Criando Sistemas de Banco de Dados (online)

1.6.3.	Criando constraints	63
1.6.3.1.	Criando constraints com CREATE TABLE	63
1.6.3.2.	Criando constraints com ALTER TABLE	66
1.6.3.3.	Criando constraints graficamente	67
Pontos principais	77	
Teste seus conhecimentos	79	
Mãos à obra!.....	83	
Aulas 10 a 13 - Manipulando dados		
1.1.	Constantes	88
1.2.	Inserindo dados	90
1.2.1.	INSERT posicional.....	92
1.2.2.	INSERT declarativo.....	93
1.3.	Utilizando TOP em uma instrução INSERT	93
1.4.	OUTPUT	94
1.4.1.	OUTPUT em uma instrução INSERT	95
1.5.	Atualizando e excluindo dados.....	96
1.6.	UPDATE.....	97
1.6.1.	Alterando dados de uma coluna	98
1.6.2.	Alterando dados de diversas colunas.....	99
1.6.3.	Utilizando TOP em uma instrução UPDATE	99
1.7.	DELETE.....	100
1.7.1.	Excluindo todas as linhas de uma tabela	100
1.7.2.	Utilizando TOP em uma instrução DELETE	101
1.8.	OUTPUT para DELETE e UPDATE.....	102
1.9.	Transações	103
1.9.1.	Transações explícitas	104
Pontos principais	106	
Teste seus conhecimentos	107	
Mãos à obra!.....	113	
Aulas 14 a 19 - Consultando dados		
1.1.	Introdução	118
1.2.	SELECT	122
1.2.1.	Consultando todas as colunas	124
1.2.2.	Consultando colunas específicas	125
1.2.3.	Redefinindo os identificadores de coluna com uso de alias	126
1.3.	Ordenando dados.....	128
1.3.1.	Retornando linhas na ordem ascendente	128
1.3.2.	Retornando linhas na ordem descendente	128
1.3.3.	Ordenando por nome, alias ou posição	129
1.3.4.	ORDER BY com TOP	131
1.3.5.	ORDER BY com TOP WITH TIES.....	132
1.3.6.	Filtrando consultas.....	134
1.4.	Operadores relacionais	135
1.5.	Operadores lógicos	137
1.6.	Consultando intervalos com BETWEEN	138
1.7.	Consulta com base em caracteres.....	139
1.8.	Consultando valores pertencentes ou não a uma lista de elementos	142
1.9.	Lidando com valores nulos	142
1.10.	Substituindo valores nulos	144
1.10.1.	ISNULL	144
1.10.2.	COALESCE	145
1.11.	UNION	145
1.11.1.	Utilizando UNION ALL	146
1.12.	EXCEPT e INTERSECT	146
Pontos principais	150	
Teste seus conhecimentos	151	
Mãos à obra!.....	155	

Sumário

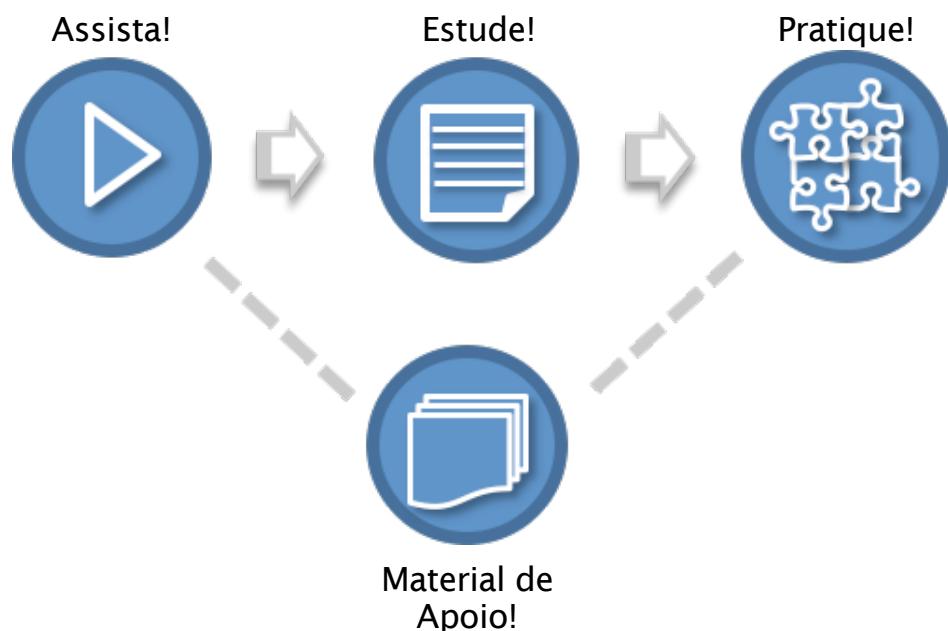
Aulas 20 a 22 - Associando tabelas	159
1.1. Introdução	160
1.2. INNER JOIN	160
1.3. OUTER JOIN	166
1.4. CROSS JOIN	169
Pontos principais	170
Teste seus conhecimentos	171
Mãos à obra!.....	175
 Aula 23 - Manipulando dados com junções	179
1.1. UPDATE com subqueries	180
1.2. DELETE com subqueries	180
1.3. UPDATE com JOIN.....	181
1.4. DELETE com JOIN.....	181
Pontos principais	182
Teste seus conhecimentos	183
Mãos à obra!.....	187
 Aulas 24 e 25 - Consultas com subqueries	191
1.1. Introdução	192
1.2. Principais características das subqueries	192
1.3. Subqueries introduzidas com IN e NOT IN	194
1.4. Subqueries introduzidas com sinal de igualdade (=)	195
1.5. Subqueries correlacionadas	196
1.5.1. Subqueries correlacionadas com EXISTS	196
1.6. Diferenças entre subqueries e associações	197
1.7. Diferenças entre subqueries e tabelas temporárias	198
Pontos principais	200
Teste seus conhecimentos	201
Mãos à obra!.....	205
 Aulas 26 e 27 - Agrupando dados	209
1.1. Introdução	210
1.2. Funções de agregação	210
1.2.1. Tipos de função de agregação	210
1.3. GROUP BY	212
1.3.1. Utilizando ALL	215
1.3.2. Utilizando HAVING	215
1.3.3. Utilizando WITH ROLLUP	216
1.3.4. Utilizando WITH CUBE.....	219
Pontos principais	221
Teste seus conhecimentos	223
Mãos à obra!.....	225
 Aulas 28 a 30 - Comandos adicionais.....	227
1.1. Funções de cadeia de caracteres	228
1.2. Função CASE	232
1.3. Manipulando campos do tipo datetime	232
1.4. Alterando a configuração de idioma a partir do SSMS	242
Pontos principais	246
Teste seus conhecimentos	247
Mãos à obra!.....	249

Apresentação

Bem-vindo!

É um prazer tê-lo como aluno do nosso curso online SQL 2016 - Criando Sistemas de Banco de Dados. Este curso é ideal para você que deseja aprender a criar e manipular um sistema de banco de dados em SQL Server. Durante o curso, você aprenderá a criar um banco de dados e utilizar diversos recursos do SQL Server e sua estrutura de consulta e manipulação de dados. Bom aprendizado!

Como estudar?



Videoaulas sobre os assuntos que você precisa saber neste curso de criação de sistemas de banco de dados com SQL Server.



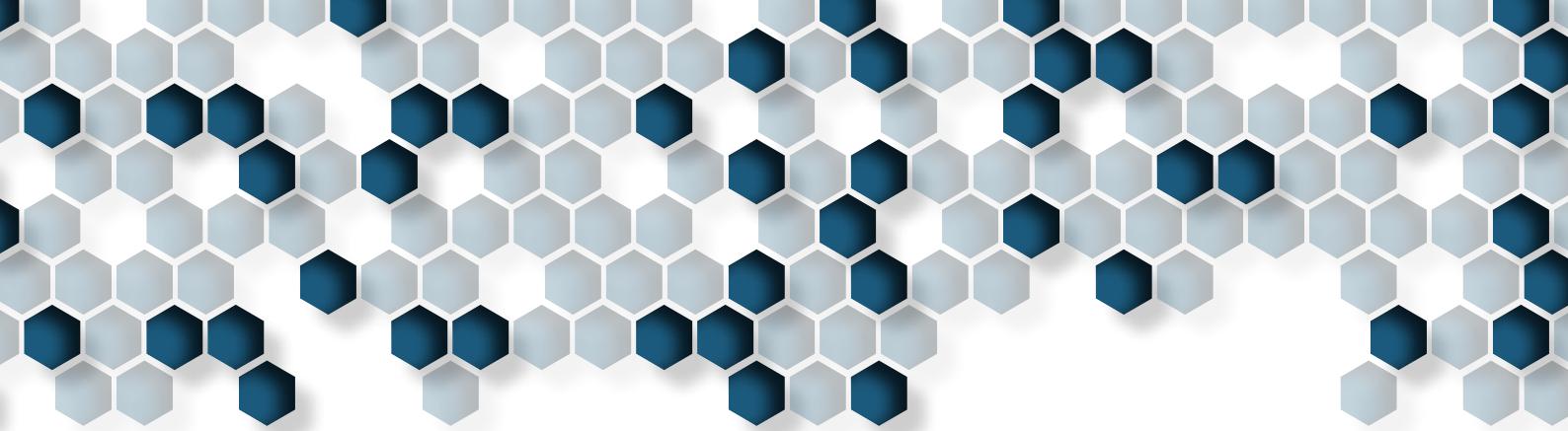
Parte teórica, com mais exemplos e detalhes para você que quer se aprofundar no assunto da videoaula.



Exercícios para você pôr à prova o que aprendeu.



Material de apoio para você testar os exemplos e exercícios das aulas.



Introdução ao SQL Server 2016

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

Esta Leitura Complementar refere-se ao conteúdo das Aulas 1 a 5.



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 empresa 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 de 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

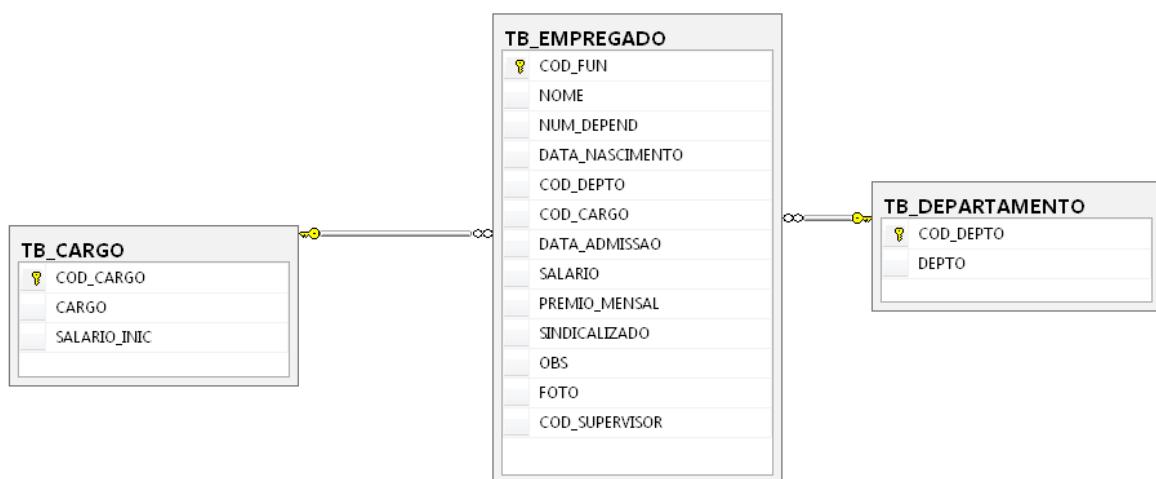
Esse modelo apresenta, em um formato de diagrama, as entidades e os atributos encontrados nos modelos anteriores. Nesse modelo, também conhecido como diagrama lógico de dados, ainda não é possível determinar qual banco de dados será utilizado.

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 2016.

Vejamos as definições:

- **Tabelas (entidades)**: Local de armazenamento das informações;
- **Campos (atributos)**: Características da tabela;
- **Chave primária**: Campo único que define a exclusividade da linha. Toda chave primária possui as seguintes características:
 - Valores únicos;
 - Não permite valores nulos;
 - A tabela será ordenada pela chave primária (índice clusterizado).
- **Relacionamento**: Relação entre tabelas através de um ou mais campos. As relações podem ser:
 - 1 para 1;
 - 1 para N;
 - N para N.

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



Esse modelo apresenta a modelagem das informações referentes aos empregados, departamentos e cargos.

1.2.5. Dicionário de dados

O dicionário de dados complementa o diagrama físico descrevendo as características da tabela.

Para entendermos a importância da descrição formal dos campos, imaginemos que existe um campo de status em uma tabela qualquer. Observe, adiante, alguns valores:

- 1 – Cancelado
- 2 – Em espera
- 3 – Encerrado
- 4 – Finalizado

Caso não seja documentada esta informação, será muito difícil a construção das consultas.

A seguir, um exemplo de dicionário de dados da tabela **TB_EMPREGADO**:

Sequência	Campo	Descrição	Tipo de Dados	NOT NULL	Identity	Bytes	PK	FK	Regras	Default
1	COD_FUN	Código do empregado	int		Sim	4	Sim			
2	NOME	Nome do empregado	varchar			35				
3	NUM_DEPEND	Nº de dependentes	tinyint	Sim		1				
4	DATA_NASCIMENTO	Data de nascimento	datetime			8				
5	COD_DEPTO	Código do departamento	int			4		Sim		
6	COD_CARGO	Código do cargo	int			4		Sim		
7	DATA ADMISSAO	Data de admissão	datetime			8				GETDATE()
8	SALARIO	Salário	decimal			9			Não permite valores negativos	
9	PREMIO_MENSAL	Prêmio mensal	decimal	Sim		9			Valores: S e N	
10	SINDICALIZADO	Campo para verificar se o empregado é sindicalizado	varchar			1				
10	OBS	Observações	varchar	Sim						
10	FOTO	Foto	varbinary	Sim						
10	COD_SUPERVISOR	Código do supervisor	int	Sim		4				

1.3. 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;
- Oferece 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.

1.3.1. Regras de normalização

A normalização inclui três 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.

Introdução ao SQL Server 2016

Aulas 1 a 5

Verifique a tabela **TB_ALUNO**, que atende as necessidades da área acadêmica de uma instituição de ensino:

TB_ALUNO	
COD_ALUNO	
NOME	
DATA_NASCIMENTO	
IDADE	
E_MAIL	
FONE_RES	
FONE_COM	
FONE_CELULAR	
FONE_RECADO	
PROFISSAO	
EMPRESA	

Ao executarmos o comando de criação, o SQL cria a tabela e é possível a inserção, alteração e exclusão de dados.

Vejamos, a seguir, 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 este formato:

CAMPO 1	CAMPO 2	CAMPO 3	CAMPO 4

E nunca poderá ter este 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 **TB_ALUNO**, que criamos anteriormente, precisa ser reestruturada para que respeite a primeira forma normal.

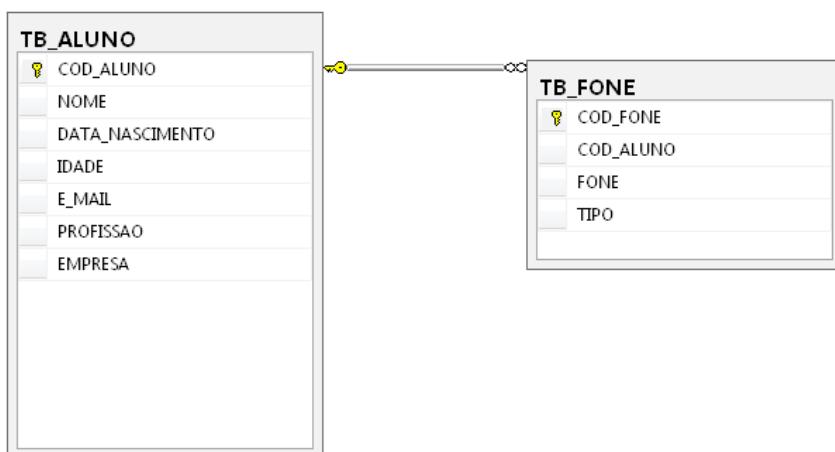
Verifique que existe um conjunto de dados repetidos: **FONE_RES**, **FONE_COML**, **FONE_CELULAR** e **FONE_RECADO**.

Numa necessidade de inclusão de um novo telefone, serão necessárias as seguintes alterações:

- Tabela;
- View;
- Procedure;
- Integrações entre banco e sistema;
- Alteração da aplicação.

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

Observe, a seguir, o modelo após a execução da primeira forma normal:



Opcionalmente, podemos eliminar o campo **COD_FONE** e utilizar os campos **NUM_ALUNO** e **FONE** como chave da tabela, impedindo que se cadastre o mesmo telefone mais de uma vez para o mesmo aluno.

- **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.

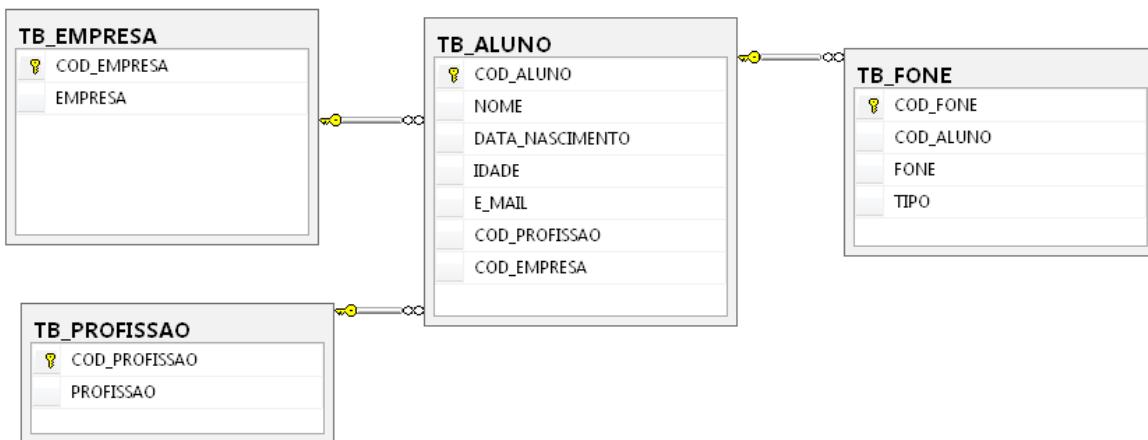
Com a finalidade de criar relacionamentos, devemos relacionar essas novas tabelas com uma chave estrangeira e identificar cada grupo de dados relacionados com uma chave primária.

Em outras palavras, a segunda forma normal pede que evitemos campos descritivos (alfanuméricicos) 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**.

A seguir, vejamos o modelo após a execução da segunda forma normal:



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

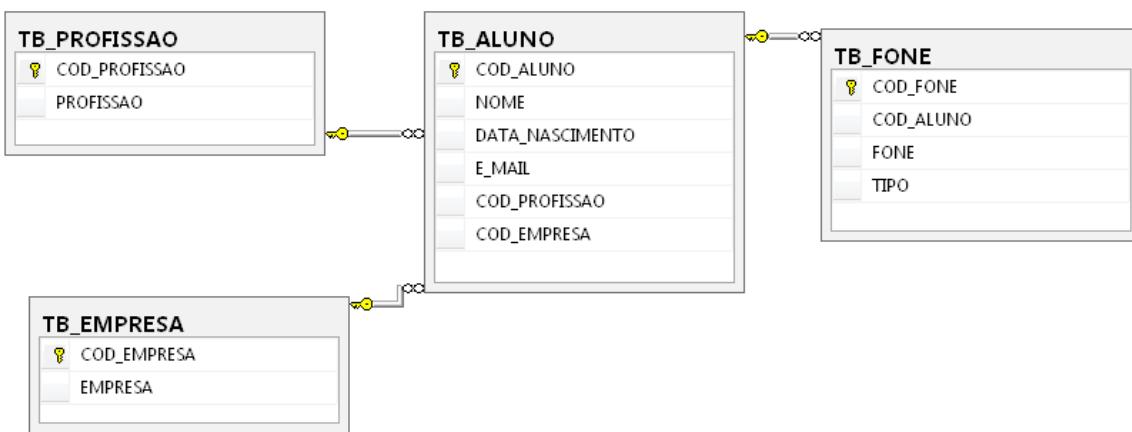
No terceiro nível de normalização, após ter concluído todas as tarefas do **1NF** e **2NF**, devemos eliminar os campos que não dependem de chaves primárias.

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 **TB_ALUNO**, não devemos ter o campo **IDADE**, pois ele não depende do número do aluno (chave primária), mas sim do campo **DATA_NASCIMENTO**.

A modelagem final após a aplicação da terceira forma normal é a seguinte:



1.4. Arquitetura cliente / servidor

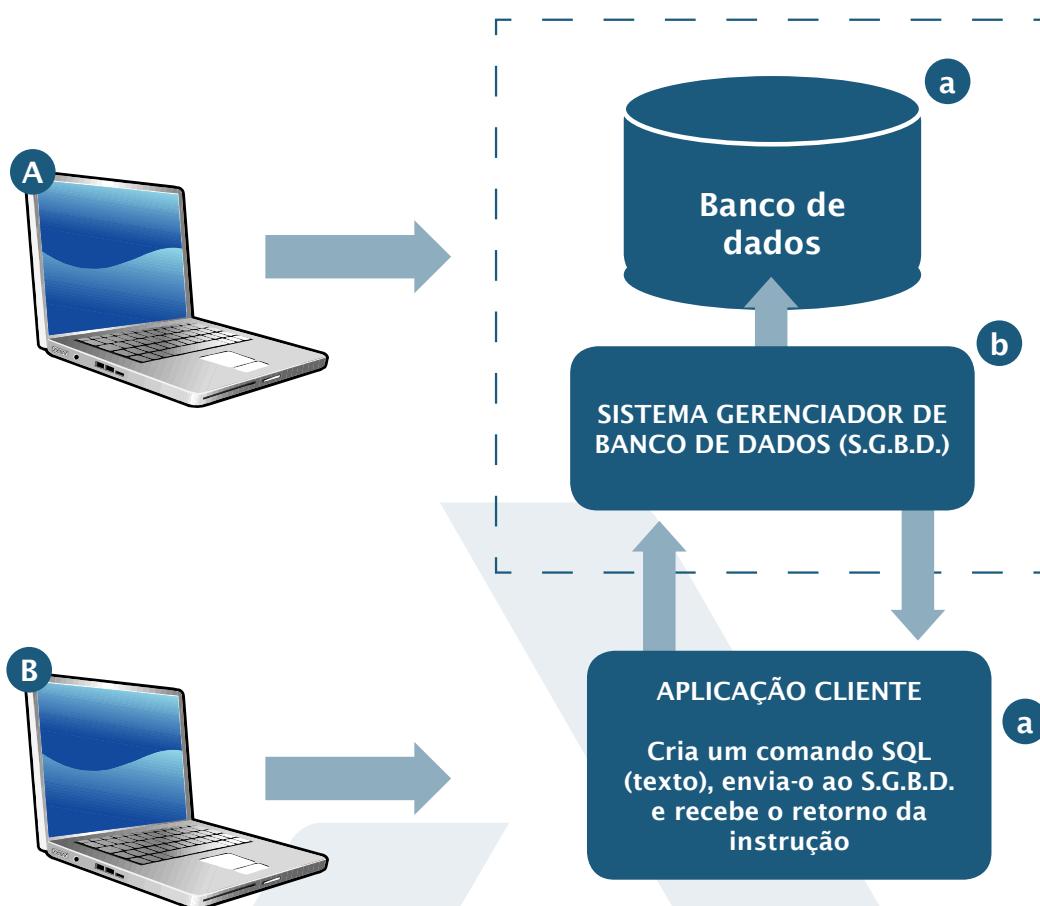
Essa arquitetura funciona da seguinte forma:

- 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.

Introdução ao SQL Server 2016

Aulas 1 a 5

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.

1.5. 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 (Structured 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 Standards 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:

Para realizar uma consulta na tabela de departamento (**TB_DEPARTAMENTO**) é utilizado o comando **SELECT**. Esse comando permite a visualização das informações contidas na referida tabela.

```
SELECT * FROM TB_DEPARTAMENTO
```

Ao executarmos o comando no SQL Server 2016, veremos o resultado conforme a ilustração adiante:

The screenshot shows the SQL Server Management Studio interface. In the top-left pane, there is a code editor with the following content:

```
1 --COMANDO DE CONSULTA SQL
2
3 select * FROM TB_DEPARTAMENTO
4
5
```

In the bottom-right pane, there is a results grid titled "Results". The grid displays the following data:

	COD_DEPTO	DEPTO
1	1	PESSOAL
2	2	T.I.
3	3	CONTROLE DE ESTOQUE
4	4	COMPRA
5	5	PRODUCAO
6	6	DIRETORIA
7	7	TELEMARKETING
8	8	FINANCIERO
9	9	RECURSOS HUMANOS
10	10	TREINAMENTO
		RECABRIMENTO

A callout bubble points to the results grid with the text: "Resultado da consulta da tabela de departamentos".

1.6. 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.

1.6.1. 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.

1.6.2. 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.

1.6.2.1. 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.

1.6.2.2. Índices

Quando realizamos uma consulta de dados, o SQL Server 2016 faz uso dos índices (**index**) para buscar, de forma fácil e rápida, informações específicas em uma tabela ou VIEW indexada.

1.6.2.3. 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**.

1.6.2.4. VIEW (Visualização)

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.

1.6.2.5. 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.

1.6.2.6. 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.

1.6.2.7. 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**.

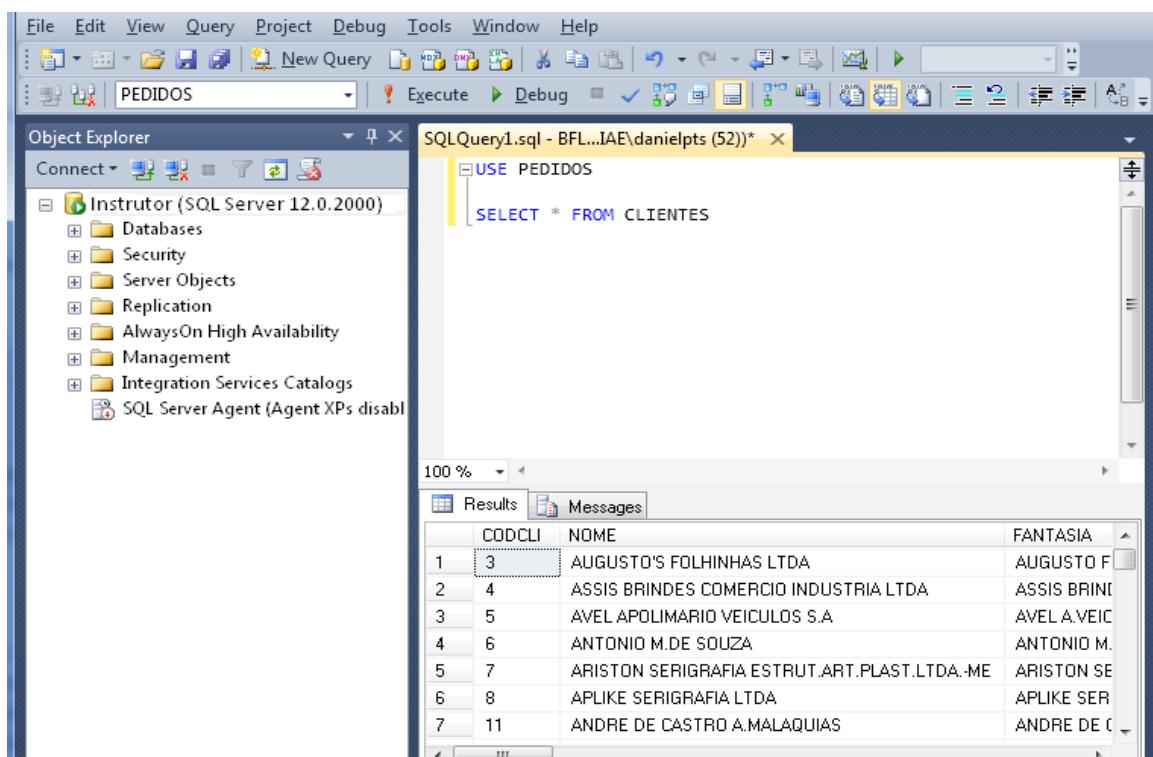
Quando atualizamos, inserimos ou excluímos dados em uma tabela, o TRIGGER automaticamente grava em uma tabela temporária os dados do registro atualizado, inserido ou excluído.

1.7. 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:

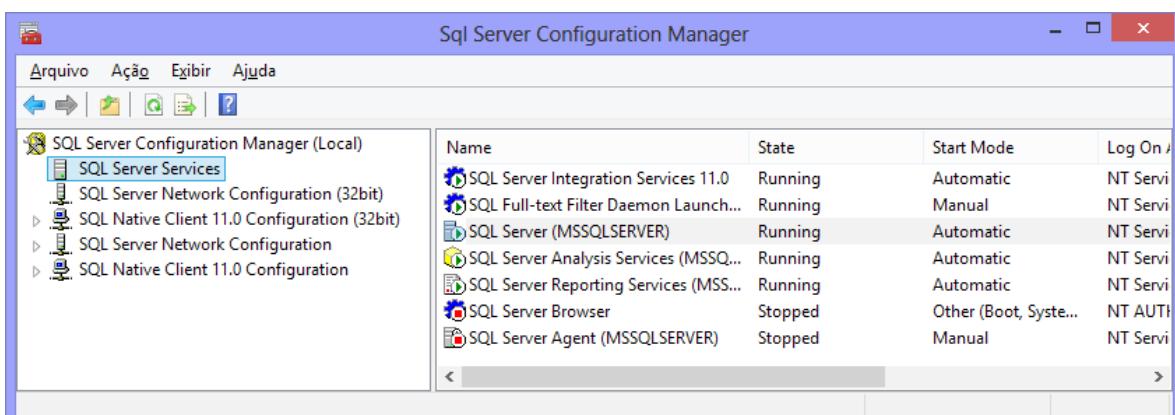
- **SQL Server Management Studio (SSMS)**

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**.

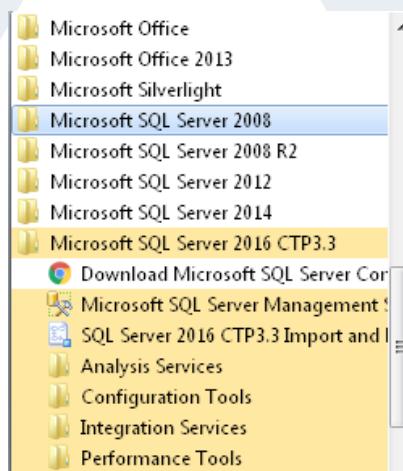
1.8. SQL Server Management Studio (SSMS)

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.

1.8.1. Inicializando o SSMS

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

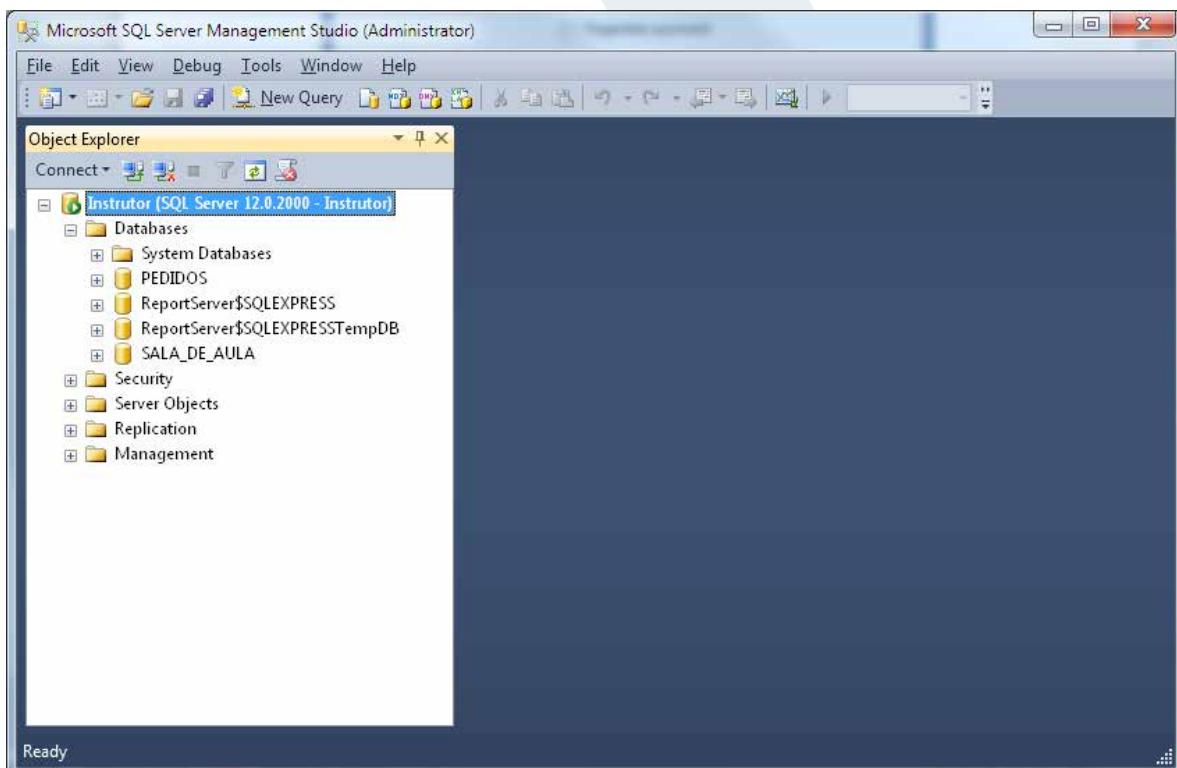
1. Clique no botão **Iniciar** e depois na opção **Todos os Programas**;
2. Clique em **Microsoft SQL Server 2016** e, em seguida, em **Microsoft SQL Server Management Studio**:



3. Na tela **Connect to Server**, escolha a opção **SQL Server 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:

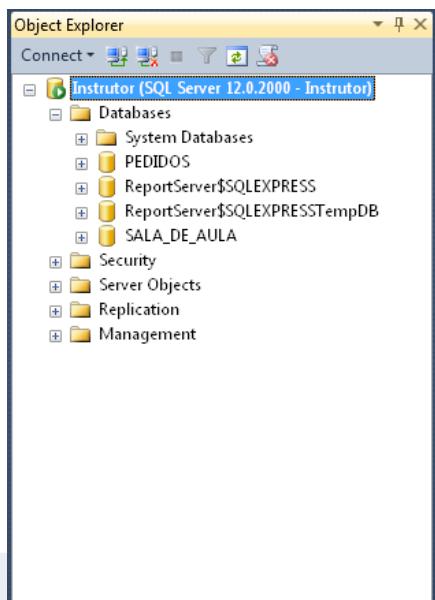


1.8.2. Interface

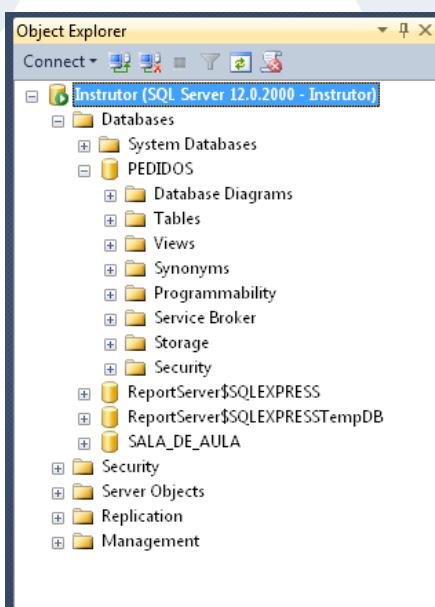
A interface do SSMS é composta pelo **Object Explorer** e pelo **Code Editor**, explicados a seguir:

- **Object Explorer**

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



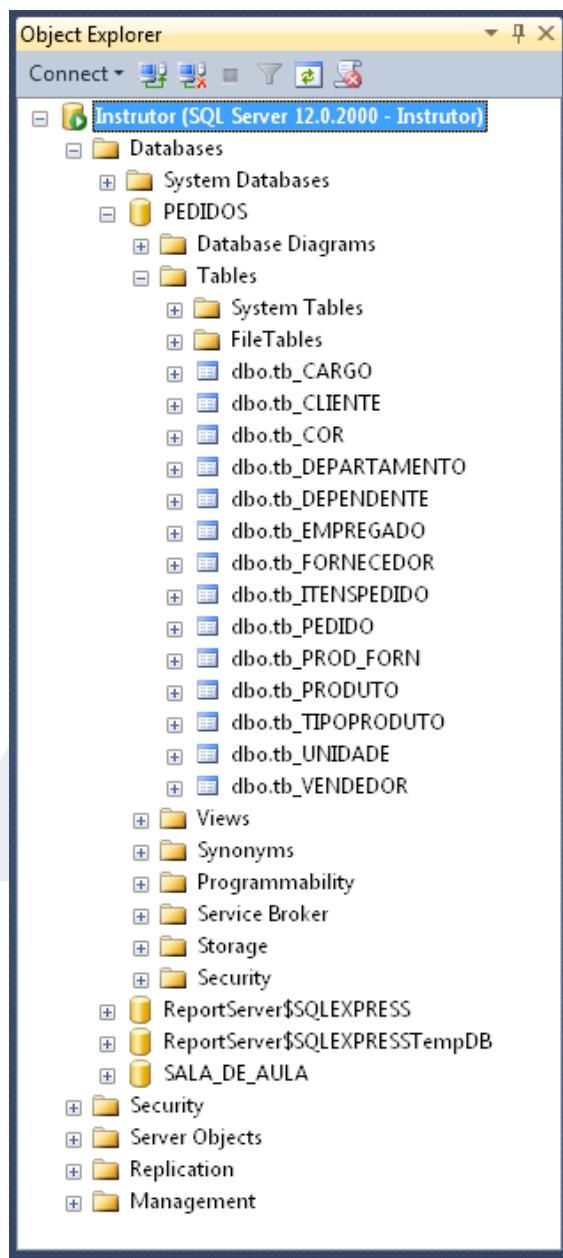
Observe que a pasta **Databases** mostra todos os bancos de dados existentes no servidor. No caso da imagem a seguir, **PEDIDOS** é um banco de dados:



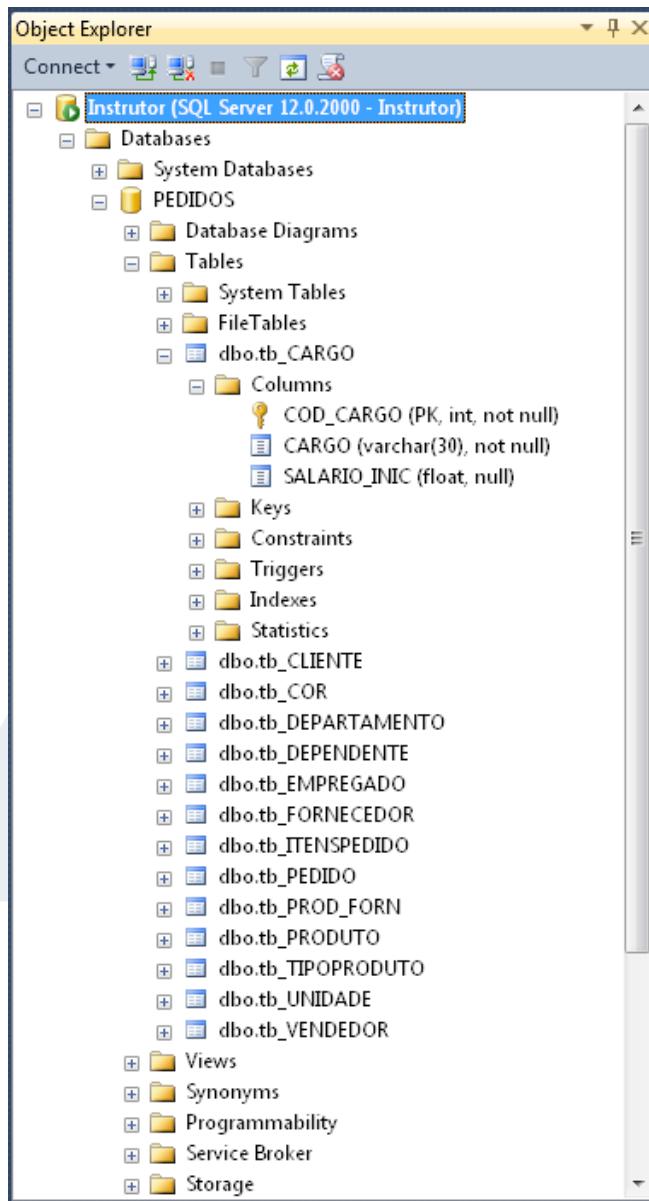
Introdução ao SQL Server 2016

Aulas 1 a 5

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

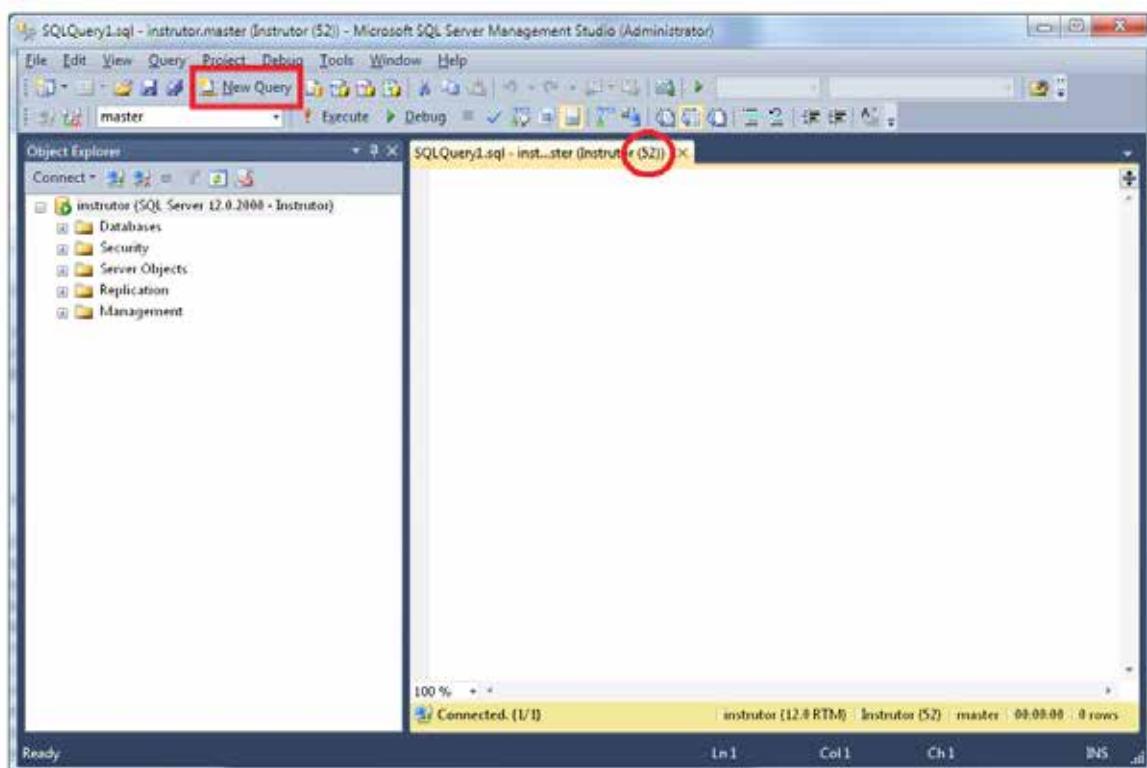


Expandindo uma das tabelas, veremos características da sua estrutura, bem como as suas colunas:



- **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 será criada:

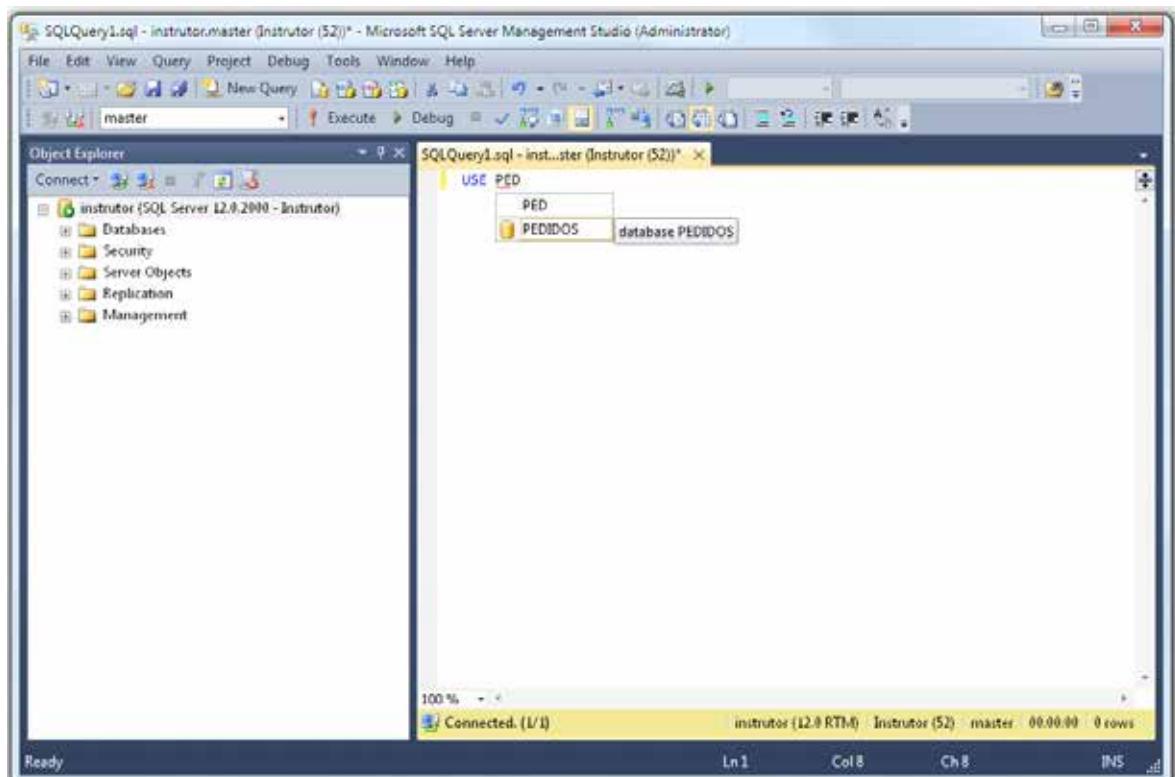


Cada uma dessas abas representa uma conexão com o banco de dados e recebe um **número de sessão**. Cada conexão tem um número de sessão único, mesmo que tenha sido aberta pelo mesmo usuário com o mesmo login e senha. Quando outra aplicação criada em outra linguagem, como Delphi, VB ou C#, abrir uma conexão,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

1.8.3. Executando um comando

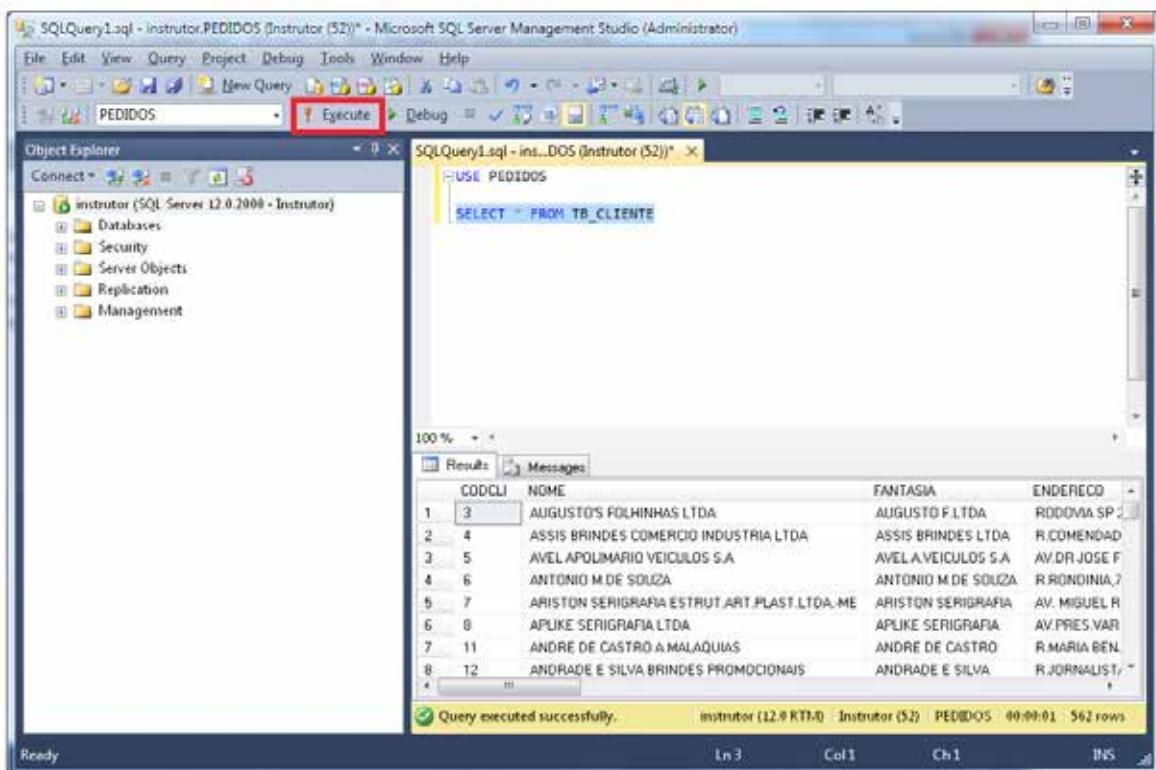
Para executar um comando a partir do SQL Server Management Studio, adote o seguinte procedimento:

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**.

1.8.4. Comentários

Todo script deve possuir comentários que ajudarão a entender e documentar as instruções. A cor do comentário é o verde e o SQL ignora o conteúdo do texto.

Para comentar uma linha utilize dois traços (--). Por exemplo:

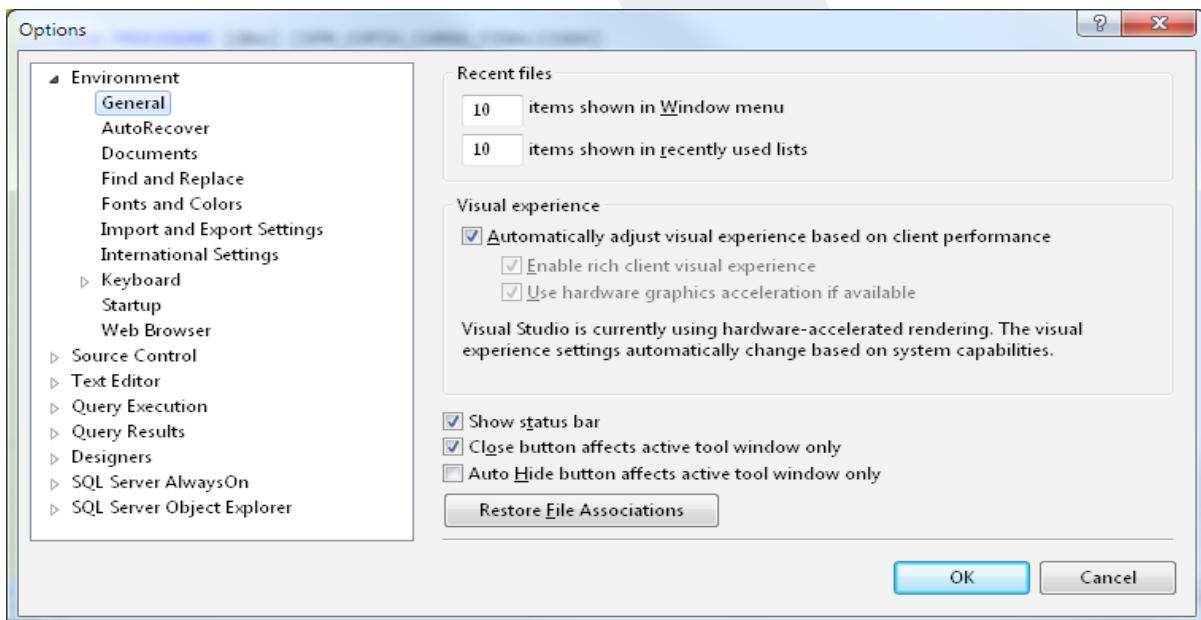
```
-- Comentário  
-- Primeira aula de SQL Server  
  
-- O comando a seguir executa uma consulta que retorna as informações da tabela  
de departamentos  
SELECT ...
```

Também é possível a criação de um bloco de comentários, utilizando /* e */:

```
/*  
Bloco de texto comentado que não é executado pelo SQL  
*/
```

1.8.5. Opções

Várias opções de customização do SSMS podem ser ajustadas conforme a preferência do usuário. Para isso, no menu **Tools**, clique em **Options**, abrindo a seguinte janela:



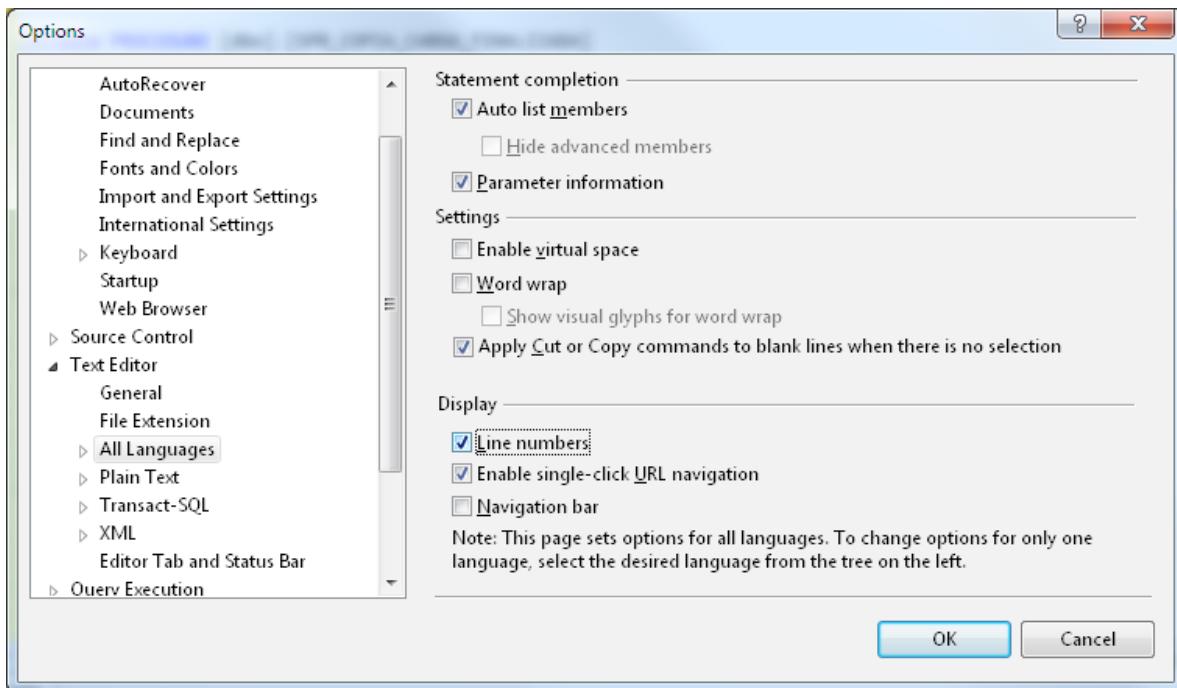
Vejamos, a seguir, alguns exemplos de customização:

Introdução ao SQL Server 2016

Aulas 1 a 5

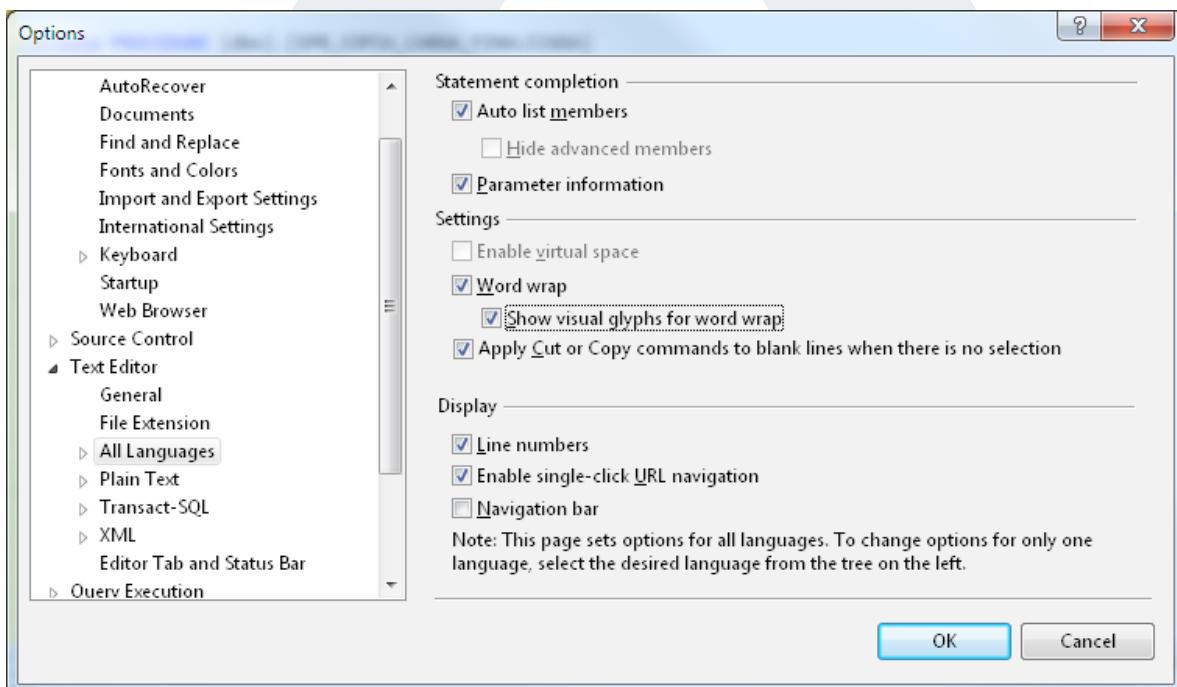
- Numeração automática da linha

Na janela Options, clique em Text Editor. Na guia All Languages, selecione Line numbers:



- Quebra de linha de automática

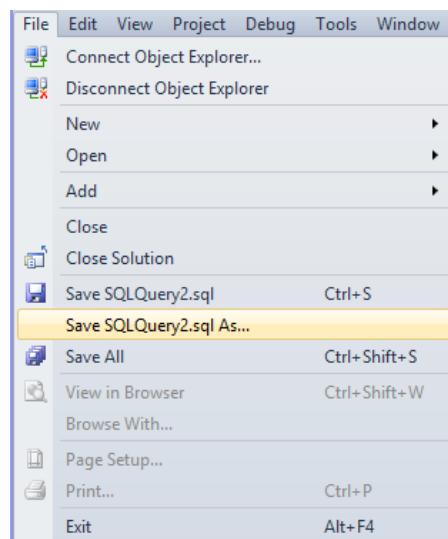
Na mesma guia, selecione Word wrap e Show visual glyphs for word wrap:



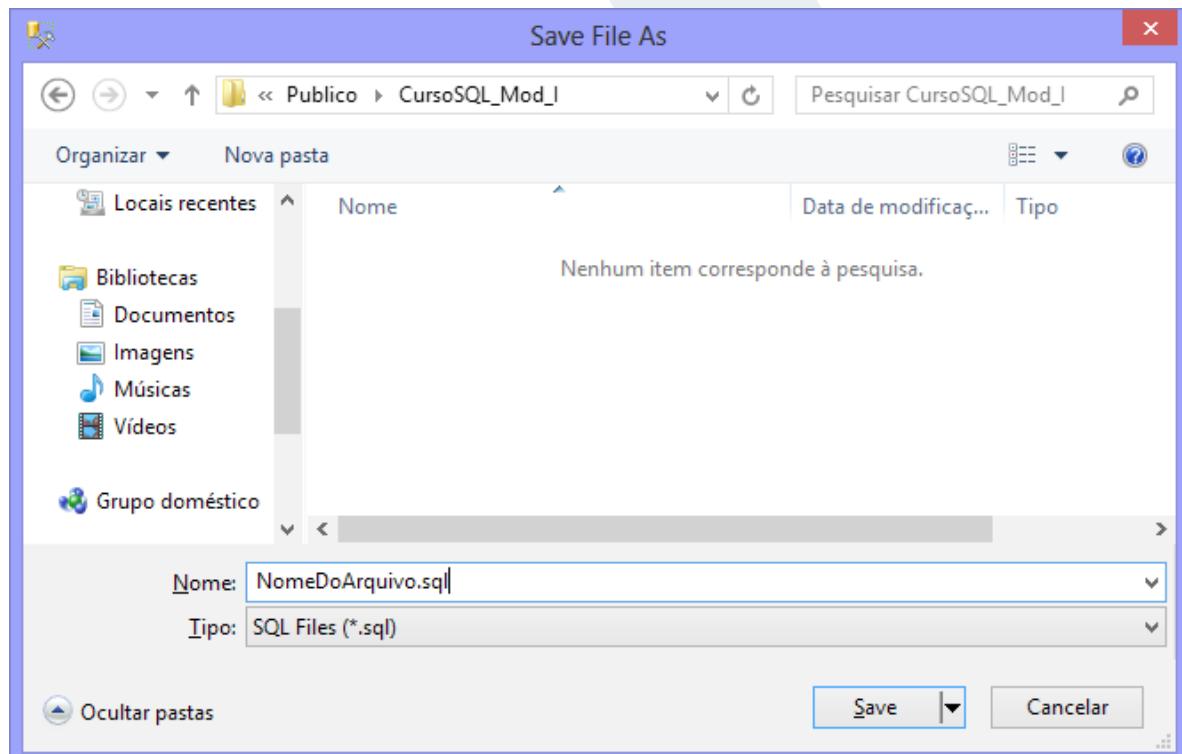
1.8.6. Salvando scripts

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**.

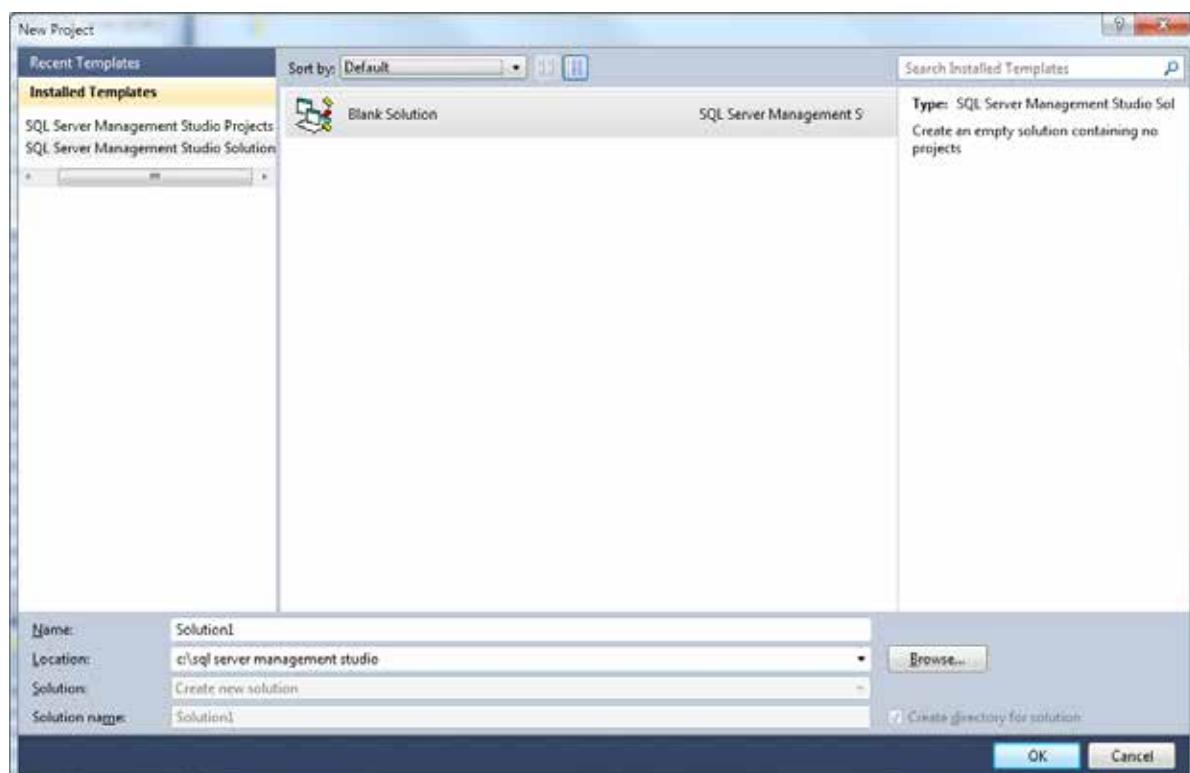
1.8.7. Soluções e Projetos

Uma solução é um conjunto de projetos, enquanto que um projeto é a coleção dos scripts. A organização é a grande vantagem deste tipo de recurso.

Vejamos, a seguir, como criar soluções e projetos:

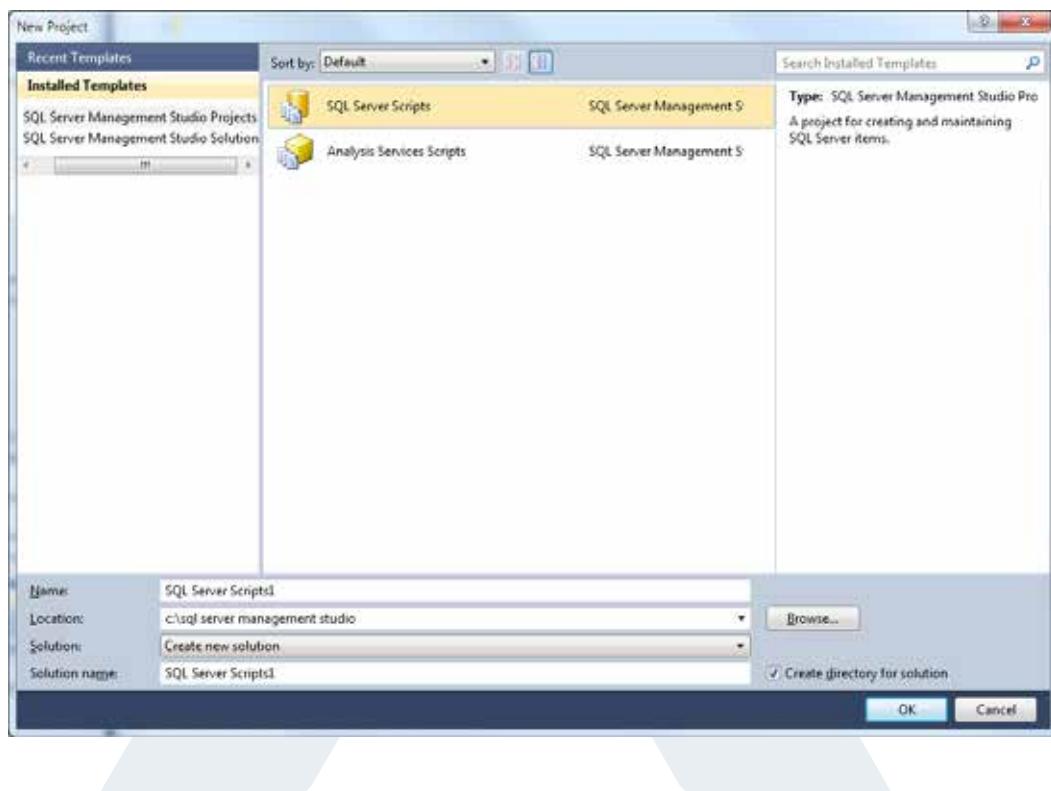
- **Criando uma solução**

No SSMS, clique no menu **File / New / Project** e, em seguida, selecione **SQL Server Management Studio Solution**:



- Criando projetos

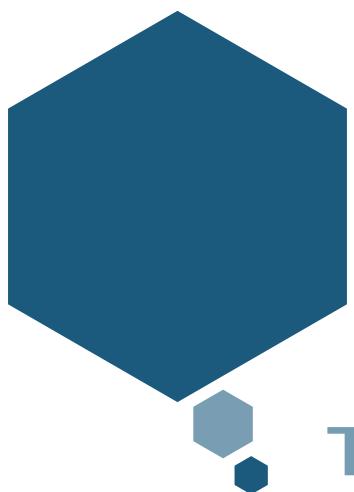
No SSMS, clique no menu **File / New / Project** e, em seguida, selecione **SQL Server Management Studio Projects**:



Pontos principais

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

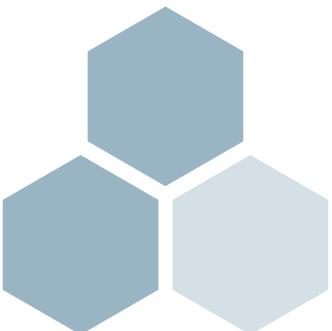
- Um banco de dados armazena informações e seu principal objeto são as tabelas;
- É fundamental o design de um banco de dados para que possua um bom desempenho;
- Os modelos de design de um banco de dados são: modelo descritivo, modelo conceitual, modelo lógico e modelo físico;
- Uma tabela precisa ter uma coluna que identifica de forma única cada uma de suas linhas. Essa coluna é chamada de chave primária;
- 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;
- 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, FUNCTION e TRIGGER;
- O SQL Server Management Studio (SSMS) é a principal ferramenta para gerenciamento de bancos de dados.



Introdução ao SQL Server 2016

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 1 a 5.



Editora
IMPACTA



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, qual das afirmações adiante está correta?

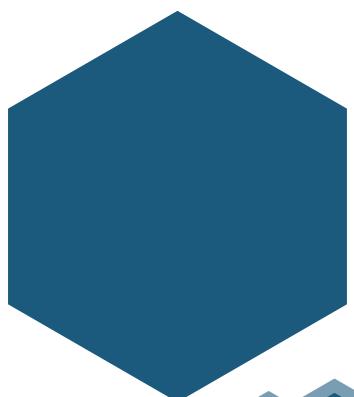
- a) A linguagem SQL foi desenvolvida pela IBM e não pode ser utilizada em outros bancos de dados.
- b) O SQL Server 2016 não utiliza a linguagem SQL.
- c) O SQL Server 2016 utiliza a linguagem T-SQL que é uma implementação da linguagem SQL.
- d) Nunca devemos utilizar a linguagem SQL, pois está ultrapassada.
- e) Nenhuma das alternativas anteriores está correta.

4. Com relação ao SQL Server 2016, qual das seguintes afirmações está correta?

- 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



Introdução ao SQL Server 2016

Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 1 a 5.



Laboratório 1

A – Construindo artefatos de modelagem de dados

O objetivo deste laboratório é a construção básica dos artefatos de modelagem de dados, permitindo o melhor conhecimento das técnicas para a construção de um banco de dados.

Observe, adiante, um modelo de formulário de cadastro de clientes de uma empresa:

Cadastro de Clientes

ID		Cidade	
Nome		Estado	
Data do Cadastro		CEP	
Dt de Nascimento		Limite de Compra	
Idade			
Profissao			
Telefone Residencial			
Telefone Comercial			
Celular			
Endereço			
Bairro			

Realize os passos a seguir:

1. Crie as tabelas e campos necessários para atender esse cadastro;
2. Desenhe o modelo lógico das tabelas do cadastro de clientes;
3. Descreva o dicionário de dados do modelo lógico:

Sequência	Campo	Descrição	Tipo de Dados	NOT NULL	Identity	Bytes	PK	FK	Regras	Default
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										

4. Utilize o recurso de normalização;
5. Descreva quais formas normais foram utilizadas.

Laboratório 2

A – Criando soluções, projetos e scripts

Neste laboratório, serão utilizados os recursos do SQL Server Management Studio (SSMS), para a criação de uma solução, projeto e scripts. Para isso, siga os seguintes passos:

1. Abra o Windows Explorer e crie uma pasta de nome **C:\SQL\Soluções**;
2. Abra o SQL Server Management Studio (SSMS);
3. Clique no menu **File / New / Project** e crie a seguinte solução:
 - Nome: **Curso Impacta – Módulo I**;
 - Localização: **C:\SQL\Soluções**.
4. Clique novamente no menu **File / New / Project** e crie o seguinte projeto:
 - Nome: **Projeto Capítulo 1**;
 - Localização: **C:\SQL\Soluções**;
 - Solution Name: **Curso Impacta – Módulo I**.
5. Clique no botão **New Query**;
6. Crie um comentário utilizando traço (-);
7. Crie um comentário utilizando o bloco de comentários /* */;
8. Digite o comando adiante:

```
SELECT GETDATE()
```
9. Execute o comando com F5 ou através do botão **Execute**;
10. Salve o script.



Criando um banco de dados

- ◆ CREATE DATABASE;
- ◆ CREATE TABLE;
- ◆ Tipos de dados;
- ◆ Campo de autonumeração (IDENTITY);
- ◆ Constraints.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 6 e 7.



1.1. Introdução

Nesta leitura, veremos os recursos iniciais para criação de banco de dados: os comandos **CREATE DATABASE** e **CREATE TABLE**, os tipos de dados e as constraints.

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.

1.2. 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.

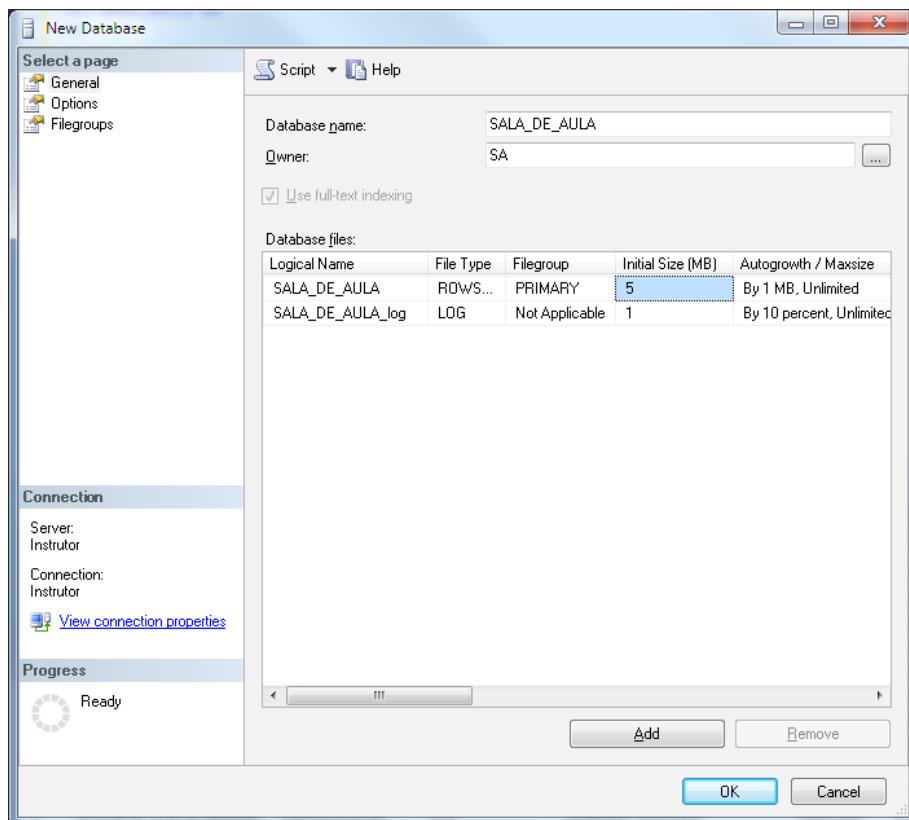
Normalmente, esses arquivos estão localizados na pasta **DATA** dentro do diretório de instalação do SQL Server.

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

Criando um banco de dados

Aulas 6 e 7

Também é possível criar um banco de dados graficamente. Através do SSMS, sobre a conexão, clique com o botão direito em **New Database**.



É importante que seja definida a localização dos arquivos que compõem o banco para o seu melhor gerenciamento.

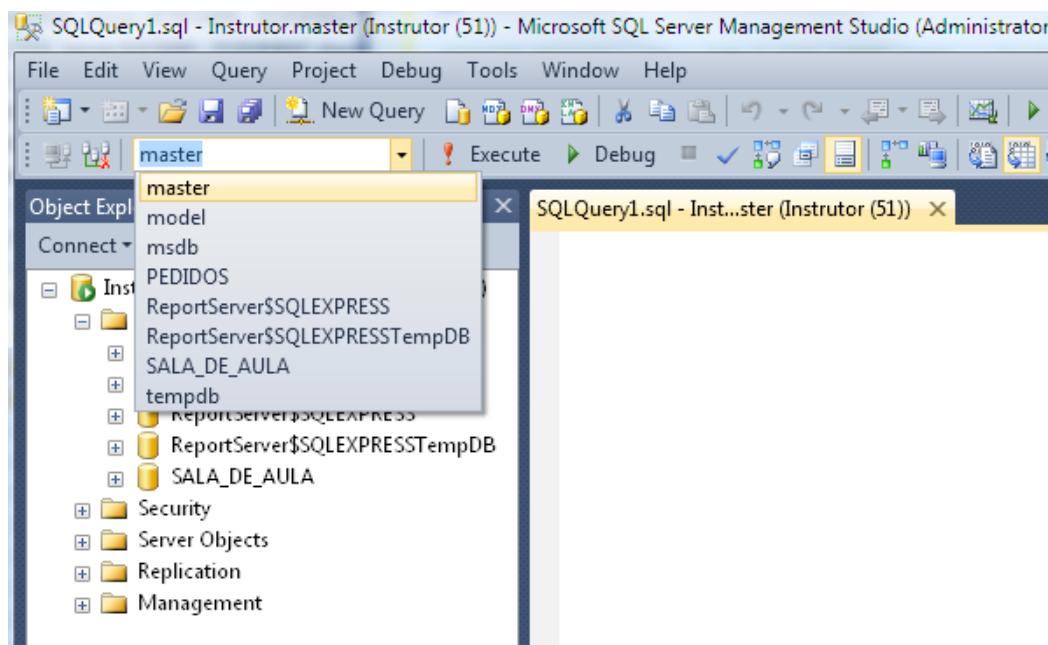
Para facilitar o acesso a um banco de dados, devemos colocá-lo em uso, mas isso não é obrigatório. Para colocar um banco de dados em uso, utilize o seguinte código:

```
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.



1.3. 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.

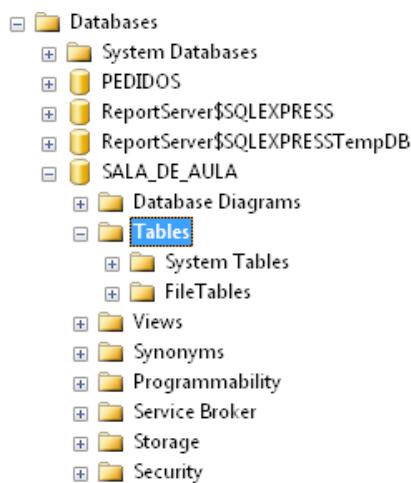
A seguir, veja um exemplo de como criar uma tabela em um banco de dados:

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

 A estrutura dessa tabela não respeita as regras de normalização de dados.

O SQL disponibiliza a criação da tabela de forma gráfica. Vejamos:

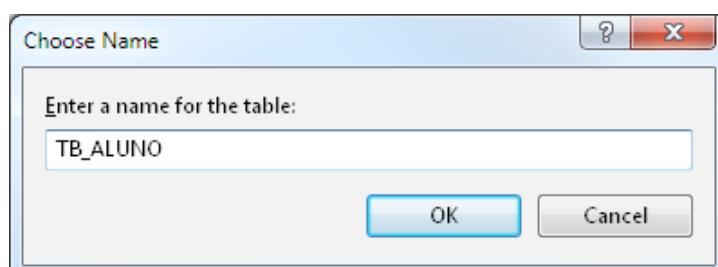
1. Expanda **Databases**;
2. Expanda o banco em que você deseja criar a tabela;
3. Clique com o botão direito em **Tables**;



4. Informe os nomes dos campos;

Column Name	Data Type	Allow Nulls
Num_ALUNO	int	<input type="checkbox"/>
NOME	varchar(30)	<input checked="" type="checkbox"/>
Data_Nascimento	datetime	<input checked="" type="checkbox"/>
IDade	tinyint	<input checked="" type="checkbox"/>
E_mail	varchar(50)	<input checked="" type="checkbox"/>
Fone_Res	char(8)	<input checked="" type="checkbox"/>
Fone_Coml	char(8)	<input checked="" type="checkbox"/>
FAX	char(8)	<input checked="" type="checkbox"/>
Celular	char(8)	<input checked="" type="checkbox"/>
Profissao	varchar(50)	<input checked="" type="checkbox"/>
Empresa	varchar(50)	<input checked="" type="checkbox"/>

5. No momento de salvar, informe o nome da tabela.



1.4. 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.

1.4.1. Numéricos exatos

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

- **Inteiros**

Nome	Descrição
bigint 8 bytes	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).
int 4 bytes	Valor de número inteiro compreendido entre -2^{31} (-2,147,483,648) e $2^{31}-1$ (2,147,483,647).
smallint 2 bytes	Valor de número inteiro compreendido entre -2^{15} (-32,768) e $2^{15}-1$ (32,767).
tinyint 1 byte	Valor de número inteiro de 0 a 255.

- **Bit**

Nome	Descrição
bit 1 byte	Valor de número inteiro com o valor 1 ou o valor 0.

- **Numéricos exatos**

Nome	Descrição
decimal(<T>,<D>)	Valor numérico de precisão e escala fixas de $-10^{38}+1$ até $10^{38}-1$.
numeric(<T>,<D>)	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
money 8 bytes	Compreende valores monetários ou de moeda corrente entre -922.337.203.685.477,5808 e 922.337.203.685.477,5807.
smallmoney 4 bytes	Compreende valores monetários ou de moeda corrente entre -214,748.3648 e +214,748.3647.

1.4.2. Numéricos aproximados

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

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

 Esses tipos são chamados de "Numéricos aproximados" porque podem gerar imprecisão na parte decimal.

1.4.3. Data e hora

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

Nome	Descrição
datetime 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.
smalldatetime 4 bytes	Data e hora compreendidas entre 1º de janeiro de 1900 e 6 de junho de 2079, com a exatidão de 1 minuto.
datetime2[(p)] 8 bytes	Data e hora compreendidas entre 01/01/0001 e 31/12/9999 com precisão de até 100 nanossegundos, dependendo do valor de p , que representa a quantidade de algarismos na fração de segundo. Omitindo p , o valor default será 7.
date 3 bytes	Data compreendida entre 01/01/0001 e 31/12/9999, com precisão de 1 dia.
time[(p)] 5 bytes	Hora no intervalo de 00:00:00.0000000 a 23.59.59.999999. O parâmetro p indica a quantidade de dígitos na fração de segundo.
Datetimeoffset[(p)]	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 p indica a quantidade de dígitos na fração de segundo.

1.4.4. 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
char(<n>)	Comprimento fixo de no máximo 8.000 caracteres no padrão ANSI. Cada caractere é armazenado em 1 byte.
varchar(<n>)	Comprimento variável de no máximo 8.000 caracteres no padrão ANSI. Cada caractere é armazenado em 1 byte.
text ou varchar(max)	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.

1.4.5. 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
nchar(<n>)	Comprimento fixo de no máximo 4.000 caracteres Unicode.
nvarchar(<n>)	Comprimento variável de no máximo 4.000 caracteres Unicode.
ntext ou nvarchar(max)	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.

1.4.6.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
binary(<n>)	Dado binário com comprimento fixo de, no máximo, 8.000 bytes.
varbinary(<n>)	Dado binário com comprimento variável de, no máximo, 8.000 bytes.
image ou varbinary(max)	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.

1.4.7.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
table	Serve para definir um dado tabular, composto de linhas e colunas, assim como uma tabela.
cursor	Serve para percorrer as linhas de um dado tabular.
sql_variant	Um tipo de dado que armazena valores de vários tipos suportados pelo SQL Server, exceto os seguintes: text , ntext , timestamp e sql_variant .
Timestamp ou RowVersion	Número hexadecimal sequencial gerado automaticamente.
uniqueidentifier	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.

Nome	Descrição
XML	Armazena dados no formato XML.
Hierarchyid	Posição de uma hierarquia.
Geography	Representa dados de coordenadas terrestres.
Geometry	Representação de coordenadas euclidianas.

1.5. 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 TB_ALUNO;          --Caso a tabela já exista
GO
CREATE TABLE TB_ALUNO
(
    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) );

```

1.6. 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**.

1.6.1. 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 TB_ALUNO
(
    CODIGO      INT      NOT NULL,
    NOME        VARCHAR(30) NOT NULL,
    E_MAIL      VARCHAR(100) NULL
);
```

1.6.2. 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.

1.6.2.1. 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 pela 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
Tabela PRODUTOS

Código do Cliente
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.

1.6.2.2. 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
Na tabela TIPOS_PRODUTO
Na tabela UNIDADES_MEDIDA

Campos CPF, RG e E-mail
Descrição do tipo de produto
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**.

1.6.2.3. 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
Tabela PRODUTOS	Preço Venda >= 0 Preço Compra >= 0 Preço Venda >= Preço Compra Data Inclusão <= Data Atual

- **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**.

1.6.2.4. 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 a ela quando o **INSERT** não fornecer valor.

- **Exemplo**

Tabela PESSOAS	Data Inclusão DEFAULT Data Atual Sexo DEFAULT '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
```

1.6.2.5. 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**.

The diagram illustrates the relationship between two tables: **TB_DEPARTAMENTO** (MESTRE) and **TB_EMPREGADO** (DETALHE). The **TB_DEPARTAMENTO** table has columns **COD_DEPTO** and **DEPTO**, with rows numbered 1 to 14. The row with **COD_DEPTO** 5, labeled **PRODUCAO**, is highlighted with a red box. The **TB_EMPREGADO** table has columns **CODFUN**, **NOME**, **NUM_DEPE...**, **DATA_NASCIMENTO**, and **COD_DEP...**, with rows numbered 1 to 12. The row with **CODFUN** 5, **NOME** **JOAO LIMA MACHADO DA SILVA**, and **COD_DEP...** 5 is highlighted with a red box. Arrows point from the primary key value 5 in the **TB_DEPARTAMENTO** table to the corresponding foreign key value 5 in the **TB_EMPREGADO** table.

	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
9	9	RECURSOS HUMANOS
10	10	TREINAMENTO
11	11	PRESIDENCIA
12	12	PORTARIA
13	13	CONTROLADORIA
14	14	P.C.P.

	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

Observando a figura, notamos que o campo **COD_DEPTO** da tabela **TB_EMPREGADO** (detalhe) é chave estrangeira, pois se relaciona com **COD_DEPTO** (chave primária) da tabela **TB_DEPARTAMENTO** (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 **TB_EMPREGADO** com um **COD_DEPTO** inexistente em **TB_DEPARTAMENTO**. Quando procurarmos o **COD_DEPTO** do empregado em **TB_DEPARTAMENTO**, 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)
```

1.6.3.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.

1.6.3.1.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 **TB_TIPO_PRODUTO** com os tipos (categorias) de produto:

```
-- Tabela de tipos (categorias) de produto
CREATE TABLE TB_TIPO_PRODUTO
(
    COD TIPO          INT IDENTITY NOT NULL,
    TIPO          VARCHAR(30) NOT NULL,
    -- Convenção de nome: PK_NomeTabela
    CONSTRAINT PK_TB_TIPO_PRODUTO PRIMARY KEY (COD TIPO),
    -- Convenção De nome: UQ_NomeTabela_NomeCampo
    CONSTRAINT UQ_TB_TIPO_PRODUTO_TIPO UNIQUE( TIPO ) );
```

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

```
-- Testando a constraint UNIQUE
INSERT TB_TIPO_PRODUTO VALUES ('MOUSE');
INSERT TB_TIPO_PRODUTO VALUES ('PEN-DRIVE');
INSERT TB_TIPO_PRODUTO VALUES ('HARD DISK');
-- Ao tentar inserir, o SQL gera um erro de violação de
-- constraint UNIQUE
INSERT TB_TIPO_PRODUTO VALUES ('HARD DISK');
```

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

```
-- Tabela de TB_PRODUTO
CREATE TABLE TB_PRODUTO
(
    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_TB_PRODUTO PRIMARY KEY( ID_PRODUTO ),
    CONSTRAINT UQ_TB_PRODUTO_DESCRICAO UNIQUE( DESCRICAO ),

    CONSTRAINT CK_TB_PRODUTO_PRECOS
        CHECK( PRECO_VENDA >= PRECO_CUSTO ),
    CONSTRAINT CK_TB_PRODUTO_DATA_CAD
        CHECK( DATA_CADASTRO <= GETDATE() ),
    CONSTRAINT CK_TB_PRODUTO_SN_ATIVO
        CHECK( SN_ATIVO IN ('N','S') ),
    -- Convenção de nome: FK_TabelaDetalhe_TabelaMestre
    CONSTRAINT FK_TB_PRODUTO_TIPO_PRODUTO
        FOREIGN KEY (COD TIPO)
        REFERENCES TB_TIPO_PRODUTO (COD TIPO) );
```

Criando um banco de dados

Aulas 6 e 7

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 )
```

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

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

```
-- Gera erro, pois viola a constraint UNIQUE  
INSERT TB_PRODUTO  
(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**:

```
-- Gera um erro, pois viola a constraint FOREIGN KEY  
INSERT TB_PRODUTO  
(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**:

```
-- Gera um erro, pois viola a constraint CHECK -  
-- (CK_PRODUTO_PRECOS)  
INSERT TB_PRODUTO  
(DESCRICAO, COD_TIPO, PRECO_CUSTO, PRECO_VENDA,  
 QTD_REAL, QTD_MINIMA)  
VALUES ('TESTANDO INCLUSAO 2', 1, 14, 12, 10, 5 );
```

1.6.3.2. Criando constraints com ALTER TABLE

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

```
USE TESTE_CONSTRAINT;

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

ALTER TABLE TB_TIPO_PRODUTO ADD
    CONSTRAINT UQ_TB_TIPO_PRODUTO_TIPO UNIQUE( TIPO );

-- Criando a tabela PRODUTOS
CREATE TABLE TB_PRODUTO
(
    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 TB_PRODUTO ADD
    CONSTRAINT PK_TB_PRODUTO PRIMARY KEY( ID_PRODUTO );

ALTER TABLE TB_PRODUTO ADD
    CONSTRAINT UQ_TB_PRODUTO_DESCRICAO UNIQUE( DESCRICAO );

-- Criando várias constraints em um único ALTER TABLE
ALTER TABLE TB_PRODUTO ADD
    CONSTRAINT CK_TB_PRODUTO_PRECOS
        CHECK( PRECO_VENDA >= PRECO_CUSTO ),
    CONSTRAINT CK_TB_PRODUTO_DATA_CAD
        CHECK( DATA_CADASTRO <= GETDATE() ),
    CONSTRAINT CK_TB_PRODUTO_SN_ATIVO
        CHECK( SN_ATIVO IN ('N','S') ),
    CONSTRAINT FK_TB_PRODUTO_TIPO_PRODUTO
        FOREIGN KEY (COD_TIPO)
            REFERENCES TB_TIPO_PRODUTO (COD_TIPO),
    CONSTRAINT DF_TB_PRODUTO_SN_ATIVO DEFAULT ('S') FOR SN_ATIVO,
    CONSTRAINT DF_TB_PRODUTO_DATA_CADASTRO DEFAULT (GETDATE())
        FOR DATA_CADASTRO;
```

1.6.3.3. 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;

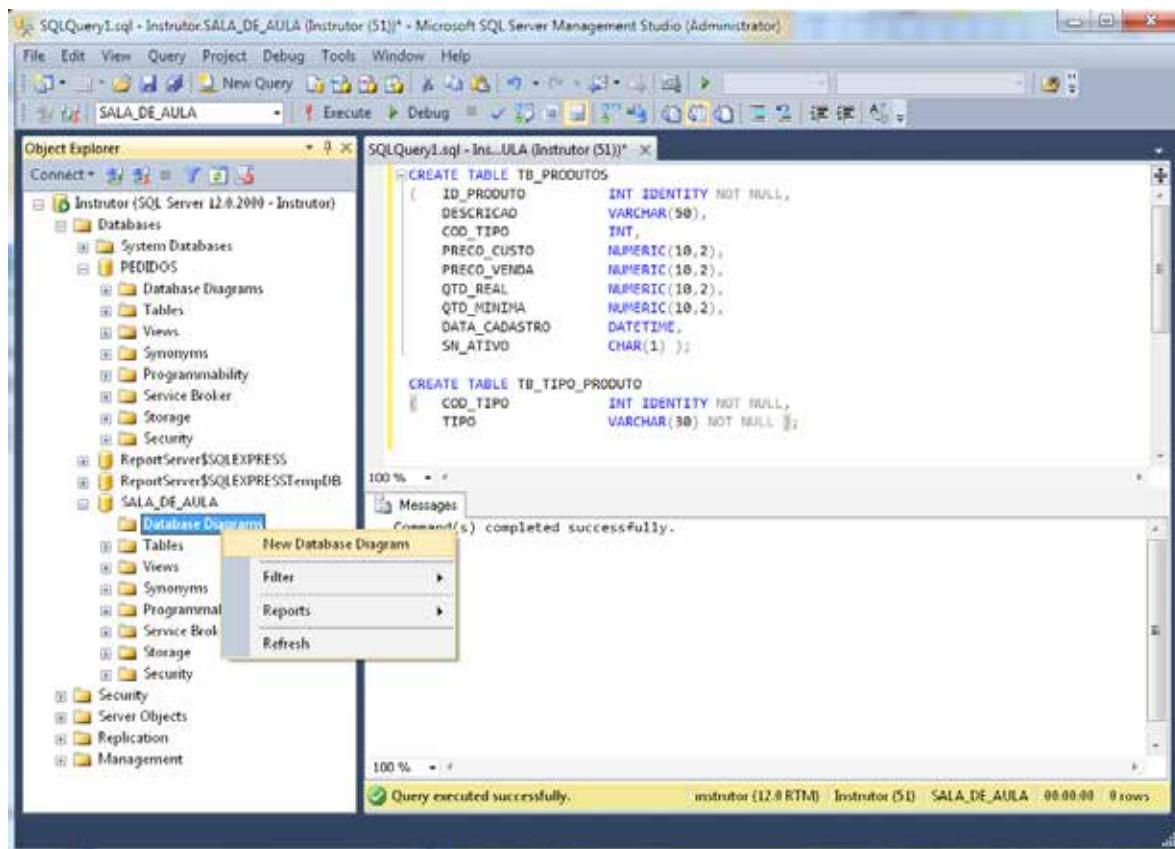
DROP TABLE TB_PRODUTO
GO
DROP TABLE TB_TIPO_PRODUTO
GO
-- Criação da tabela TIPO_PRODUTO
CREATE TABLE TB_TIPO_PRODUTO
(
    COD_TIPO           INT IDENTITY NOT NULL,
    TIPO               VARCHAR(30) NOT NULL );

-- Criando a tabela PRODUTO
CREATE TABLE TB_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) );
```

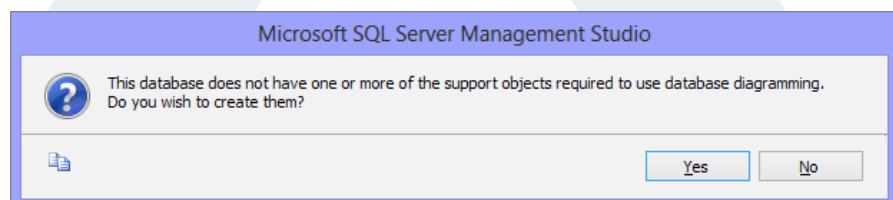
Depois, realize os seguintes passos:

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**:



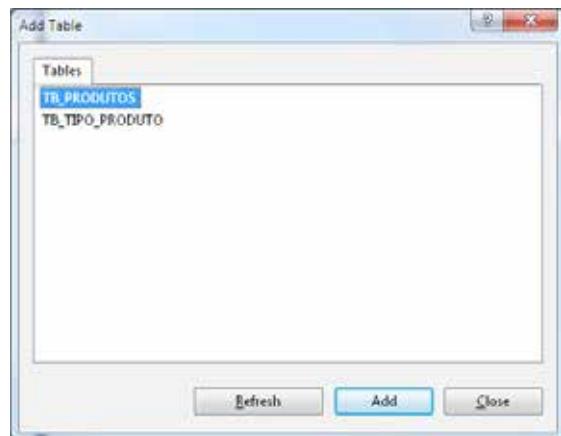
3. Na caixa de diálogo exibida, clique em **Yes (Sim)**;



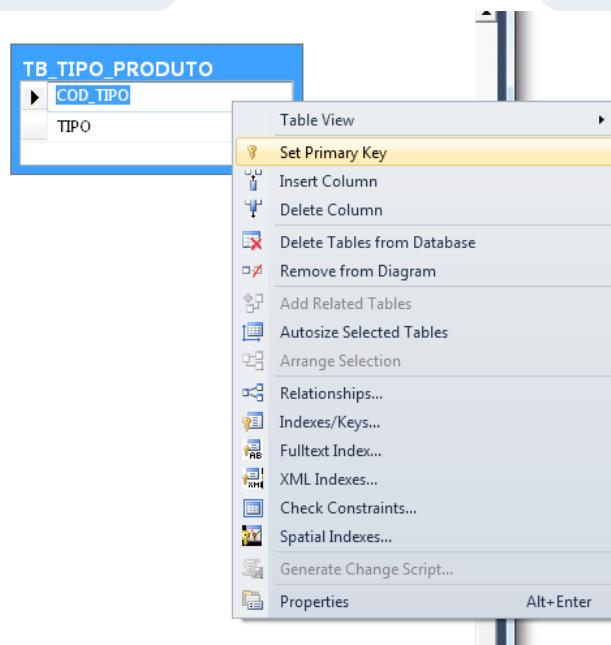
Criando um banco de dados

Aulas 6 e 7

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;



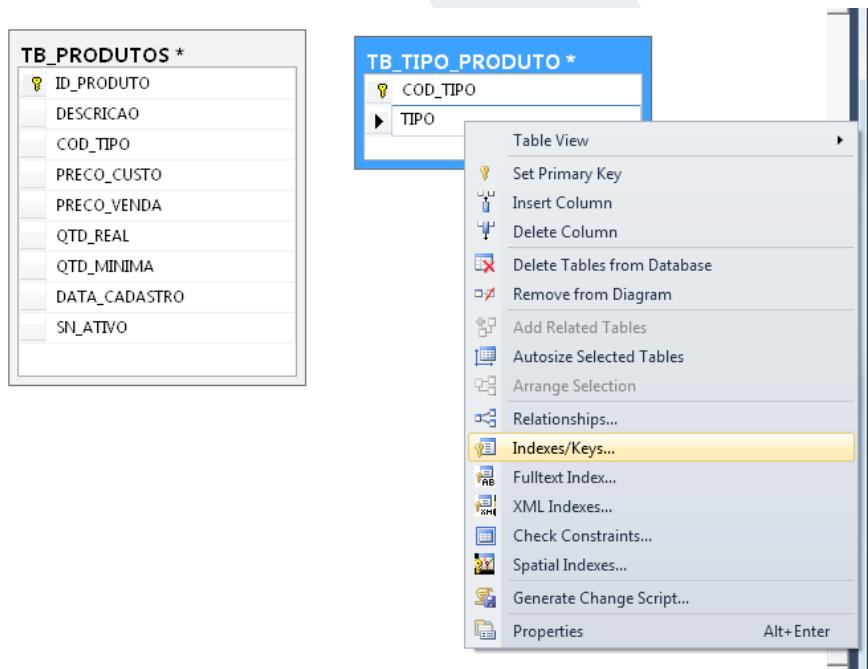
5. Clique com o botão direito do mouse no campo **COD TIPO** da tabela **TB_TIPO_PRODUTO** e selecione a opção **Set Primary Key (Definir Chave Primária)**;



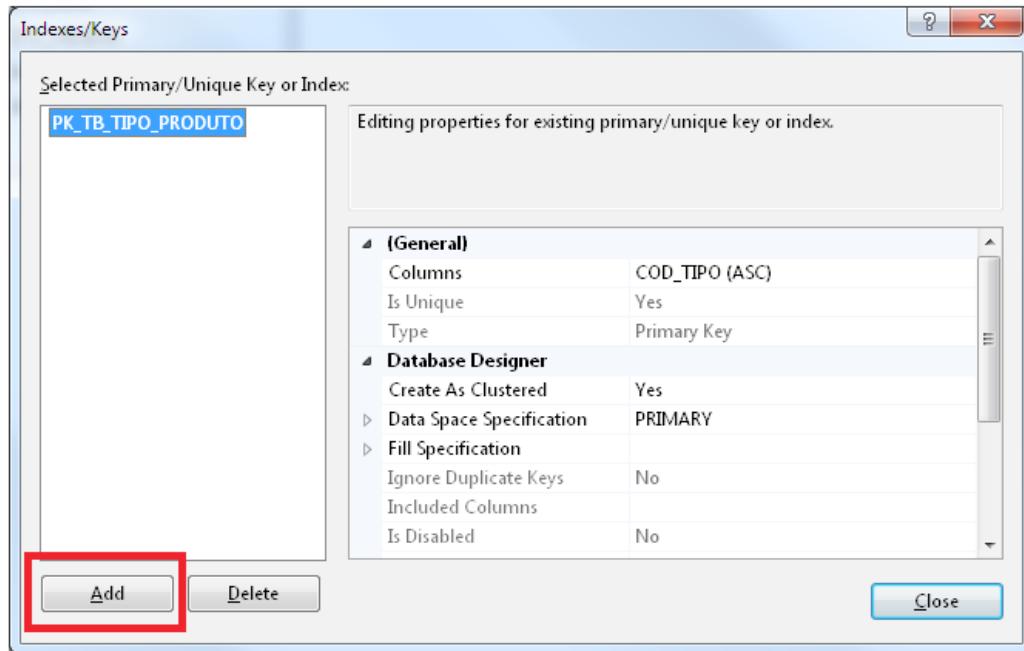
6. Clique com o botão direito do mouse no campo **ID_PRODUTO** da tabela **TB_PRODUTO** e selecione a opção **Set Primary Key (Definir Chave Primária)**. Com isso, serão criadas as chaves primárias das duas tabelas;



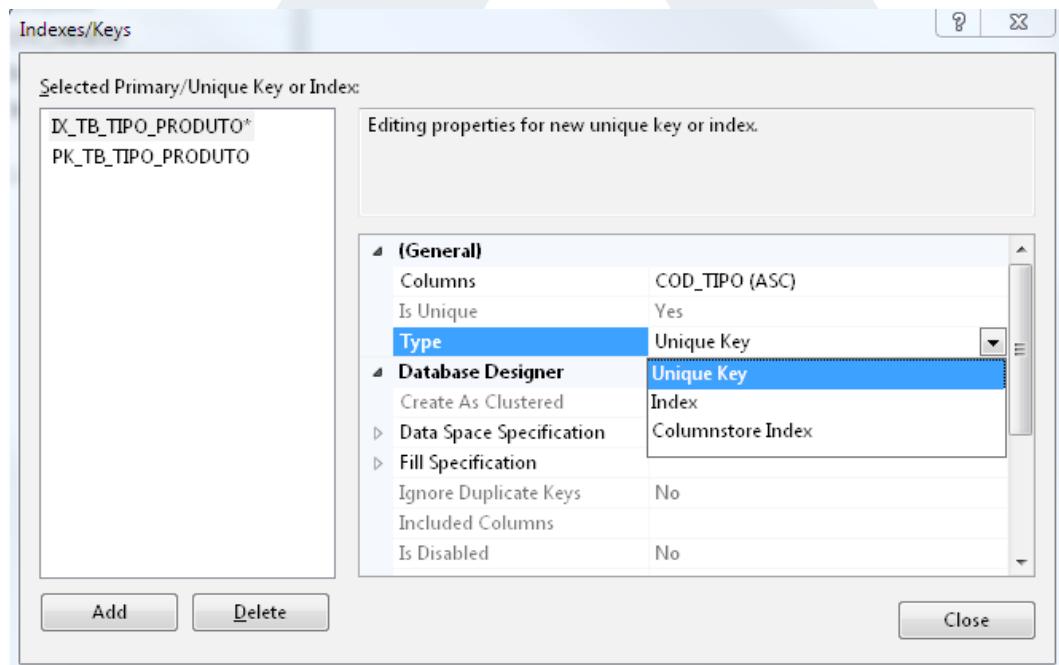
7. Para criar a chave única (**UNIQUE CONSTRAINT**), selecione a tabela **TB_TIPO_PRODUTO**, clique com o botão direito do mouse sobre ela e clique na opção **Indexes/Keys...** (Índices/Chaves...) para abrir a caixa de diálogo **Indexes/Keys**;

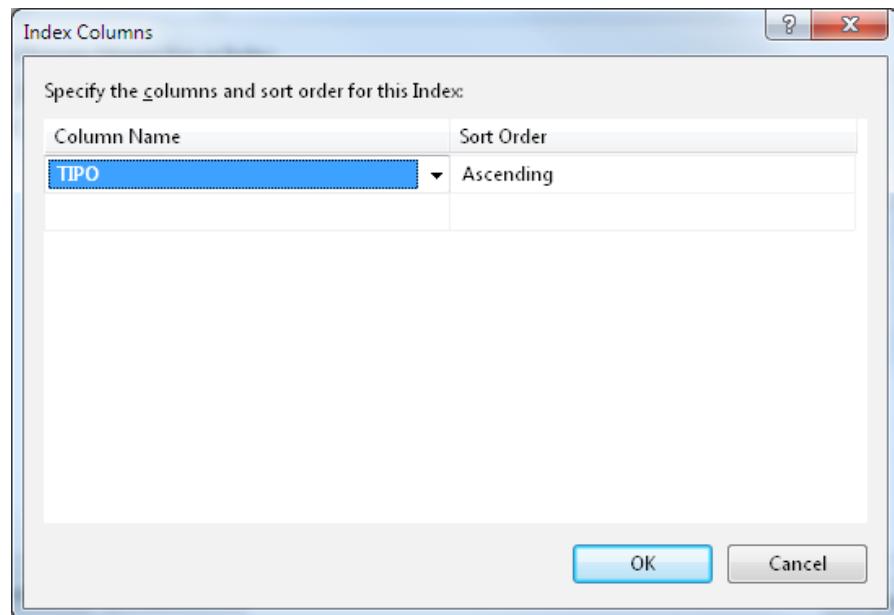


8. Clique no botão Add (Adicionar);

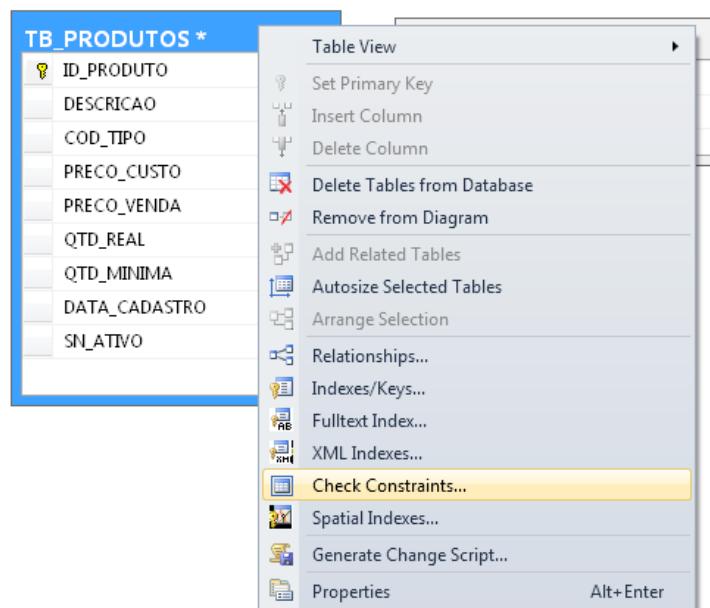


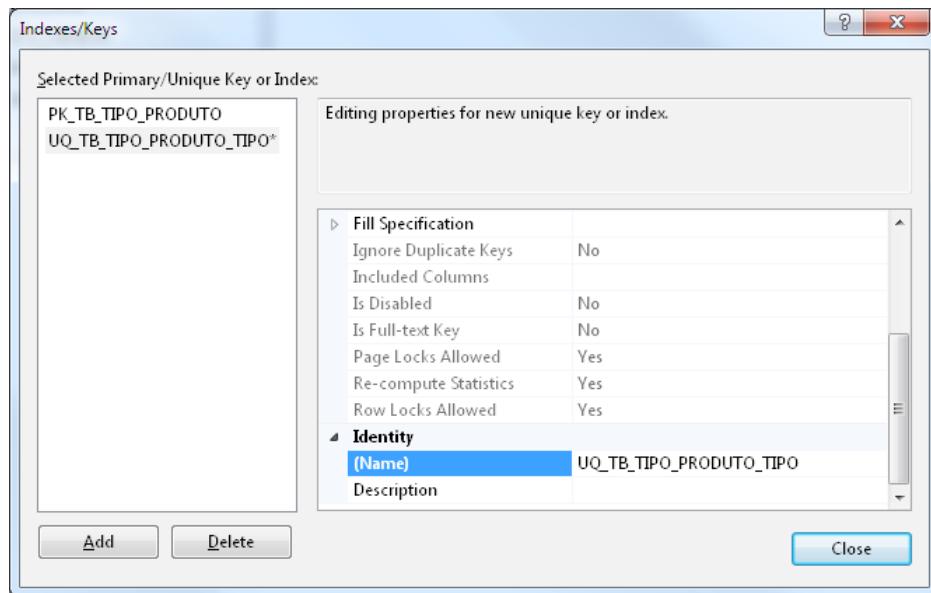
9. Altere as propriedades Type (Tipo) para Unique Key (Chave Exclusiva) e Columns (Colunas) para TIPO;



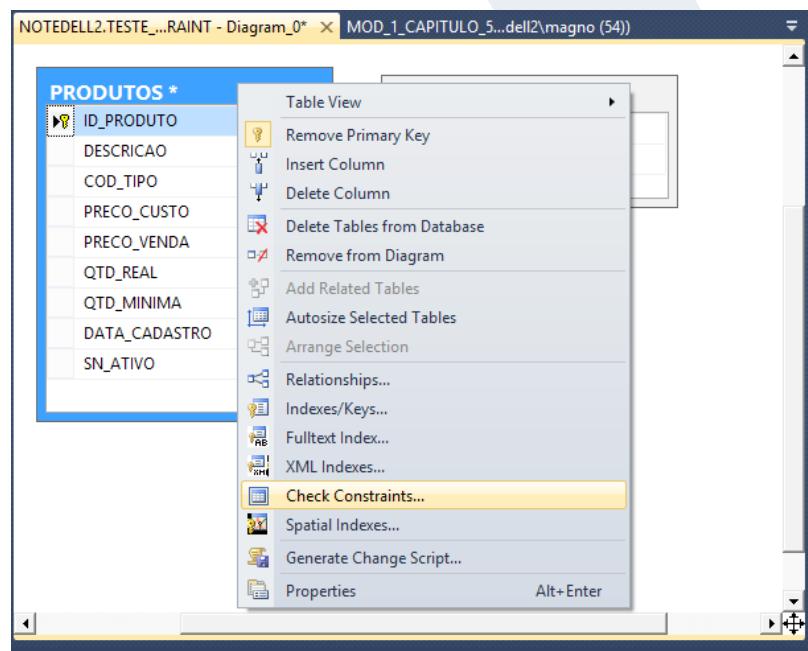


10. Altere as propriedades Name (Nome) para UQ_TB_TIPO_PRODUTO_TIPO. Depois, clique em Close (Fechar);

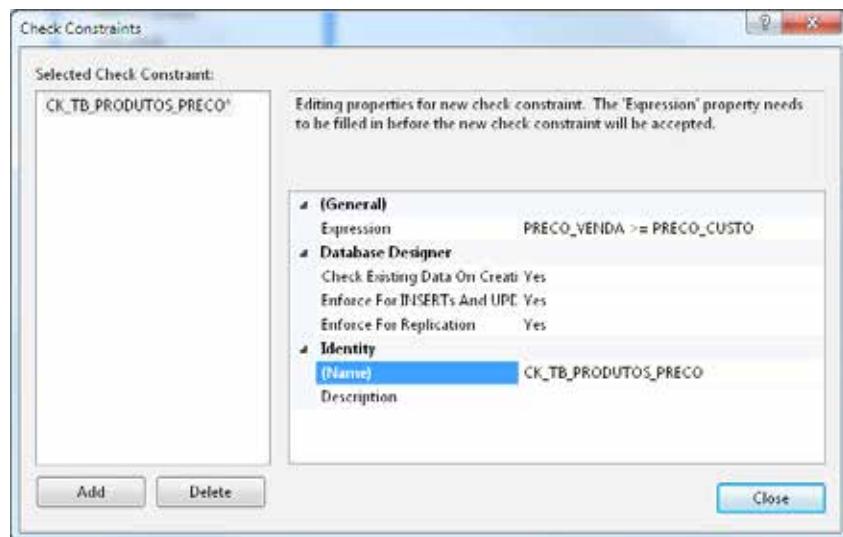




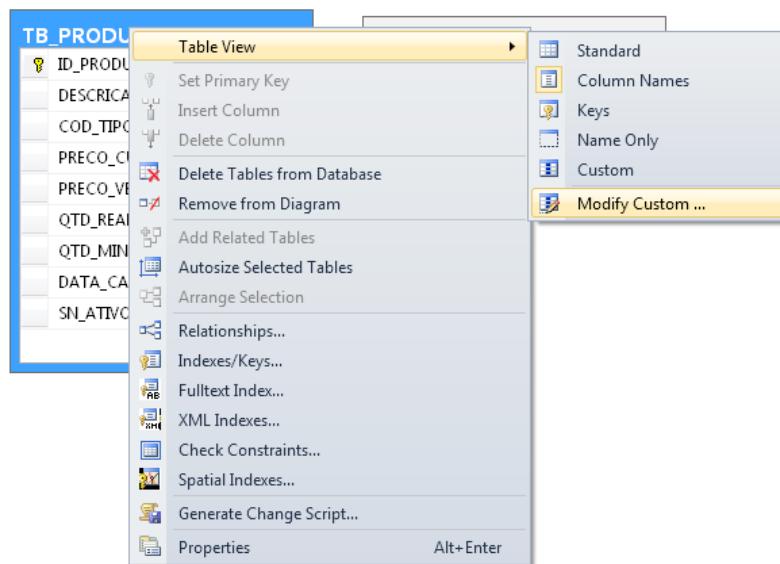
11. Para criar as constraints **CHECK**, clique com o botão direito do mouse sobre a tabela **TB_PRODUTO** e selecione a opção **Check Constraints...** (Restrições de Verificação...);



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



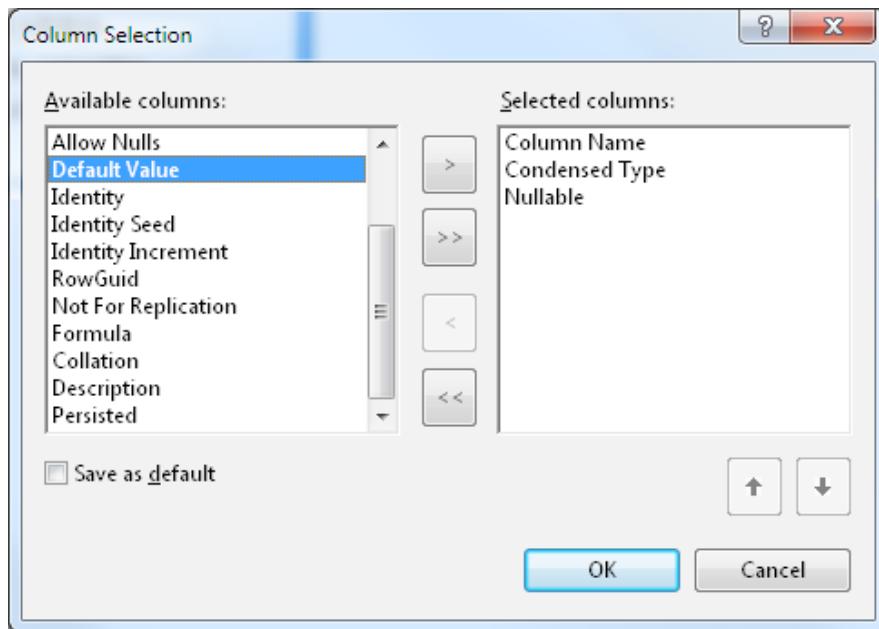
13. Para definir os valores default (padrão) de cada campo, clique com o botão direito do mouse na tabela **TB_PRODUTO**, selecione **Table View (Exibição da Tabela...)** e depois **Modify Custom View (Modificar Personalização)**:



Criando um banco de dados

Aulas 6 e 7

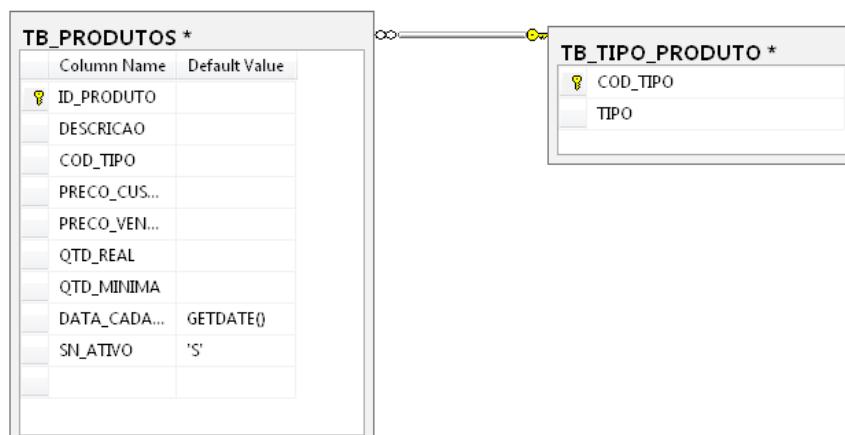
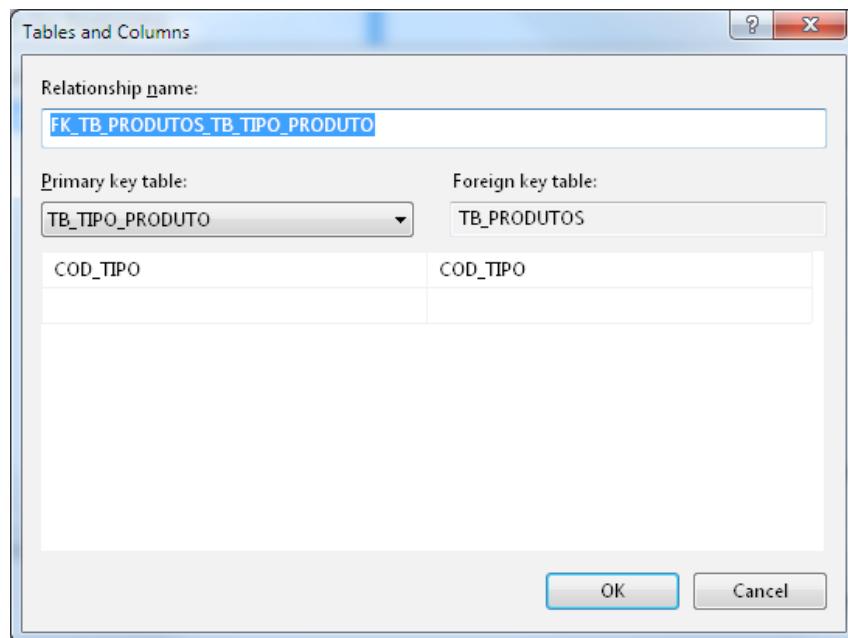
14. Na janela aberta, selecione as colunas **Condensed Type** e **Nullable** e remova-as clicando no botão <. Em seguida, adicione a coluna **Default Value** clicando no botão >. Feche a janela clicando em **OK**;



15. Para exibir os valores padrão, clique com o botão direito do mouse sobre a tabela **TB_PRODUTO**, 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**;

The screenshot shows the 'Table View' dialog box for the 'TB_PRODUTOS' table. It displays two tables: 'TB_PRODUTOS' and 'TB_TIPO_PRODUTO'. The 'TB_PRODUTOS' table has columns: ID_PRODUTO, DESCRICAO, COD TIPO, PRECO_CUS..., PRECO_VEN..., QTD_REAL, QTD_MINIMA, DATA_CADA..., and SN_ATIVO. The 'SN_ATIVO' column has a 'Default Value' of 'GETDATE()' and a 'Value' of 'S'. The 'TB_TIPO_PRODUTO' table has columns: COD_TIPO and TIPO.

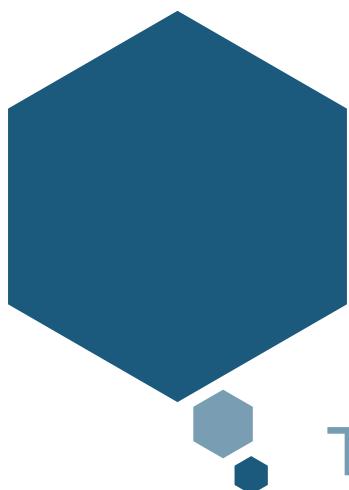
16. Para definir a chave estrangeira, selecione o campo **COD_TIPO** da tabela **TB_PRODUTO** e arraste-o até o campo **COD_TIPO** da tabela **TIPO_PRODUTO**.



Pontos principais

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

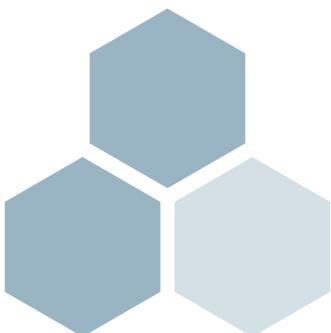
- Os objetos que fazem parte de um sistema são criados dentro de um objeto denominado **database**, ou seja, uma estrutura lógica formada por dois tipos de arquivo: um responsável pelo armazenamento de dados e outro que armazena as transações feitas. Para que um banco de dados seja criado no SQL Server, é necessário utilizar a instrução **CREATE DATABASE**;
- Os dados de um sistema são armazenados em objetos denominados tabelas (**tables**). 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 o uso de **CREATE TABLE**, **ALTER TABLE** ou graficamente (a partir da interface do SQL Server Management Studio).



Criando um banco de dados

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 6 e 7.



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

- a) A coluna NOME armazenará sempre 40 caracteres, independentemente 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 TB_ALUNO
(
    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 TB_ALUNO
(
    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 tipo de campo é mais adequado para armazenar a quantidade de pessoas que moram em uma casa?

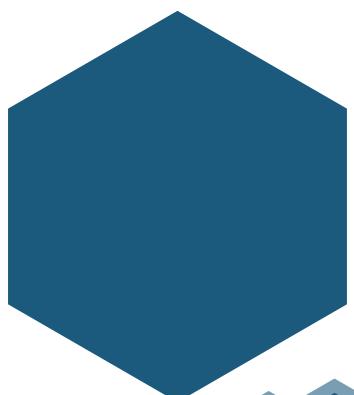
- a) CHAR(2)
- b) INT
- c) SMALLINT
- d) TINYINT
- e) NUMERIC(4,2)

6. 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)

7. 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!

Este laboratório refere-se ao conteúdo das Aulas 6 e 7.



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 **TB_PEDIDO**:
 - Com **TB_CLIENTE**;
 - Com **TB_VENDEDOR**.
4. Crie chaves estrangeiras para a tabela **TB_PRODUTO**:
 - Com **TB_TIPOPRODUTO**;
 - Com **TB_UNIDADE**.
5. Crie chaves estrangeiras para a tabela **TB_ITENSPEDIDO**:
 - Com **TB_PEDIDO**;
 - Com **TB_PRODUTO**;
 - Com **TB_COR**.
6. Crie uma chave única para o campo **UNIDADE** da tabela **TB_UNIDADE**;
7. Crie uma chave única para o campo **TIPO** da tabela **TB_TIPOPRODUTO**;

8. Crie constraints **CHECK** para a tabela **TB_PRODUTO**, considerando os seguintes aspectos:

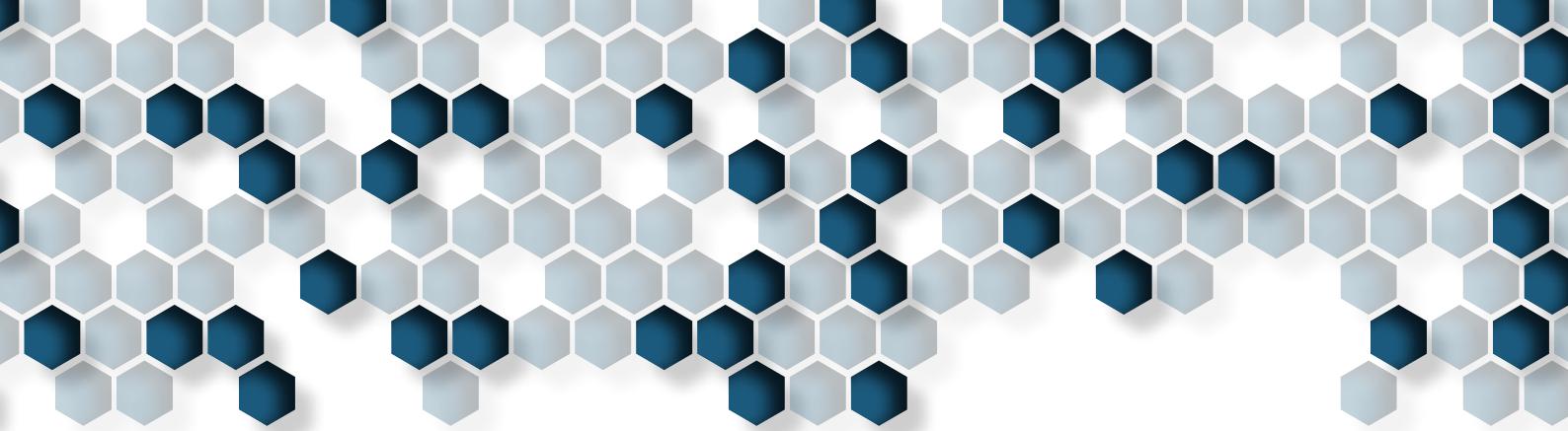
- O preço de venda não pode ser menor que o preço de custo;
- O preço de custo precisa ser maior que zero;
- O campo **QTD_REAL** não pode ser menor que zero.

9. Crie constraints **CHECK** para a tabela **TB_ITENSPEDIDO**, 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 **TB_PRODUTO**, considerando os seguintes aspectos:

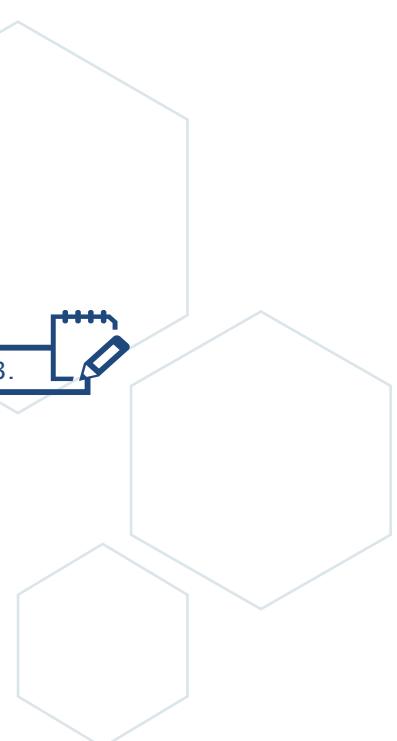
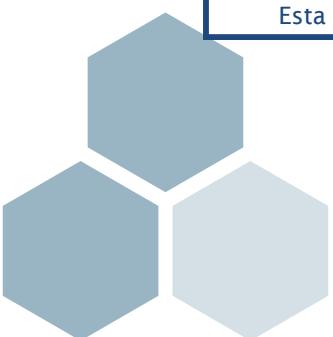
- Zero para **PRECO_CUSTO** e **PRECO_VENDA**;
- Zero para **QTD_REAL**, **QTD_MINIMA** e **QTD_ESTIMADA**;
- Zero para **COD_TIPO** e **COD_UNIDADE**.



Manipulando dados

- 
- 
- ◆ Constantes;
 - ◆ Inserção de dados;
 - ◆ Utilização de TOP em uma instrução INSERT;
 - ◆ OUTPUT;
 - ◆ Atualização e exclusão de dados;
 - ◆ UPDATE;
 - ◆ DELETE;
 - ◆ OUTPUT para DELETE e UPDATE;
 - ◆ Transações.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 10 a 13.



1.1. 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 escrito 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 10 ⁴	2.53 x 10000	25300
4.5E-2	4.5 / 10 ²	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 quatro 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.

1.2. 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:

```
--Caso a tabela não tenha sido criada
CREATE TABLE TB_ALUNO
(
    COD_ALUNO           INT IDENTITY PRIMARY KEY,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO    DATETIME,
    IDADE               TINYINT,
    E_MAIL               VARCHAR(50),
    FONE_RES             CHAR(9),
    FONE_COM             CHAR(9),
    FAX                 CHAR(9),
    CELULAR              CHAR(9),
    PROFISSAO            VARCHAR(40),
    EMPRESA              VARCHAR(50) );
GO

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

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

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

```
INSERT INTO TB_ALUNO
(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 TB_ALUNO;
```

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 TB_ALUNO;
```

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.

1.2.1. **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 TB_ALUNO
VALUES
('MARIA LUIZA', '1997.10.29', 15, 'luiza@luiza.com',
'23456789','23459876','','998765432',
'ESTUDANTE', 'COLÉGIO MONTE VIDEL');

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

1.2.2.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 TB_ALUNO
(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 TB_ALUNO;
```

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.

1.3.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 **TB_CLIENTE** 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 TB_CLIENTE
-- para a tabela CLIENTES_MG
INSERT TOP( 20 ) INTO CLIENTES_MG
SELECT CODCLI, NOME, ENDERECO, BAIRRO, CIDADE, FONE1
FROM TB_CLIENTE
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 MIL...	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. SOMA5\SQLEXPRESS2008 (10.0 ...) SOMA5\CARLOS MAGNO SOU... PEDIDOS 00:00:00 | 20 linhas

1.4. 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.

1.4.1. 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 **TB_EMPREGADO** 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 TB_EMPREGADO;
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 TB_EMPREGADO WHERE COD_DEPTO = 2;
GO
```

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 TB_EMPREGADO 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
```

1.5. Atualizando e excluindo dados

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.

1.6. 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.

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

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
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Retorna o resto inteiro de uma divisão
=	Atribuição

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

Operadores	Descrição
<code>+=</code>	Soma e atribui
<code>-=</code>	Subtrai e atribui
<code>*=</code>	Multiplica e atribui
<code>/=</code>	Divide e atribui
<code>%=</code>	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
```

1.6.1.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 TB_EMPREGADO
SET SALARIO = SALARIO * 1.2;
-- OU
UPDATE TB_EMPREGADO
SET SALARIO *= 1.2;

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

1.6.2. 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 TB_CLIENTE WHERE CODCLI = 5;

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

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

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

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

-- Alterar os dados do grupo de produtos
UPDATE TB_PRODUTO 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 TB_PRODUTO
WHERE COD_TIPO = 5;
```

1.6.3. 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;
```

1.7. 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>
```

1.7.1. 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 **TB_EMPREGADO**:

```
SELECT * INTO EMPREGADOS_TMP FROM TB_EMPREGADO;
```

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 os 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;
```

1.7.2. 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 TB_CLIENTE;  
-- 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	EUDELIO 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	0313623087
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	CAMPESTRE	035743 1554
9	202	EIDER PERPETUO	R.RIO PARAOPERA,1364	RIACHO DAS PED...	CONTAGEM	0313515683
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

1.8. 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;
```

```
-- Gerar uma cópia da tabela TB_EMPREGADO 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
```

```
-- Inserir dados e exibir os registros inseridos
INSERT INTO EMP_TEMP OUTPUT INSERTED.*
SELECT CODFUN, NOME, COD_DEPTO, COD_CARGO, SALARIO
FROM TB_EMPREGADO;
GO
```

```
-- Deletar registros e mostrar os registros deletados
DELETE FROM EMP_TEMP OUTPUT DELETED.*
WHERE COD_DEPTO = 2
GO
```

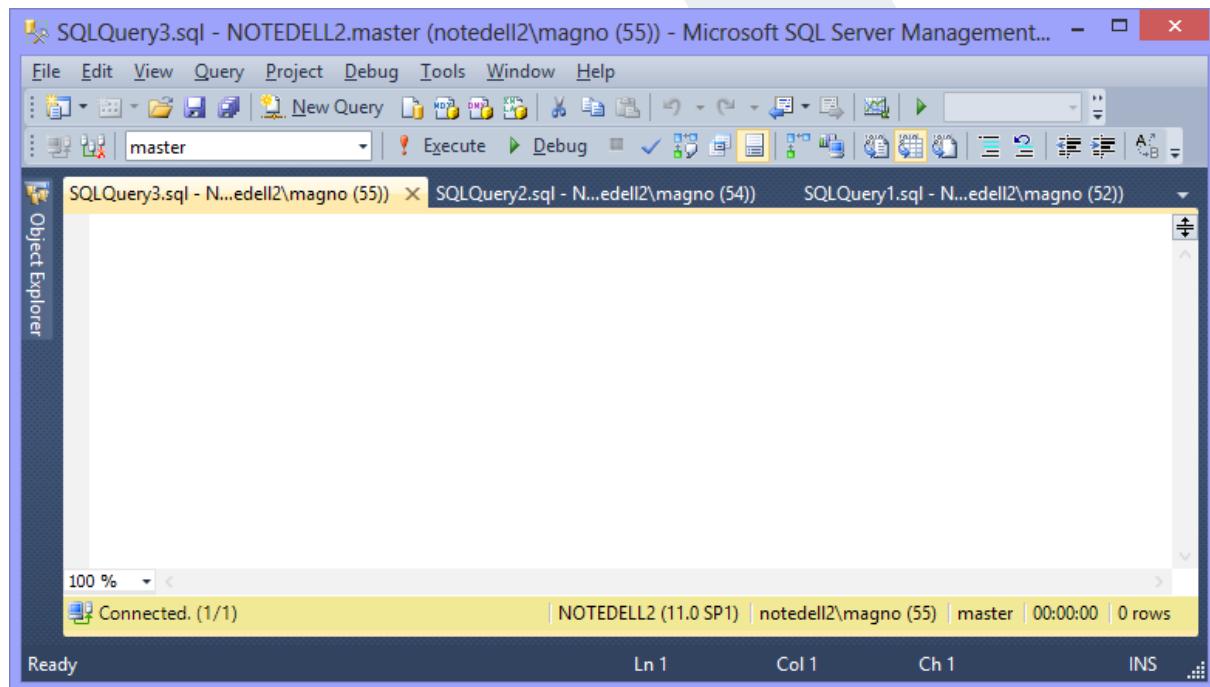
```
-- Alterar registros e mostrar os dados antes e depois da
-- alteração
UPDATE EMP_TEMP SET SALARIO *= 1.5
OUTPUT
    INSERTED.CODFUN, INSERTED.NOME, INSERTED.COD_DEPTO,
    DELETED.SALARIO AS SALARIO_ANTIGO,
    INSERTED.SALARIO AS SALARIO_NOVO
WHERE COD_DEPTO = 3;
GO
```

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.

1.9. 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.

1.9.1. 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.

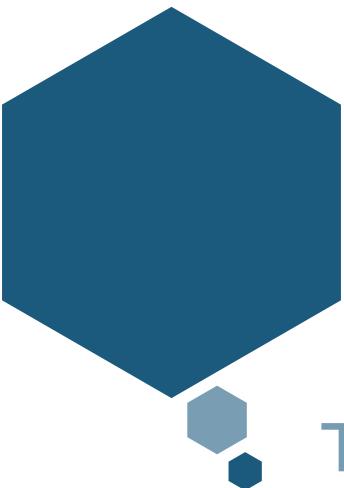
Veja exemplos de transação:

```
-- 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 TB_EMPREGADO 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 da leitura.

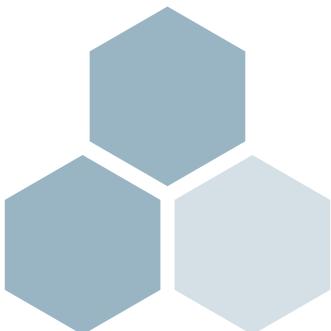
- Para acrescentar novas linhas de dados em uma tabela, utilizamos o comando **INSERT**;
- No **INSERT** posicional não é necessário mencionar o nome dos campos, porém será obrigatório incluir todos os campos, exceto o campo autonumerável. Já no **INSERT** declarativo, os campos deverão ser informados 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**;
- 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.



Manipulando dados

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 10 a 13.



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

```
INSERT TB_ALUNO
( 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, 'luiza@luiza.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 TB_EMPREGADO;
```

- 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 TB_CLIENTE
```

- 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 TB_CLIENTE.
- d) Serão inseridos 20 registros na tabela CLIENTES_MG, a partir da consulta da tabela TB_CLIENTE.
- e) Serão inseridos 20 registros na tabela CLIENTES_MG, a partir da consulta da tabela TB_CLIENTE, 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.

6. Qual dos comandos adiante insere uma linha com dados na tabela chamada ALUNOS, admitindo-se 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')

7. Qual comando deve ser utilizado para alterar um registro?

- a) DELETE
- b) SELECT
- c) TRUNCATE
- d) UPDATE
- e) INSERT

8. O que o comando a seguir realizará?

```
UPDATE TB_EMPREGADO  
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.

9. Verifique o comando a seguir e selecione a afirmação correta:

```
UPDATE TOP(15) EMP_TEMP SET SALARIO = 10*SALARIO;
```

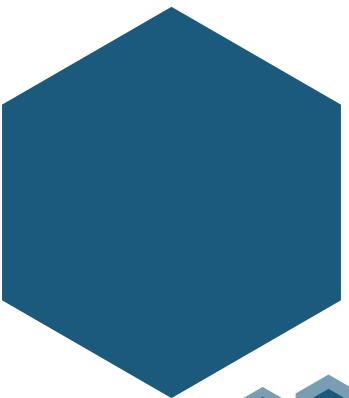
- a) A sintaxe está errada.
- b) A cláusula TOP somente pode ser utilizada no comando SELECT.
- c) O comando UPDATE realizará a atualização de todos os registros da tabela EMP_TEMP.
- d) Serão alterados 15 registros da tabela EMP_TEMP.
- e) A sintaxe está incompleta.

10. Qual comando devemos utilizar para apagar apenas um registro?

- a) DELETE
- b) DELETE ou TRUNCATE
- c) UPDATE
- d) SELECT
- e) INSERT

11. 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.



Manipulando dados



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 10 a 13.



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 **TB_PRODUTO** com os seguintes campos:

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

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

Código da unidade	Inteiro, autonumeração e chave primária
Nome da unidade	Alfanumérico

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

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

Código da categoria	Inteiro, autonumeração e chave primária
Nome da categoria	Alfanumérico

6. Na tabela **TB_CATEGORIA**, 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

Laboratório 2

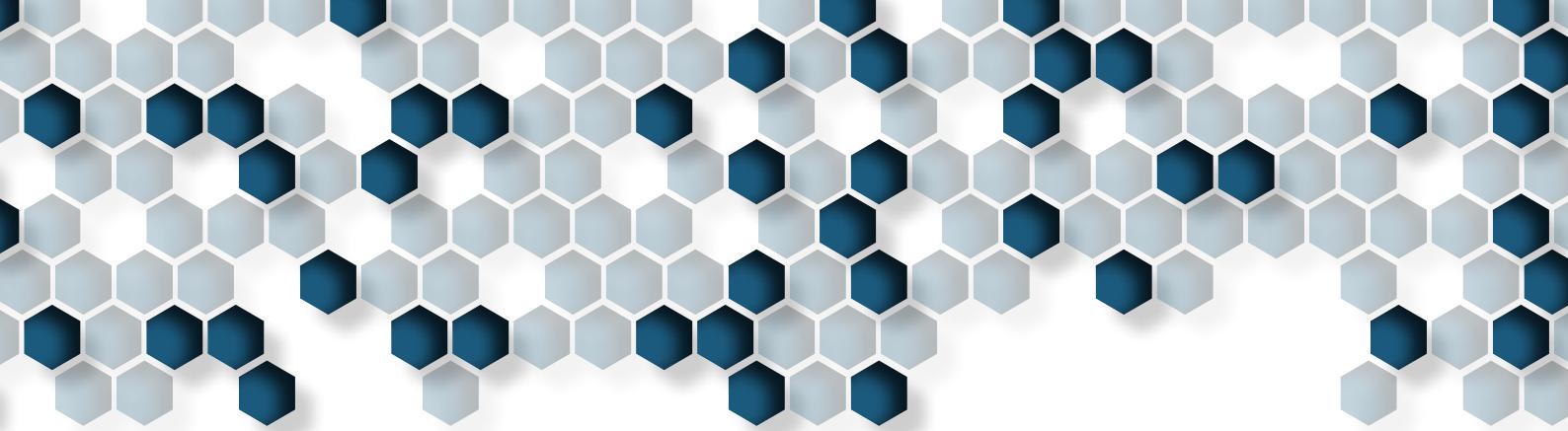
A – Atualizando e excluindo dados

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 **TB_CLIENTE**, 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 **TB_ITENSPEDIDO**, mas somente dos itens do produto com **ID_PRODUTO = 8**, com data de entrega em janeiro de 2014 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-2014** 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 **TB_PEDIDO** 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 2014;
18. Exclua todos os registros restantes da tabela **COPIA_PEDIDOS**;
19. Exclua a tabela **COPIA_PEDIDOS** do banco de dados.



Consultando dados

- ◆ SELECT;
- ◆ Ordenação de dados;
- ◆ Operadores relacionais;
- ◆ Operadores lógicos;
- ◆ Consulta de intervalos com BETWEEN;
- ◆ Consulta com base em caracteres;
- ◆ Consulta de valores pertencentes ou não a uma lista de elementos;
- ◆ Lidando com valores nulos;
- ◆ Substituição de valores nulos;
- ◆ UNION;
- ◆ EXCEPT e INTERSECT.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 14 a 19.



1.1. Introdução

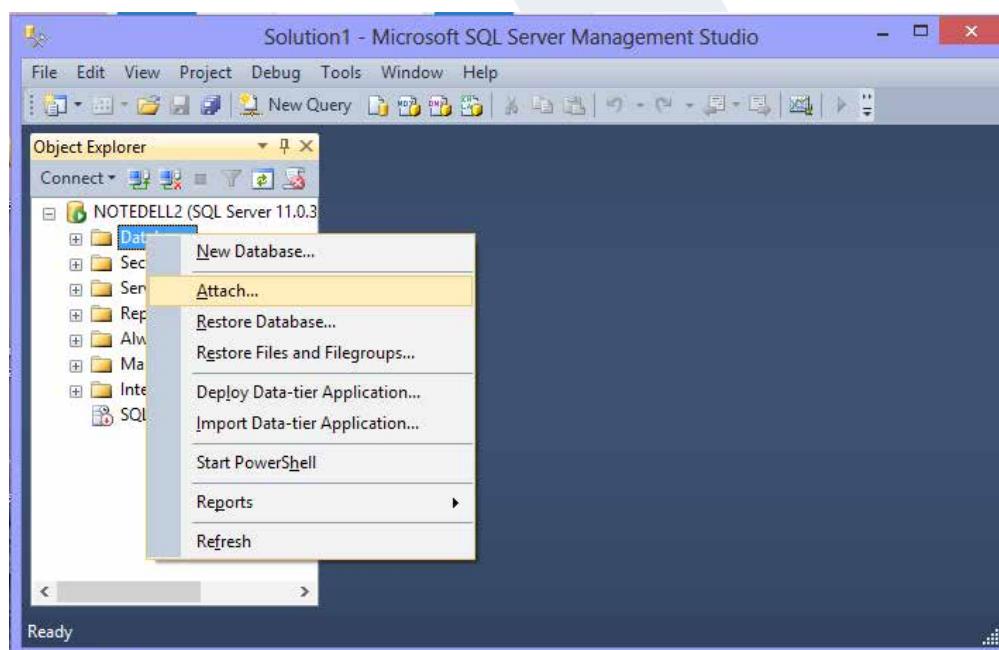
Na linguagem SQL, o principal comando utilizado para a realização de consultas é o **SELECT**. Por meio dele, torna-se possível consultar dados pertencentes a uma ou mais tabelas de um banco de dados.

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

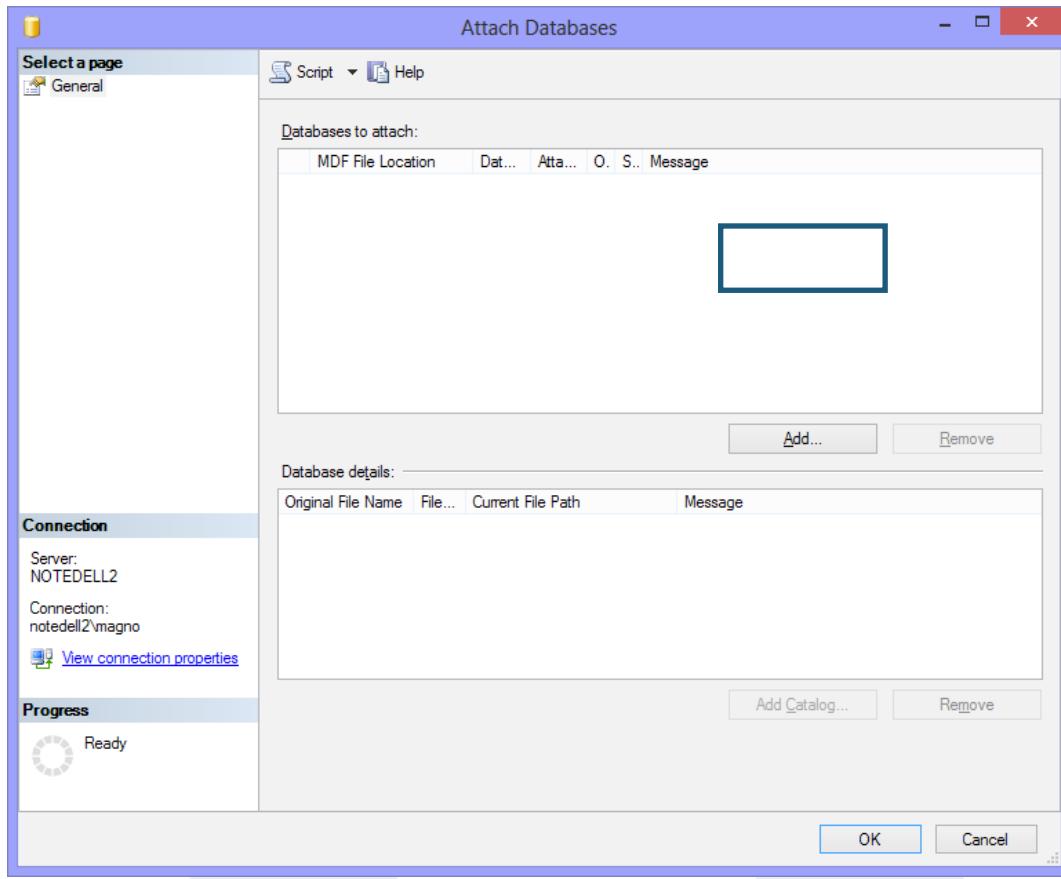
Para que possamos fazer exemplos que demonstrem as técnicas mais apuradas de consulta, precisamos ter um banco de dados com um volume razoável de informações já cadastradas.

Siga os passos adiante:

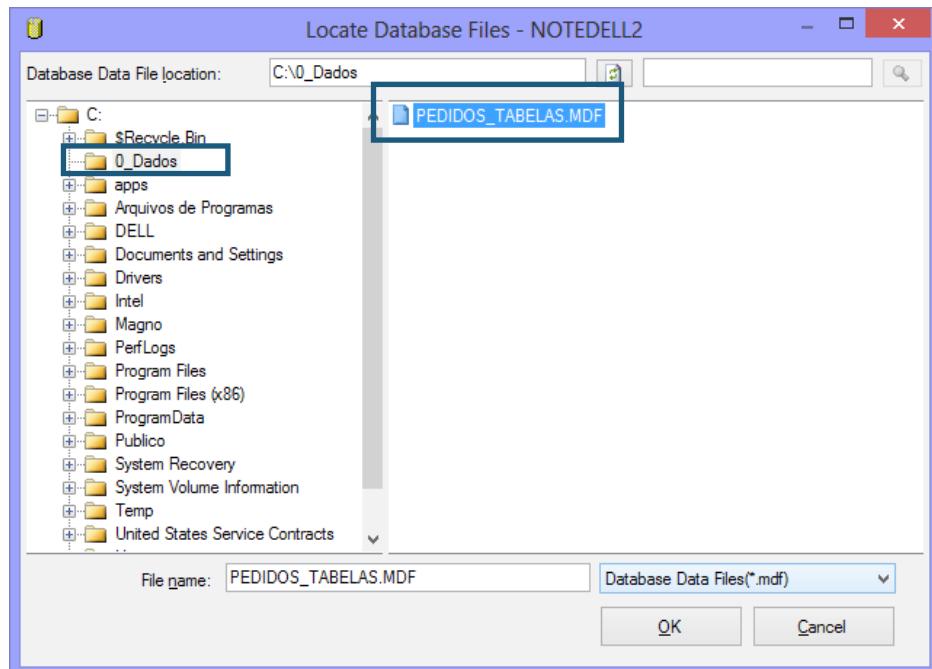
1. No Object Explorer, clique com o botão direito do mouse sobre o item **Databases** e selecione a opção **Attach**:



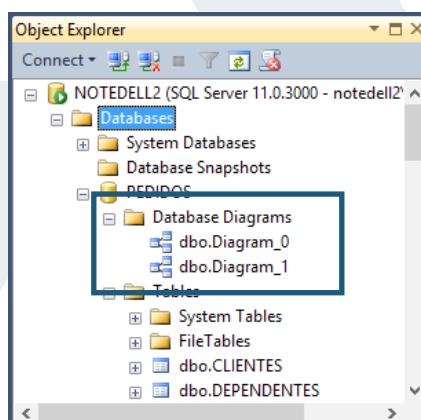
2. Na próxima tela, clique no botão **Add**:



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

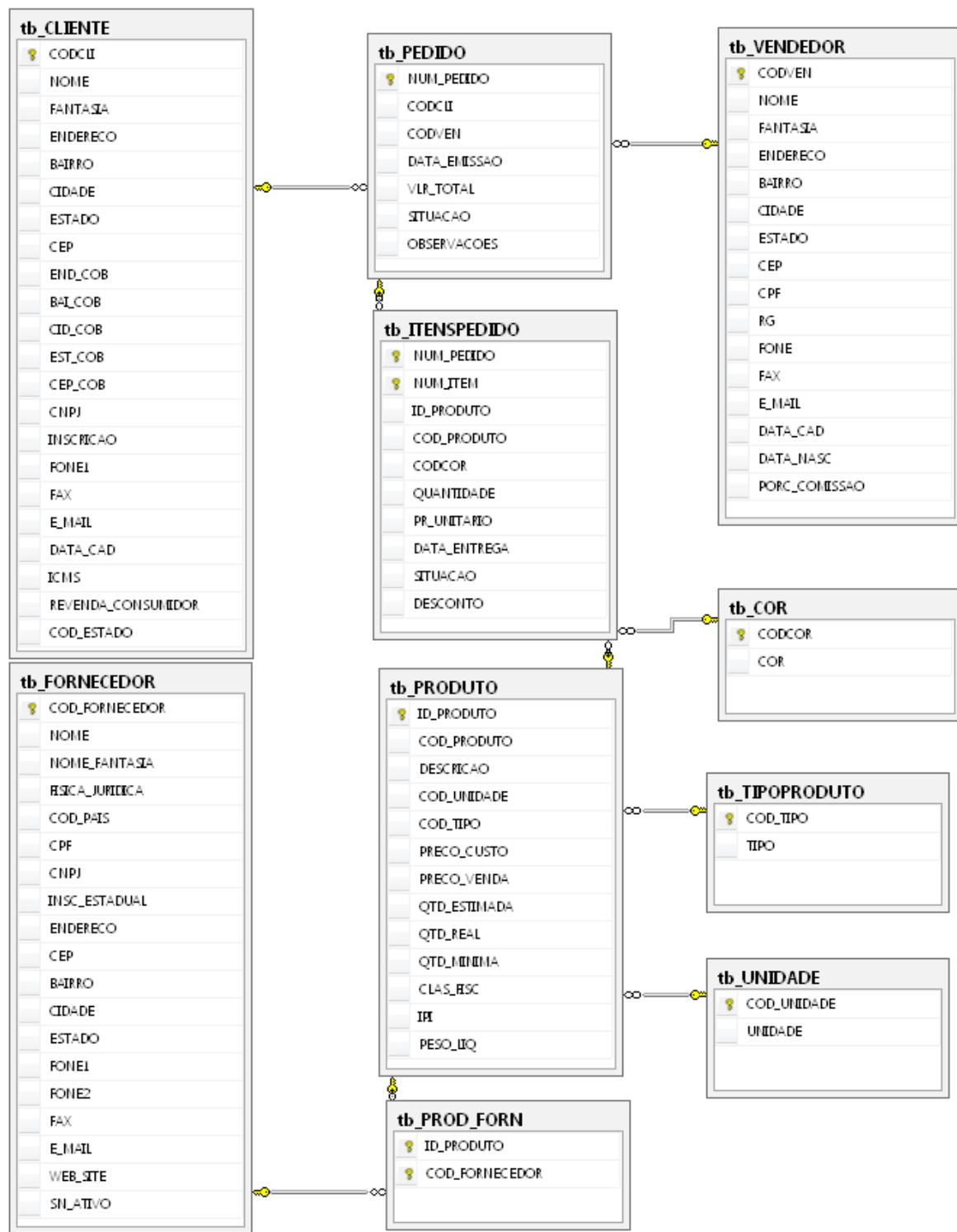


4. Confirme a operação clicando no botão **OK**.

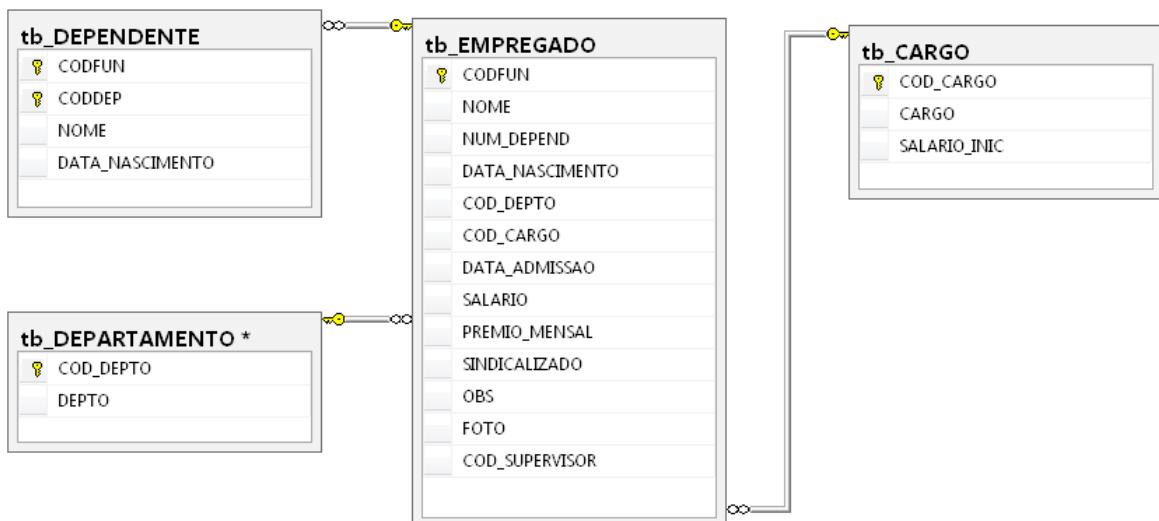


Observe que o banco de dados aparecerá no Object Explorer. Neste banco, foram criados dois diagramas que mostram as tabelas existentes nele. Você conseguirá visualizar o diagrama executando um duplo-clique sobre o nome:

- DIAGRAMA DE PEDIDOS



- **DIAGRAMA DE EMPREGADOS**



1.2. 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 ex-
prJoin [...]]]
    [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. Ambas 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**.

1.2.1. Consultando todas as colunas

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

```
USE PEDIDOS;
SELECT * FROM TB_EMPREGADO;
```

1.2.2. 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>
```

O código a seguir consulta todas as colunas da tabela **TB_EMPREGADO**:

The screenshot shows the Microsoft SQL Server Management Studio (SSMS) interface. The 'Object Explorer' (A) on the left shows the database structure for 'instrutor'. The 'SQLQuery1.sql' window (B) contains the query: 'USE PEDIDOS' followed by 'SELECT * FROM TB_EMPREGADO'. The 'Execute' button (C) is highlighted. The 'Results' pane (D) displays the output of the query, showing columns: CODFUN, NOME, NUM_DEPEND, DATA_NASCIMENTO, and COD. The results grid contains 8 rows of employee data. The status bar at the bottom indicates the query was executed successfully.

CODFUN	NOME	NUM_DEPEND	DATA_NASCIMENTO	COD
1	OLAVO TRINDADE	1	1950-06-06 00:00:00.000	4
2	JOSE REIS	6	1952-10-09 00:00:00.000	2
3	MARCELO SOARES	1	1950-06-06 00:00:00.000	5
4	PAULO CESAR JUNIOR	2	1952-03-19 00:00:00.000	8
5	JOAO LIMA MACHADO DA SILVA	2	1955-10-30 00:00:00.000	4
6	CARLOS ALBERTO SILVA	0	1961-07-06 00:00:00.000	11
7	FELIANE PERFIRA	0	1955-11-14 00:00:00.000	6
	!!!			

- **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 **TB_EMPREGADO**:

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

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 TB_EMPREGADO;
```

Veja o resultado:

	CODFUN	NOME	SALARIO	(Nenhum nome de colu...)
1	1	OLAVO 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.

1.2.3. 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>
```

Vejamos 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 TB_EMPREGADO;
```

Confira o resultado:

	CODFUN	NOME	SALARIO	SALARIO_MAIS_10_POR_CEN...
1	1	OLAVO 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
8	9	RUDGE RAMOS SANTANA DA PENHA	800.00	880.0000
9	10	MARIA CARMEM	1200.00	1320.0000
10	11	FERNANDO OLIVEIRA	1200.00	1320.0000
11	12	JOAO ROBERTO OLIVEIRA	1200.00	1320.0000
12	13	OSMAR PRADO	2400.00	2640.0000
13	14	CASSIANO OLIVEIRA	1200.00	1320.0000
14	15	MARCO ANTONIO	2400.00	2640.0000
15	16	ALTAMIR CARCIO	3300.00	3630.0000
16	17	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 TB_EMPREGADO;
```

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

1.3. Ordenando dados

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

1.3.1. 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.

Vejamos um exemplo:

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

1.3.2. 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.

Vejamos um exemplo:

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

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

1.3.3. 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 TB_EMPREGADO  
ORDER BY Salário;
```

-- Idem ao anterior

```
SELECT CODFUN AS Código,  
       NOME AS Nome,  
       SALARIO AS Salário,  
       SALARIO * 1.10 [Salário com 10% de aumento]  
FROM TB_EMPREGADO  
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 TB_EMPREGADO  
ORDER BY [Salário com 10% de Aumento];
```

-- Idem ao anterior

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

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

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

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

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

1.3.4. 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 ficará sem sentido. Vejamos o exemplo:

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

	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 **TB_EMPREGADO**, 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 TB_EMPREGADO
ORDER BY DATA_ADMISSAO;

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

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

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

1.3.5. 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 TB_EMPREGADO
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

Esse 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 TB_EMPREGADO  
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.

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 TB_EMPREGADO  
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 **TB_EMPREGADO** possui sessenta linhas, então, se pedirmos pra ver 10% das linhas, deverão aparecer seis. Confira:

```
-- Mostrar 10% das linhas da tabela TB_EMPREGADO
SELECT TOP 10 PERCENT CODFUN, NOME, SALARIO FROM TB_EMPREGADO
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. | SQLSERVER1 (11.0 RTM) | SA (54) | PEDIDOS | 00:00:00 | 6 rows

1.3.6. 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).

1.4. 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 TB_EMPREGADO
WHERE SALARIO < 1000
ORDER BY SALARIO;
```

- **Mostrando os funcionários com SALÁRIO acima de 5000**

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

- **Mostrando os funcionários com campo COD_DEPTO menor ou igual a 3**

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

- **Mostrando os funcionários com campo COD_DEPTO igual a 2**

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

- Mostrando os funcionários com campo COD_DEPTO diferente de 2

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

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

- Mostrando os funcionários que tenham NOME alfabeticamente maior que RAQUEL

```
SELECT CODFUN, NOME, SALARIO  
FROM TB_EMPREGADO  
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 TB_EMPREGADO  
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

1.5. 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 TB_EMPREGADO
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 TB_EMPREGADO
WHERE COD_DEPTO = 2 OR SALARIO > 5000;
```

Resultados		Mensagens		
	CODFUN	NOME	NUM_DEPE...	DATA_NASI
1	2	JOSE REIS	6	1952-10-09
2	9	RUDGE RAMOS SANTANA DA PENHA	3	1961-07-22
3	18	ROBERTO PINHEIRO	4	1950-04-12
4	19	SEBASTIÃO SILVA	0	1951-12-19
5	20	EURICO BRANDÃO	0	1950-04-19
6	25	MARIA DA PENHA	0	1958-02-12
7	28	MARIANO DE OLIVEIRA	3	1954-01-18
8	38	LUIS FERNANDO LEMOS	0	1978-07-23
9	40	JOAQUIM ALBERTO	0	1991-03-04
10	66	JOSÉ CARLOS MOREIRA	0	1900-01-01

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, digitariamso o seguinte:

```
SELECT * FROM TB_EMPREGADO
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. Então, o correto é o seguinte:

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

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

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

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

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

```
SELECT * FROM TB_EMPREGADO  
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 TB_EMPREGADO  
WHERE NOT (SALARIO >= 3000 AND SALARIO <= 5000)  
ORDER BY SALARIO;
```

1.6. 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 **>=** e **<=** ou interligadas pelo operador **OR**.

A funcionalidade da cláusula **BETWEEN** assemelha-se à dos operadores **AND**, **>=** e **<=**, 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 TB_EMPREGADO
WHERE SALARIO >= 3000 AND SALARIO <= 5000
ORDER BY SALARIO;

SELECT * FROM TB_EMPREGADO
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 TB_EMPREGADO
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 TB_EMPREGADO
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 mesmos valores da consulta anterior:

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

1.7. 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.

Vejamos os seguintes exemplos:

- Nomes que começam com MARIA

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

- Nomes que começam com M

```
SELECT * FROM TB_EMPREGADO  
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 que terminam com MARIA

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

- Nomes que terminam com SOUZA

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

- Nomes que terminam com ZA

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

- Nomes contendo a palavra MARIA

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

- Nomes contendo a palavra SOUZA

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

- **Nomes cujo penúltimo caractere seja a letra Z**

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

- **Nomes terminados em LU e seguidos de 3 outras letras**

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

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

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

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

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

1.8. 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 TB_EMPREGADO  
WHERE COD_DEPTO IN (1,3,4,7)  
ORDER BY COD_DEPTO;
```

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

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

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

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

1.9. 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 TB_EMPREGADO (NOME)  
VALUES ('JOSE EMANUEL');  
  
-- Ver o resultado  
SELECT * FROM TB_EMPREGADO;
```

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 TB_EMPREGADO
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

Como dito anteriormente, caso faça parte de cálculo, o resultado é sempre **NULL**:

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

-- 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 **TB_EMPREGADO**, os registros cujo **COD_CARGO** seja nulo:

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

Este exemplo busca, na tabela **TB_EMPREGADO**, os registros cuja **DATA_NASCIMENTO** seja nula:

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

O exemplo adiante busca, na tabela **TB_EMPREGADO**, os registros cuja **DATA_NASCIMENTO** não seja nula:

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

1.10. 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.

1.10.1. ISNULL

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

- **Exemplo 1**

```
SELECT  
    CODFUN, NOME, SALARIO, PREMIO_MENSAL,  
    ISNULL(SALARIO,0) + ISNULL(PREMIO_MENSAL,0) AS RENDA_TOTAL  
FROM TB_EMPREGADO  
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 TB_EMPREGADO;
```

1.10.2. 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 **TB_CLIENTE**. 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 TB_CLIENTE
ORDER BY 3;
```

1.11. 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 TB_CLIENTE
UNION
SELECT NOME, FONE1 FROM TB_CLIENTE ORDER BY NOME;
```

1.11.1. 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 TB_CLIENTE  
UNION ALL  
SELECT NOME, FONE1 FROM TB_CLIENTE ORDER BY NOME;
```

1.12. 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 R\$ 5.000,00:

```
SELECT COD_DEPTO FROM TB_DEPARTAMENTO
INTERSECT
SELECT COD_DEPTO FROM TB_EMPREGADO
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 TB_DEPARTAMENTO  
EXCEPT  
SELECT COD_DEPTO FROM TB_EMPREGADO
```

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 TB_CARGO  
EXCEPT  
SELECT COD_CARGO FROM TB_EMPREGADO
```

O resultado do código anterior é exibido a seguir:

COD_CARGO
7
13
15

- **Exemplo 4**

O código a seguir consulta os clientes que compraram em janeiro de 2014:

```
SELECT CODCLI FROM TB_CLIENTE  
INTERSECT  
SELECT CODCLI FROM TB_PEDIDO  
WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
```

O resultado do código anterior é exibido a seguir:

CODCLI
100
111
235
267
313
393

- **Exemplo 5**

O código a seguir consulta os clientes que não compraram em janeiro de 2014:

```
SELECT CODCLI FROM TB_CLIENTE  
EXCEPT  
SELECT CODCLI FROM TB_PEDIDO  
WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
```

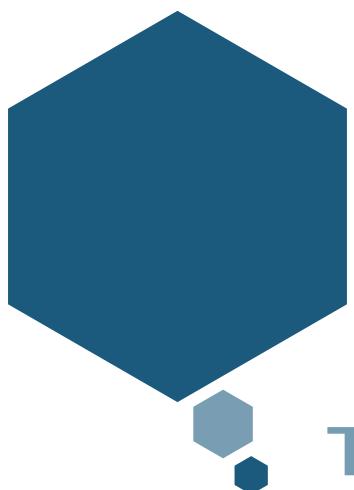
O resultado do código anterior é exibido a seguir:

CODCLI
6
33
37
66
70
240

Pontos principais

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

- 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;
- **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**.



Consultando dados

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 14 a 19.



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 TB_EMPREGADO
```

- a) O comando acima altera a tabela TB_EMPREGADO 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 TB_EMPREGADO
```

- 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 TB_CLIENTE WHERE CODCLI 10
- b) SELECT * FROM TB_CLIENTE WHERE CODCLI LIKE '%10%'
- c) SELECT * FROM TB_CLIENTE WHERE CODCLI
- d) SELECT * FROM TB_CLIENTE WHERE ISNULL(CODCLI , 0) >10
- e) SELECT * FROM TB_CLIENTE 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. Após verificar o comando a seguir, qual a melhor afirmação?

```
SELECT * FROM TB_EMPREGADO  
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 incorreta?

- 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. Qual afirmação está incorreta 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.

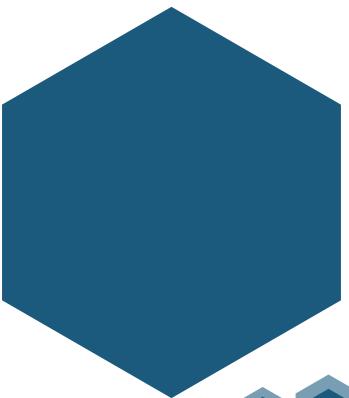
9. Qual comando é utilizado para comparação de consultas?

- a) WHERE
- b) INTERSECT
- c) SELECT
- d) SELECT com subqueries
- e) UNION

10. Após analisar o comando a seguir, qual a resposta correta?

```
SELECT COD_CARGO FROM TB_CARGO  
EXCEPT  
SELECT COD_CARGO FROM TB_EMPREGADO
```

- 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.

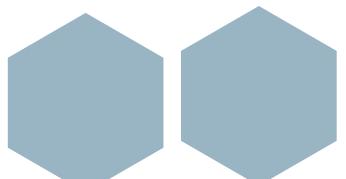


Consultando dados



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 14 a 19.



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 **TB_PRODUTO**, mostrando as colunas **COD_PRODUTO**, **DESCRICAO**, **PRECO_CUSTO**, **PRECO_VENDA** e calculando o lucro unitário (**PRECO_VENDA - PRECO_CUSTO**);
3. Liste a tabela **TB_PRODUTO**, mostrando os campos **COD_PRODUTO**, **DESCRICAO** e calculando o valor total investido no estoque daquele produto (**QTD_REAL * PRECO_CUSTO**);
4. Liste a tabela **TB_ITENSPEDIDO**, 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 **TB_PRODUTO**, 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 **TB_PRODUTO**, 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 **TB_PRODUTO**, 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 **TB_PRODUTO**, 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 **TB_PEDIDO** 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

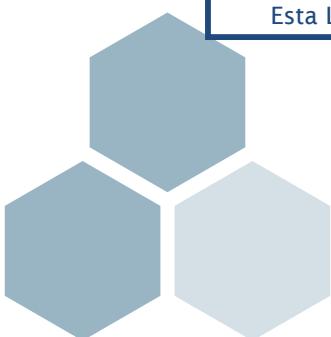
1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste clientes do estado de São Paulo (SP);
3. Liste clientes dos estados de Minas Gerais e Rio de Janeiro (MG, RJ);
4. Liste clientes dos estados de São Paulo, Minas Gerais e Rio de Janeiro (SP, MG, RJ);
5. Liste os vendedores com o nome **LEIA**;
6. 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**.



Associando tabelas

- ◆ INNER JOIN;
- ◆ OUTER JOIN;
- ◆ CROSS JOIN.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 20 a 22.



1.1. 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 nesta leitura: **INNER JOIN**, **OUTER JOIN** e **CROSS JOIN**.

1.2. 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.

	CODFUN	NOME	NUM_DEPE...	DATA_NASCIMENTO	COD_DEP...		COD_DEP...	DEPTO
1	1	OLAVO TRINDADE	1	1950-06-06 00:00:00.000	4		1	PESSOAL
2	2	JOSE REIS	6	1952-10-09 00:00:00.000	2		2	C.P.D.
3	3	MARCELO SOARES	1	1950-06-06 00:00:00.000	5		3	CONTROLE DE ESTOQUE
4	4	PAULO CESAR JUNIOR	2	1952-03-19 00:00:00.000	8		4	COMPRAS
5	5	JOAO LIMA MACHADO DA SILVA	2	1955-10-30 00:00:00.000	4	4	5	PRODUCAO
6	7	CARLOS ALBERTO SILVA	0	1961-07-06 00:00:00.000	11	6	6	DIRETORIA
7	8	ELIANE PEREIRA	0	1955-01-14 00:00:00.000	6	7	7	TELEMARKETING
8	9	RUDGE RAMOS SANTANA DA PENHA	3	1961-07-22 00:00:00.000	2	8	8	FINANCIERO
9	10	MARIA CARMEM	0	1954-03-14 00:00:00.000	5	9	9	RECURSOS HUMANOS
						10	10	TREINAMENTO
						11	11	PRESIDENCIA
						12	12	PORTARIA
						13	13	CONTROLADORIA
						14	14	P.C.P.

TB_EMPREGADO

TB_DEPARTAMENTO

Observando as duas tabelas, é possível concluir que o funcionário 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 TB_EMPREGADO.CODFUN, TB_EMPREGADO.NOME,  
       TB_DEPARTAMENTO.DEPTO  
  FROM TB_EMPREGADO JOIN TB_DEPARTAMENTO  
    ON TB_EMPREGADO.COD_DEPTO = TB_DEPARTAMENTO.COD_DEPTO;
```

```
SELECT TB_EMPREGADO.CODFUN, TB_EMPREGADO.NOME, TB_DEPARTAMENTO.DEPTO  
  FROM TB_EMPREGADO, TB_DEPARTAMENTO  
 WHERE TB_EMPREGADO.COD_DEPTO = TB_DEPARTAMENTO.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 TB_EMPREGADO JOIN TB_DEPARTAMENTO  
    ON TB_EMPREGADO.COD_DEPTO = TB_DEPARTAMENTO.COD_DEPTO;
```

Este exemplo está correto, porque não há duplicidade nos campos **CODFUN**, **NOME** e **DEPTO**.

- **Exemplo 2**

```
SELECT CODFUN, NOME, DEPTO  
  FROM TB_EMPREGADO JOIN TB_DEPARTAMENTO  
    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 TB_EMPREGADO AS E JOIN TB_DEPARTAMENTO AS D  
ON E.COD_DEPTO = D.COD_DEPTO;
```

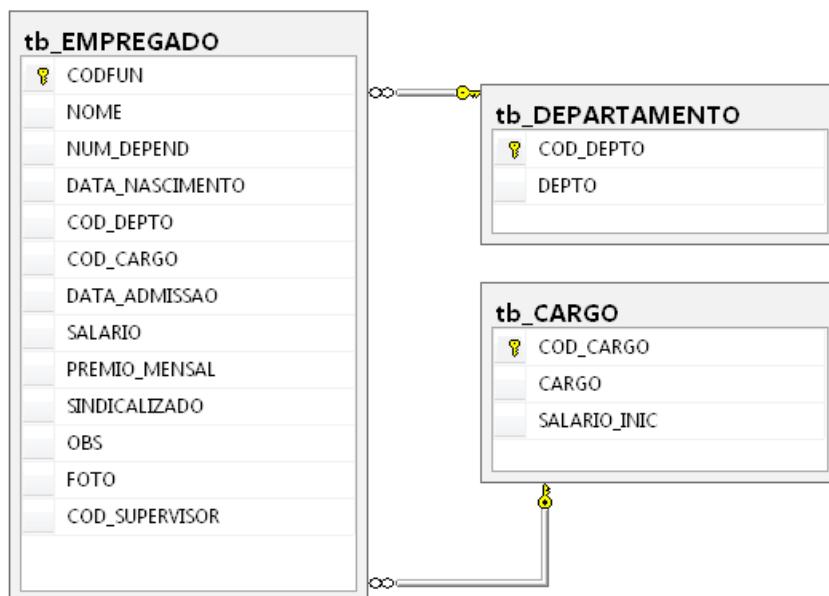
-- OU

```
SELECT E.CODFUN, E.NOME, D.DEPTO  
FROM TB_EMPREGADO E JOIN TB_DEPARTAMENTO 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 **TB_EMPREGADO** e **TB_DEPARTAMENTO**. Essa chave está ao lado da tabela **TB_DEPARTAMENTO**, o que indica que é a chave primária dessa tabela (**COD_DEPTO**), e se relaciona com a tabela **TB_EMPREGADO**, que também possui um campo **COD_DEPTO**, que é a chave estrangeira.

Há, também, um **JOIN** entre as tabelas **TB_EMPREGADO** e **TB_CARGO**. Podemos perceber a existência de uma chave horizontalmente posicionada na linha que liga essas tabelas, e ela está ao lado de **TB_CARGO**. Então, é a chave primária de **TB_CARGO** (**COD_CARGO**) que se relaciona com a tabela **TB_EMPREGADO**. Em **TB_EMPREGADO**, 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 TB_EMPREGADS E
  JOIN TB_DEPARTAMENTO 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**:

```
-- TB_EMPREGADO e TB_CARGO (cargos)
SELECT E.CODFUN, E.NOME, C.CARGO
FROM TB_EMPREGADO E JOIN TB_CARGO C ON E.COD_CARGO = C.COD_CARGO;
-- OU
SELECT E.CODFUN, E.NOME, C.CARGO
FROM TB_CARGO C JOIN TB_EMPREGADO E ON E.COD_CARGO = C.COD_CARGO;
```

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 TB_EMPREGADO E
    JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO
    JOIN TB_CARGO 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 TB_DEPARTAMENTO D
    JOIN TB_EMPREGADO E ON E.COD_DEPTO = D.COD_DEPTO
    JOIN TB_CARGO 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 TB_CARGO C
    JOIN TB_EMPREGADO E ON E.COD_CARGO = C.COD_CARGO
    JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO;
```

Na consulta a seguir, temos dois erros. O primeiro é que não há **JOIN** entre **TB_DEPARTAMENTO** e **TB_CARGO**, 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 TB_CARGO C
    JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO    --<< (1)
    JOIN TB_EMPREGADO 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:
    TB_ITENSPEDIDO.NUM_PEDIDO
    TB_ITENSPEDIDO.NUM_ITEM
    TB_ITENSPEDIDO.COD_PRODUTO
    TB_PRODUTO.DESCRICAO
    TB_ITENSPEDIDO.QUANTIDADE
    TB_ITENSPEDIDO.PR_UNITARIO
    TB_TIPOPRODUTO.TIPO
    TB_UNIDADE.UNIDADE
    TB_COR.COR
    TB_PEDIDO.DATA_EMISSAO

    Filtrando TB_PEDIDO emitidos em Janeiro de 2014
*/
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 TB_ITENSPEDIDO I
    JOIN TB_PRODUTO PR    ON I.ID_PRODUTO      = PR.ID_PRODUTO
    JOIN TB_COR CR        ON I.CODCOR         = CR.CODCOR
    JOIN TB_TIPOPRODUTO T ON PR.COD_TIPO      = T.COD_TIPO
    JOIN TB_UNIDADE U     ON PR.COD_UNIDADE   = U.COD_UNIDADE
    JOIN TB_PEDIDO PE    ON I.NUM_PEDIDO     = PE.NUM_PEDIDO
WHERE PE.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31';
```

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 **TB_EMPREGADO**, 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 TB_EMPREGADO E JOIN TB_EMPREGADO S
    ON E.COD_SUPERVISOR = S.CODFUN;
```

1.3. OUTER JOIN

A cláusula **INNER JOIN**, vista anteriormente, tem como característica retornar apenas as linhas em que o campo de relacionamento exista 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 TB_EMPREGADO; -- retorna 61 linhas
--

SELECT -- retorna 58 linhas
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM TB_EMPREGADO E
    INNER JOIN TB_CARGO 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 TB_EMPREGADO E
    JOIN TB_CARGO C ON E.COD_CARGO = C.COD_CARGO;

/*
Existem 61 linhas na tabela TB_EMPREGADO, mas quando fazemos
INNER JOIN com TB_CARGO, retorna apenas com 58 linhas.
A explicação para isso é que existem 3 linhas em TB_EMPREGADO
com COD_CARGO inválido, inexistente em TB_DEPARTAMENTO.
*/
```

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 (TB_EMPREGADO)
*/
-- 
SELECT -- retorna 61 linhas
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, C.CARGO
FROM TB_EMPREGADO E
    LEFT OUTER JOIN TB_CARGO 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 TB_EMPREGADO E
    LEFT JOIN TB_CARGO 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 **TB_EMPREGADO**, 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 TB_EMPREGADO E
    LEFT JOIN TB_CARGO 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 **TB_DEPARTAMENTO**:

```
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO
    FROM TB_EMPREGADO E RIGHT JOIN TB_DEPARTAMENTO 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 TB_EMPREGADO E RIGHT JOIN TB_DEPARTAMENTO 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 TB_EMPREGADO E FULL JOIN TB_CARGO 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 TB_EMPREGADO E FULL JOIN TB_CARGO C ON E.COD_CARGO = C.COD_CARGO
    WHERE E.COD_CARGO IS NULL OR C.COD_CARGO IS NULL;
```

1.4. 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**:

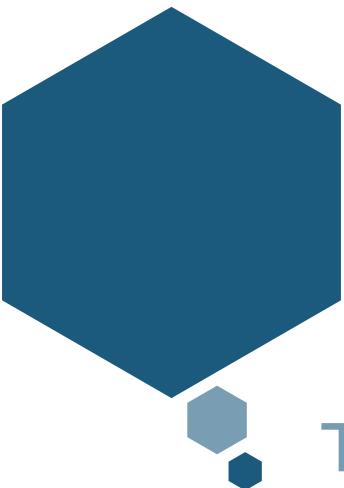
```
-- CROSS JOIN
SELECT -- retorna 854 linhas
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO
FROM TB_EMPREGADO E CROSS JOIN TB_DEPARTAMENTO D;
```

 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 da leitura.

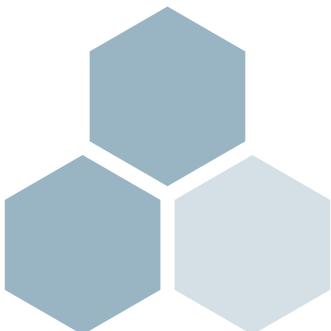
- A associação de tabelas pode ser realizada, por exemplo, para converter em informação os dados encontrados em duas ou mais tabelas. As tabelas podem ser combinadas por meio de uma condição ou um grupo de condições de junção;
- É importante ressaltar que as tabelas devem ser associadas em pares, embora 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**.



Associando tabelas

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 20 a 22.



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 TB_EMPREGADO E JOIN TB_CARGO 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 TB_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 TB_ITENSPEDIDO I  
    JOIN TB_PRODUTO PR    ON I.ID_PRODUTO      = PR.ID_PRODUTO  
    JOIN TB_COR CR        ON I.CODCOR         = CR.CODCOR  
    JOIN TB_TIPOPRODUTO T ON PR.COD_TIPO     = T.COD_TIPO  
    JOIN TB_UNIDADE U      ON PR.COD_UNIDADE = U.COD_UNIDADE  
    JOIN TB_PEDIDO PE      ON I.NUM_PEDIDO   = PE.NUM_PEDIDO  
WHERE PE.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31';
```

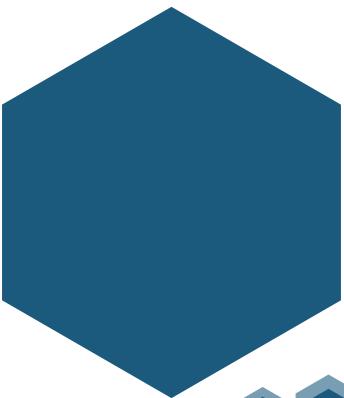
- a) A sintaxe está errada.
- b) Existem muitas tabelas e o SQL não consegue resolver a consulta.
- c) O comando executa um JOIN com várias tabelas e retorna as informações.
- d) Não é necessário utilizar a tabela TB_PEDIDO, pois não é utilizado nenhum campo dela.
- e) O comando executa um JOIN com várias tabelas, porém não retorna as informações.

4. Para trazer todas as informações da tabela TABELA_A, independentemente da relação, qual comando está correto?

- 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.



Associando tabelas



Mãos à obra!

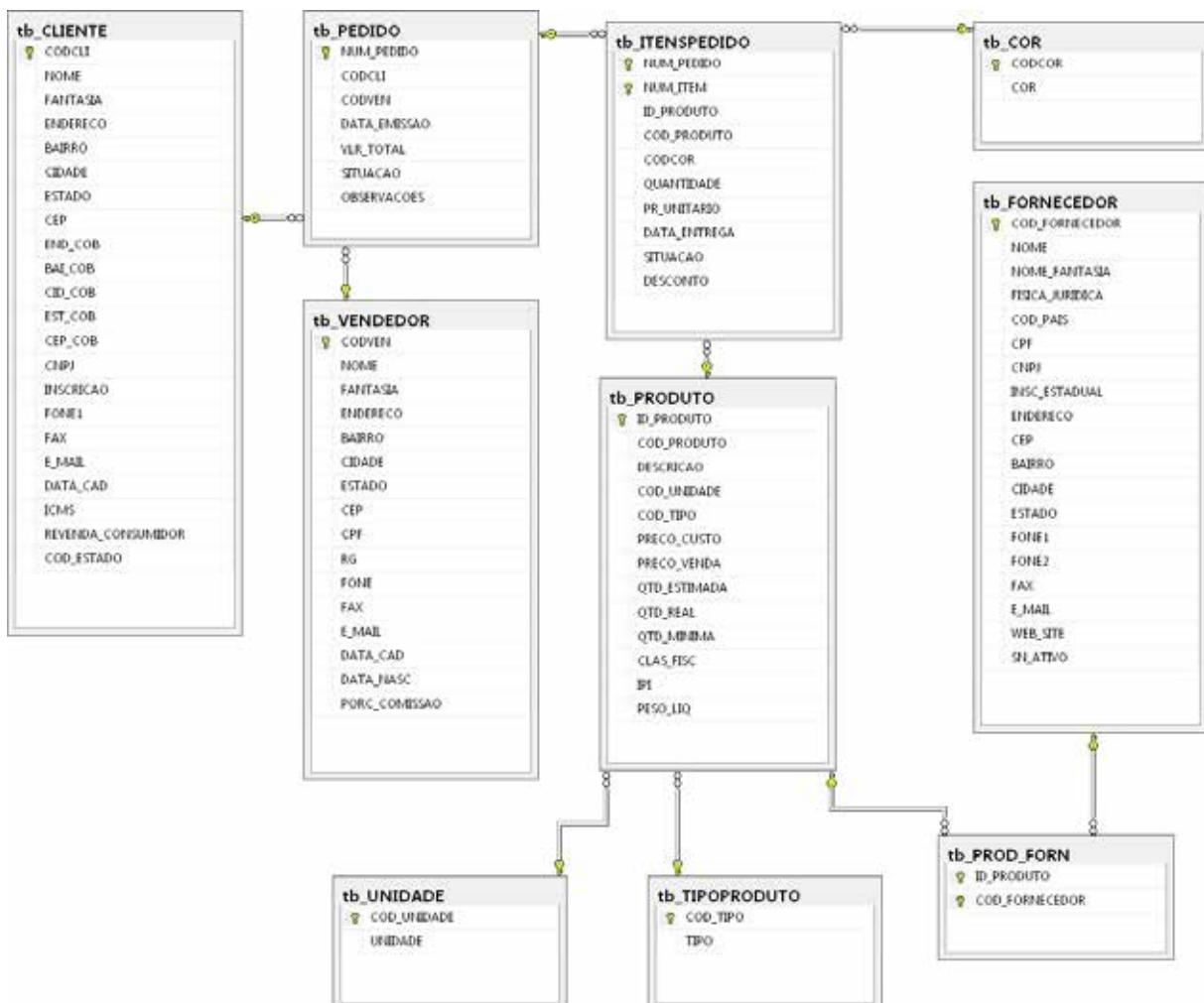
Este laboratório refere-se ao conteúdo das Aulas 20 a 22.



Laboratório 1

A – Utilizando o comando JOIN para associar tabelas

Considere o seguinte diagrama relacional de banco de dados para associar tabelas:

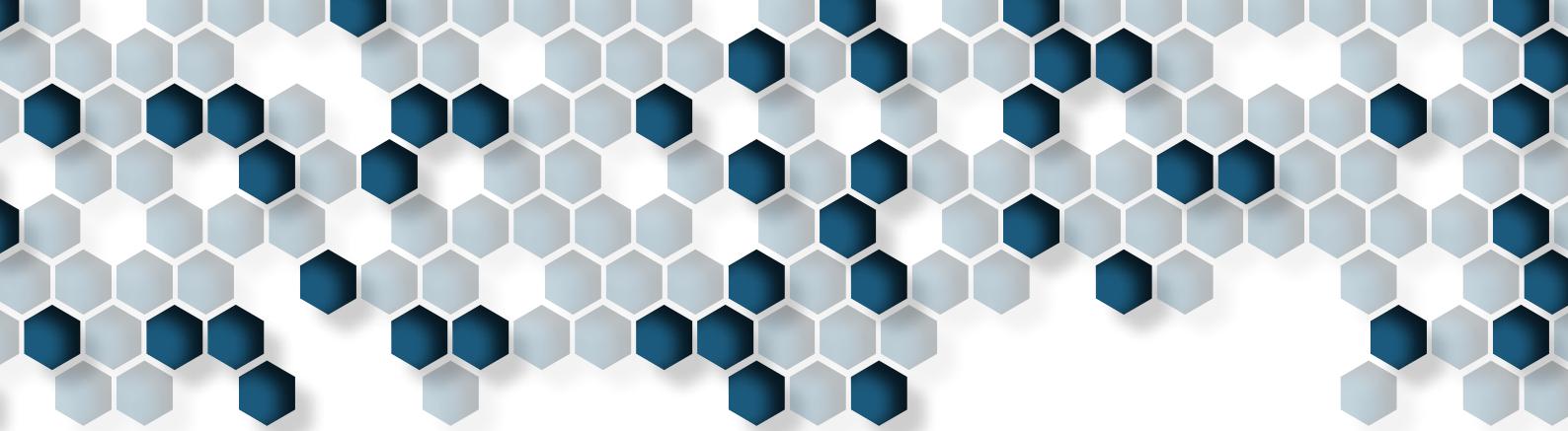


1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste os campos **NUM_PEDIDO**, **DATA_EMISSAO** e **VLR_TOTAL** de **PEDIDOS**, seguidos de **NOME** do vendedor;
3. Liste os campos **NUM_PEDIDO**, **DATA_EMISSAO** e **VLR_TOTAL** de **PEDIDOS**, seguidos de **NOME** do cliente;
4. Liste os pedidos com o nome do vendedor e o nome do cliente;
5. Liste os itens de pedido (**TB_ITENSPEDEDIDO**) com o nome do produto (**TB_PRODUTO**.**DESCRICAO**);
6. Liste os campos **COD_PRODUTO** e **DESCRICAO** da tabela **TB_PRODUTO**, seguidos da descrição do tipo de produto (**TB_TIPOPRODUTO.TIPO**);

7. Liste os campos **COD_PRODUTO** e **DESCRICAO** da tabela **TB_PRODUTO**, seguidos da descrição do tipo de produto (**TB_TIOPRODUTO.TIPO**) e do nome da unidade de medida (**TB_UNIDADE.UNIDADE**);
8. Liste os campos **NUM_PEDIDO**, **NUM_ITEM**, **COD_PRODUTO**, **QUANTIDADE** e **PR_UNITARIO** da tabela **TB_ITENSPEDIDO**, e os campos **COD_PRODUTO** e **DESCRICAO** da tabela **TB_PRODUTO**, seguidos da descrição do tipo de produto (**TB_TIOPRODUTO.TIPO**) e do nome da unidade de medida (**TB_UNIDADE.UNIDADE**);
9. Liste os campos **NUM_PEDIDO**, **NUM_ITEM**, **COD_PRODUTO**, **QUANTIDADE** e **PR_UNITARIO** da tabela **TB_ITENSPEDIDO**, e os campos **COD_PRODUTO** e **DESCRICAO** da tabela **TB_PRODUTO**, seguidos da descrição do tipo de produto (**TB_TIOPRODUTO.TIPO**), do nome da unidade de medida (**TB_UNIDADE.UNIDADE**) e do nome da cor (**TB_COR.COR**);
10. Liste todos os pedidos (**TB_PEDIDO**) do vendedor **MARCELO** em Jan/2014;

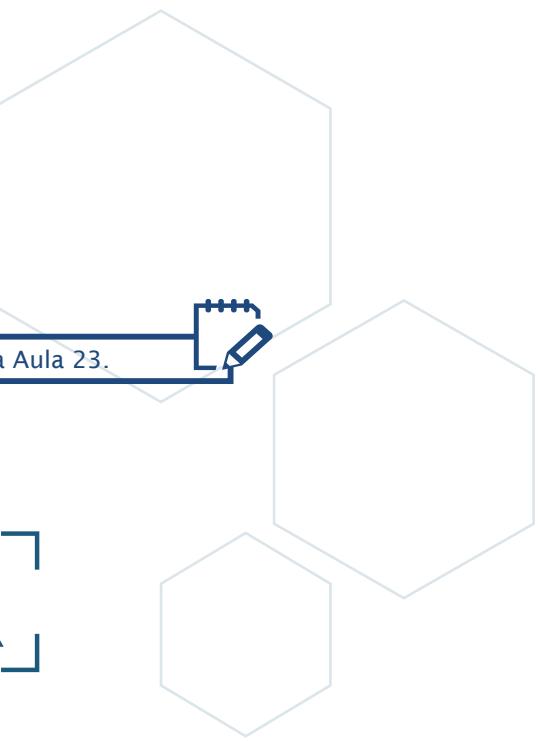
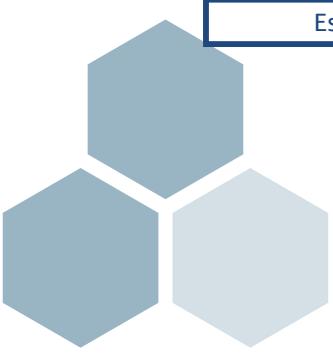
Este exercício não especifica quais campos devem ser exibidos. Escolha você os campos que devem ser mostrados.

11. Liste os nomes dos clientes (**TB_CLIENTE.NOME**) que efetuaram compras em janeiro de 2014;
12. Liste os nomes de produtos (**TB_PRODUTO.DESCRIÇÃO**) que foram vendidos em janeiro de 2014;
13. Liste **NUM_PEDIDO**, **VLR_TOTAL(PEDIDOS)** e **NOME(TB_CLIENTE)**. Mostre apenas pedidos de janeiro de 2014 e clientes que tenham **NOME** iniciado com **MARCIO**;
14. Liste **NUM_PEDIDO**, **QUANTIDADE** vendida e **PR_UNITARIO(TB_ITENSPEDIDO)**, **DESCRICAO(TB_PRODUTO)**, **NOME** do vendedor que vendeu cada item de pedido (**TB_VENDEDOR**);
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 2014 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**.



Manipulando dados com junções

- ◆ UPDATE com subqueries;
- ◆ DELETE com subqueries;
- ◆ UPDATE com JOIN;
- ◆ DELETE com JOIN.



Esta Leitura Complementar refere-se ao conteúdo da Aula 23.

1.1. 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 TB_EMPREGADO  
SET SALARIO = SALARIO * 1.10  
WHERE COD_DEPTO = (SELECT COD_DEPTO FROM TB_DEPARTAMENTO  
WHERE DEPTO = 'CPD');  
  
UPDATE TB_PRODUTO SET PRECO_VENDA = PRECO_CUSTO * 1.2  
WHERE COD_TIPO = (SELECT COD_TIPO FROM TB_TIPOPRODUTO  
WHERE TIPO = 'REGUA');
```

1.2. 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 2013:

```
DELETE FROM TB_PEDIDO  
WHERE DATA_EMISSAO BETWEEN '2013.12.1' AND '2013.12.15' AND  
CODVEN = (SELECT CODVEN FROM TB_VENDEDOR WHERE NOME = 'MAR-  
CELO');
```

1.3. 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 TB_EMPREGADO  
SET SALARIO *= 1.10  
FROM TB_EMPREGADO E JOIN TB_DEPARTAMENTO 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 TB_PRODUTO SET PRECO_VENDA = PRECO_CUSTO * 1.2  
FROM TB_PRODUTO P JOIN TB_TIPOPRODUTO T ON P.COD TIPO = T.COD TIPO  
WHERE T.TIPO = 'REGUA';
```

1.4. 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 2013:

```
DELETE FROM TB_PEDIDO  
FROM TB_PEDIDO P JOIN TB_VENDEDOR V ON P.CODVEN = V.CODVEN  
WHERE P.DATA_EMISSAO BETWEEN '2013.12.1' AND '2013.12.15' AND  
V.NOME = 'MARCELO';
```

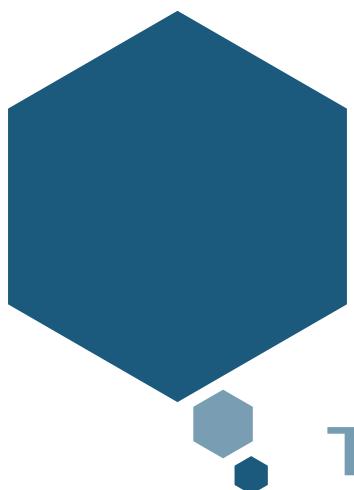


Pontos principais

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

- 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**.





Manipulando dados com junções

Teste seus conhecimentos

Estes testes referem-se ao conteúdo da Aula 23.



1. Analise o comando a seguir e verifique qual afirmação é a correta:

```
UPDATE TB_EMPREGADO  
SET SALARIO = SALARIO * 1.10  
WHERE COD_DEPTO = (SELECT COD_DEPTO FROM TB_DEPARTAMENTO  
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 salário 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 adiante e verifique qual afirmação é a correta:

```
DELETE FROM TB_PEDIDO  
WHERE DATA_EMISSAO BETWEEN '2013.12.1' AND '2013.12.15' AND  
CODVEN = (SELECT CODVEN FROM TB_VENDEDOR WHERE NOME = 'MAR-  
CELO');
```

- 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 2013.

3. Analise o comando a seguir e verifique qual afirmação é a correta:

```
UPDATE TB_EMPREGADO  
SET SALARIO *= 1.10  
FROM TB_EMPREGADO E JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_  
DEPTO  
WHERE D.DEPTO = 'CPD';
```

- a) Os empregados do CPD terão um aumento de 10%.
- b) Os empregados do CPD terão um aumento de 110%.
- c) Somente os empregados do CPD terão aumento, inclusive os que possuírem COD_DEPTO nulo.
- d) Nenhum empregado terá aumento.
- e) Deve ser utilizada uma subquery em vez de JOIN.

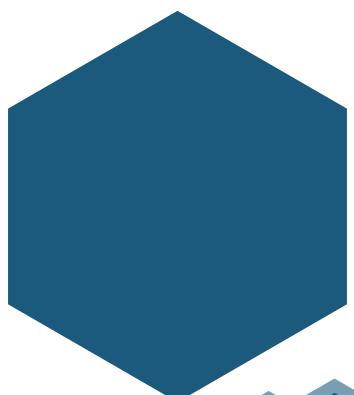
4. Analise o comando adiante e verifique qual afirmação é a correta:

```
TRUNCATE FROM TB_PEDIDO  
FROM TB_PEDIDO P JOIN TB_VENDEDOR V ON P.CODVEN = V.CODVEN  
WHERE P.DATA_EMISSAO BETWEEN '2013.12.1' AND '2013.12.15' 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 2013.

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 delas.
- d) As subqueries podem simplificar o código de atualização.
- e) Utiliza-se a instrução UPDATE para atualizar uma tabela com informações de outra tabela.

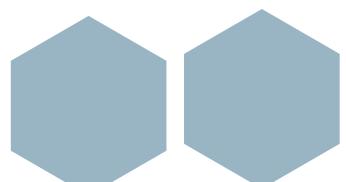


Manipulando dados com junções



Mãos à obra!

Este laboratório refere-se ao conteúdo da Aula 23.



Laboratório 1

A – Atualizando tabelas com associações e subqueries

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Altere a tabela **TB_CARGO**, 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 **TB_EMPREGADO** o salário inicial cadastrado no cargo correspondente da **TB_CARGO**;
5. Reajuste os preços de venda de todos os produtos de modo que fiquem 30% acima do preço de custo (**PRECO_VENDA = 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:
 - **PRECO_VENDA = PRECO_CUSTO * 1.4;**
 - Para produtos com **TB_TIPOPRODUTO.TIPO = 'REGUA'**;
 - É necessário fazer um **JOIN** de **TB_PRODUTO** com **TB_TIPOPRODUTO**.
8. Altere a tabela **TB_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 2014;
9. Altere o campo **ICMS** da tabela **TB_CLIENTE** para 12. Considere apenas clientes dos estados: RJ, RO, AC, RR, MG, PR, SC, RS, MS e MT;

Manipulando dados com junções

Aula 23

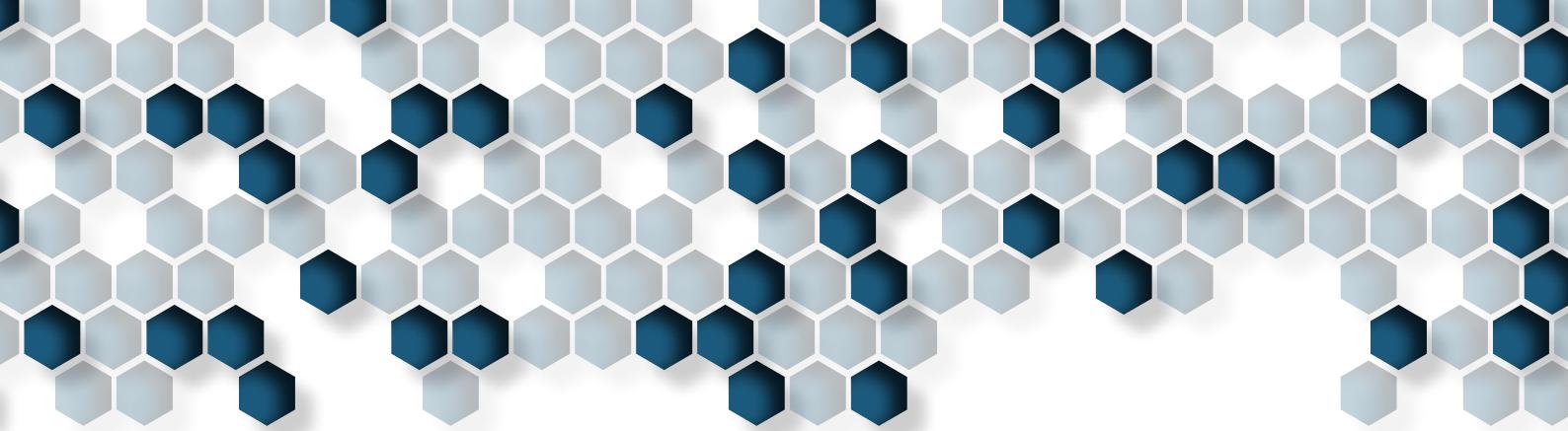
10. Altere o campo **ICMS** para 18, apenas para clientes de SP;
11. Altere o campo **ICMS** da tabela **TB_CLIENTE** para 7. Considere apenas clientes que não sejam dos estados: RJ, RO, AC, RR, MG, PR, SC, RS, MS, MT e SP;
12. Crie a tabela **ESTADOS** com os respectivos campos:
 - **COD_ESTADO**: Inteiro, 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 TB_CLIENTE  
WHERE ESTADO IS NOT NULL
```

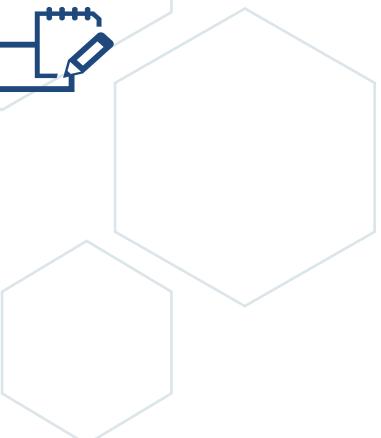
O **SELECT** deve retornar 21 linhas e não repetir o Estado. Se o resultado for diferente, é porque os **UPDATES** de **ICMS** estão incorretos.

14. Crie o campo **COD_ESTADO** na tabela **TB_CLIENTE**;
15. Copie para **TB_CLIENTE.COD_ESTADO** o código do Estado gerado na tabela **ESTADOS**.



Consultas com subqueries

- 
- 
- 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.



Esta Leitura Complementar refere-se ao conteúdo das Aulas 24 e 25.

1.1. 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.

1.2. 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 2014:

```
SELECT * FROM TB_CLIENTE
WHERE EXISTS (SELECT * FROM TB_PEDIDO
    WHERE CODCLI = TB_CLIENTE.CODCLI AND
        DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31');
-- OU
SELECT * FROM TB_CLIENTE
WHERE CODCLI IN (SELECT CODCLI FROM TB_PEDIDO
    WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31');
```

No próximo exemplo, é verificada a existência de clientes que não compraram em janeiro de 2014:

```
SELECT * FROM TB_CLIENTE
WHERE NOT EXISTS (SELECT * FROM TB_PEDIDO
    WHERE CODCLI = TB_CLIENTE.CODCLI AND
        DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31');
-- OU
SELECT * FROM TB_CLIENTE
WHERE CODCLI NOT IN (SELECT CODCLI FROM TB_PEDIDO
    WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31');
```

1.3. 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 TB_EMPREGADO
WHERE COD_CARGO IN (SELECT COD_CARGO FROM TB_CARGO
    WHERE SALARIO_INIC < 5000);
```

- **Exemplo 2**

```
-- Lista de departamentos em que não existe nenhum
-- funcionário cadastrado
SELECT * FROM TB_DEPARTAMENTO
WHERE COD_DEPTO NOT IN
    (SELECT DISTINCT COD_DEPTO FROM TB_EMPREGADO
     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 TB_EMPREGADO E RIGHT JOIN TB_DEPARTAMENTO 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 TB_CARGO
WHERE COD_CARGO NOT IN
    (SELECT DISTINCT COD_CARGO FROM TB_EMPREGADO
     WHERE COD_CARGO IS NOT NULL)
-- O mesmo que
SELECT
    C.COD_CARGO, C.CARGO
FROM TB_EMPREGADO E RIGHT JOIN TB_CARGO C ON E.COD_CARGO = C.COD_
CARGO
WHERE E.COD_CARGO IS NULL
```

1.4. 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 TB_EMPREGADO
WHERE SALARIO = (SELECT MIN(SALARIO) FROM TB_EMPREGADO)
-- o mesmo que
SELECT TOP 1 WITH TIES * FROM TB_EMPREGADO
WHERE SALARIO IS NOT NULL
ORDER BY SALARIO
```

- Exemplo 2

```
-- Funcionário mais novo na empresa
SELECT * FROM TB_EMPREGADO
WHERE DATA_ADMISSAO =
    (SELECT MAX(DATA_ADMISSAO) FROM TB_EMPREGADO);

-- O mesmo que
SELECT TOP 1 WITH TIES * FROM TB_EMPREGADO
ORDER BY DATA_ADMISSAO DESC;
```

1.5. 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 TB_EMPREGADO SET SALARIO = (SELECT SALARIO_INIC
FROM TB_CARGO
WHERE COD_CARGO = TB_EMPREGADO.COD_CARGO);
```

1.5.1. 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 2014:

```
SELECT * FROM TB_CLIENTE
WHERE NOT EXISTS (SELECT * FROM TB_PEDIDO
    WHERE CODCLI = TB_CLIENTE.CODCLI AND
        DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31');
```

1.6. 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 2014 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 TB_PEDIDO
      WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31') AS POR-
CENTAGEM
  FROM TB_PEDIDO P JOIN TB_VENDEDOR V ON P.CODVEN = V.CODVEN
 WHERE P.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
 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 TB_EMPREGADO E
      WHERE E.SINDICALIZADO = 'S' AND
            E.COD_DEPTO = TB_EMPREGADO.COD_DEPTO) AS TOT_SALARIO_SIND,
       (SELECT SUM(E.SALARIO) FROM TB_EMPREGADO E
      WHERE E.SINDICALIZADO = 'N' AND
            E.COD_DEPTO = TB_EMPREGADO.COD_DEPTO) AS TOT_SALARIO_NAO_-
SIND
  FROM TB_EMPREGADO
 GROUP BY COD_DEPTO;
```

1.7. 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 2014, mas compraram em dezembro de 2013:

```
SELECT * FROM TB_PEDIDO
WHERE CODVEN IN (SELECT CODVEN FROM TB_VENDEDOR WHERE NOME = 'LEIA')
AND CODCLI IN (
    SELECT CODCLI FROM TB_CLIENTE
    WHERE CODCLI NOT IN (SELECT CODCLI FROM TB_PEDIDO
        WHERE DATA_EMISSAO BETWEEN
        '2014.1.1' AND '2014.1.31') AND
    CODCLI IN (SELECT CODCLI FROM TB_PEDIDO
        WHERE DATA_EMISSAO BETWEEN
        '2013.12.1' AND '2013.12.31')
    AND ESTADO = 'SP' );
```

Consultas com subqueries

Aulas 24 e 25

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 TB_VENDEDOR WHERE NOME = 'LEIA';

-- Tabela temporária 2 - Clientes que compraram em Jan/2014
SELECT CODCLI INTO #CLI_COM_PED_JAN_2014 FROM TB_PEDIDO
WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31';

-- Tabela temporária 3 - Clientes que compraram em Dez/2013
SELECT CODCLI INTO #CLI_COM_PED_DEZ_2013 FROM TB_PEDIDO
WHERE DATA_EMISSAO BETWEEN '2013.12.1' AND '2013.12.31';

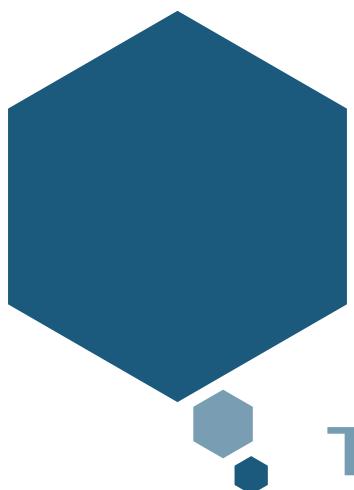
-- Tabela temporária 4 - Clientes de SP que compraram
-- em Dez/2013, mas não compraram em Jan de 2014
SELECT CODCLI INTO #CLI_FINAL FROM TB_CLIENTE
WHERE CODCLI NOT IN (SELECT CODCLI FROM #CLI_COM_PED_JAN_2014)
AND
CODCLI IN (SELECT CODCLI FROM #CLI_COM_PED_DEZ_2013)
AND
ESTADO = 'SP';

-- SELECT de TB_PEDIDO
SELECT * FROM TB_PEDIDO
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 da leitura.

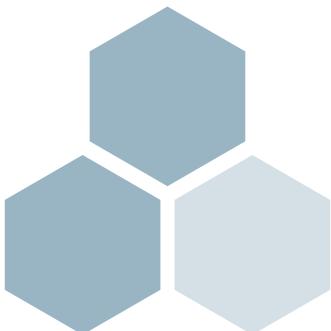
- Uma consulta aninhada em uma instrução **SELECT, INSERT, DELETE ou UPDATE** é denominada subquery (subconsulta). As subqueries são também referidas como queries internas. Já a instrução em que está inserida a subquery pode ser chamada de query externa;
- Vejamos algumas das diversas características 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

Estes testes referem-se ao conteúdo das Aulas 24 e 25.



1. Qual das afirmações a seguir não é uma característica das subqueries?

- a) A utilização da cláusula ORDER BY só é possível caso a cláusula TOP seja especificada.
- b) Subqueries podem ser utilizadas para realizar testes de existência de linhas. Nesse caso, é adotado o operador EXISTS.
- c) É possível obter mais de uma coluna por subquery.
- 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 adiante e verifique qual afirmação é a correta:

```
SELECT * FROM TB_CLIENTE  
WHERE CODCLI IN (SELECT CODCLI FROM PEDIDOS  
                  WHERE DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31');
```

- a) Apresenta os clientes que compraram em janeiro de 2014.
- b) Apresenta os clientes que não compraram em janeiro de 2014.
- 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 2014.

3. Analise o comando adiante e verifique qual afirmação é a correta:

```
SELECT * FROM TB_EMPREGADO  
WHERE COD_CARGO = (SELECT COD_CARGO FROM TABELACAR  
                     WHERE SALARIO_INIC < 5000);
```

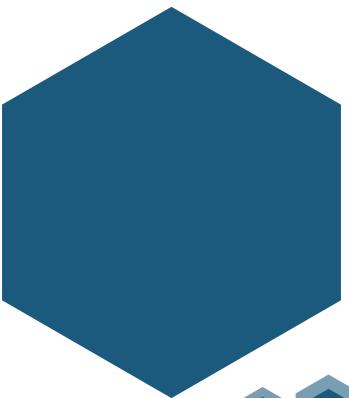
- a) A sintaxe está errada.
- b) O comando retornará todos os empregados que possuem cargo com salário inicial menor que 5000.
- c) O comando não retorna nenhum registro.
- d) Não existe diferença entre o operador IN e igual.
- e) Ocorrerá um erro quando a subconsulta retornar mais de um registro.

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 correlacionadas não podem incluir funções definidas pelo usuário.
- d) A cláusula EXISTS não retorna dados, mas apenas TRUE ou FALSE.
- e) A subquery será executada para cada linha da consulta externa.

5. Qual afirmação está errada, com relação a tabelas temporárias?

- a) Quando declaramos com sinal #, a tabela é global.
- b) Normalmente, uma subquery é mais fácil de manter do que uma tabela temporária.
- 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!

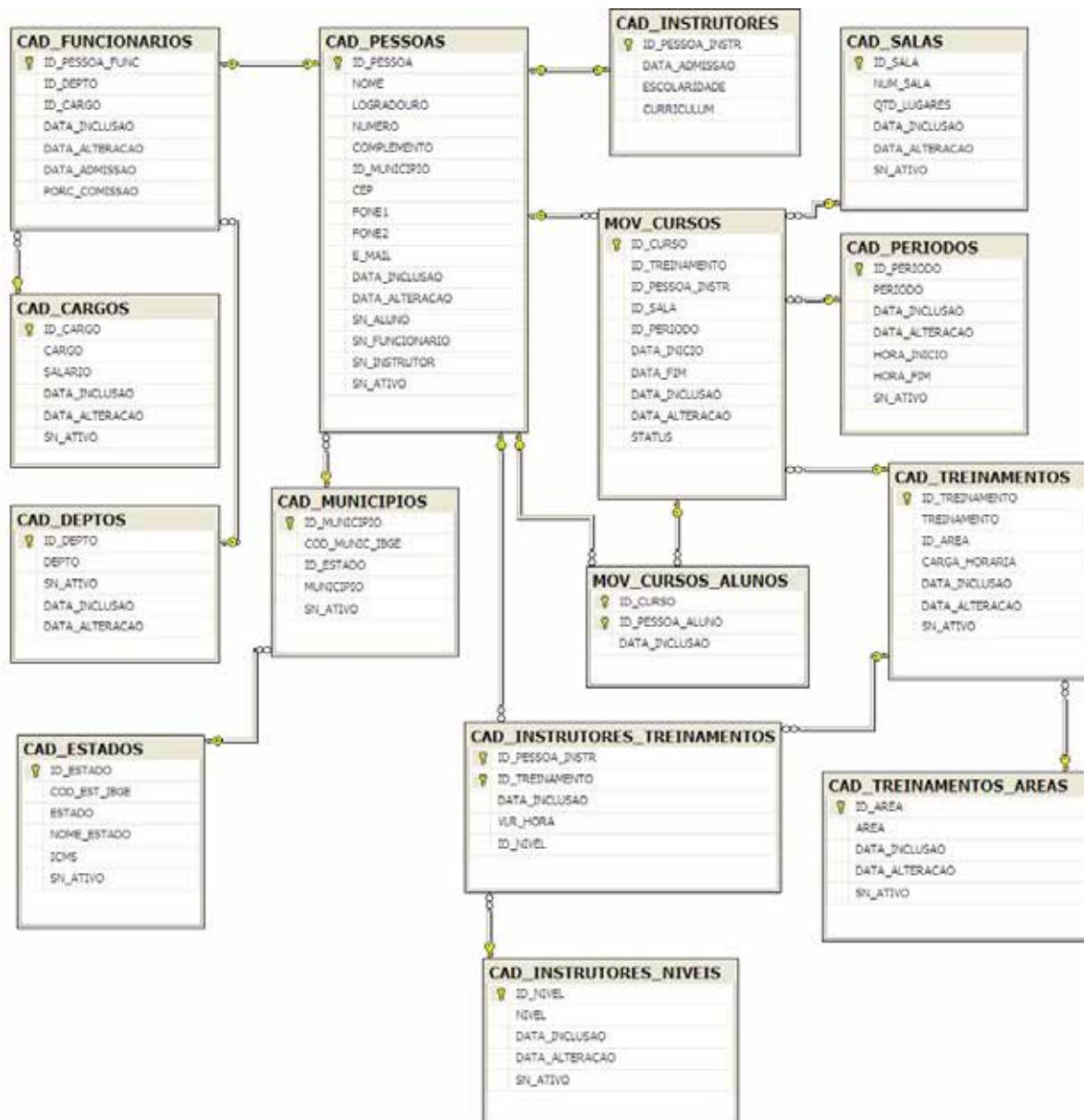
Este laboratório refere-se ao conteúdo das Aula 24 e 25.



Laboratório 1

A – Trabalhando com JOIN e subquery

Para auxiliar na execução do laboratório, utilize o diagrama a seguir:



Consultas com subqueries

Aulas 24 e 25

1. Execute o script: **Cap06_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**).



Agrupando dados

- ◆ Funções de agregação;
- ◆ GROUP BY.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 26 e 27.



1.1. Introdução

Nesta leitura, 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**.

1.2. 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.

1.2.1. 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 TB_EMPREGADO
SELECT AVG(SALARIO) AS SALARIO_MEDIO
FROM TB_EMPREGADO;
-- neste caso, o grupo corresponde aos empregados com
-- COD_DEPTO = 2
SELECT AVG(SALARIO) AS SALARIO_MEDIO FROM TB_EMPREGADO
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 TB_EMPREGADO
SELECT COUNT(*) AS QTD_EMPREGADOS
FROM TB_EMPREGADO;
-- neste caso, o grupo corresponde aos empregados com
-- COD_DEPTO = 2
SELECT COUNT(COD_DEPTO) AS QTD_EMPREGADOS FROM TB_EMPREGADO
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 TB_EMPREGADO
SELECT MIN(SALARIO) AS MENOR_SALARIO FROM TB_EMPREGADO;
-- neste caso, o grupo corresponde aos empregados com
-- COD_DEPTO = 2
SELECT MIN(SALARIO) AS MENOR_SALARIO FROM TB_EMPREGADO
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 TB_EMPREGADO
SELECT MAX(SALARIO) AS MAIOR_SALARIO
FROM TB_EMPREGADO;
-- neste caso, o grupo corresponde aos empregados com
-- COD_DEPTO = 2
SELECT MAX(SALARIO) AS MAIOR_SALARIO FROM TB_EMPREGADO
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 TB_EMPREGADO  
SELECT SUM(SALARIO) AS SOMA_SALARIOS  
FROM TB_EMPREGADO;  
-- neste caso, o grupo corresponde aos empregados com  
-- COD_DEPTO = 2  
SELECT SUM(SALARIO) AS SOMA_SALARIOS FROM TB_EMPREGADO  
WHERE COD_DEPTO = 2
```

1.3. 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>**.

Observe o resultado da instrução:

```
SELECT COD_DEPTO, SALARIO FROM TB_EMPREGADO 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 TB_EMPREGADO
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 TB_EMPREGADO E
    JOIN TB_DEPARTAMENTO 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 TB_EMPREGADO E
    JOIN TB_DEPARTAMENTO 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 2014
SELECT TOP 10 C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRA
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI
WHERE P.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
GROUP BY C.CODCLI, C.NOME
ORDER BY TOT_COMPRA DESC;
```

1.3.1. 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 2014. Veremos que
-- todas as linhas do resultado terão um total não nulo.
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRA
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI
WHERE P.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
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_COMPRA
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI
WHERE P.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
GROUP BY ALL C.CODCLI, C.NOME;
```

1.3.2. 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 TB_EMPREGADO E
JOIN TB_DEPARTAMENTO 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 2014:

```
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRA  
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI  
WHERE P.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'  
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 2014:

```
SELECT C.CODCLI, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRA  
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI  
WHERE P.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'  
GROUP BY ALL C.CODCLI, C.NOME HAVING SUM(P.VLR_TOTAL) IS NULL;
```

1.3.3. 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 **TB_VENDEDOR**) realizaram para cada cliente (tabela **TB_CLIENTE**) no primeiro semestre de 2013:

```

SELECT
    V.NOME AS VENDEDOR, C.NOME AS CLIENTE,
    T.TIPO AS TIPO_PRODUTO, SUM( I.QUANTIDADE ) AS QTD_TOT
FROM
    TB_PEDIDO Pe
    JOIN TB_CLIENTE C ON Pe.CODCLI = C.CODCLI
    JOIN TB_VENDEDOR V ON Pe.CODVEN = V.CODVEN
    JOIN TB_ITENSPEDIDO I ON Pe.NUM_PEDIDO = I.NUM_PEDIDO
    JOIN TB_PRODUTO Pr ON I.ID_PRODUTO = Pr.ID_PRODUTO
    JOIN TB_TIPOPRODUTO T ON Pr.COD_TIPO = T.COD_TIPO
WHERE Pe.DATA_EMISSAO BETWEEN '2013.1.1' AND '2013.6.30'
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

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    TB_PEDIDO Pe
    JOIN TB_CLIENTE C ON Pe.CODCLI = C.CODCLI
    JOIN TB_VENDEDOR V ON Pe.CODVEN = V.CODVEN
    JOIN TB_ITENSPEDIDO I ON Pe.NUM_PEDIDO = I.NUM_PEDIDO
    JOIN TB_PRODUTO Pr ON I.ID_PRODUTO = Pr.ID_PRODUTO
    JOIN TB_TIPOPRODUTO T ON Pr.COD_TIPO = T.COD_TIPO
WHERE Pe.DATA_EMISSAO BETWEEN '2013.1.1' AND '2013.6.30'
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 **CELSON MARTINS**, ou seja, **55912** produtos, independentemente do cliente e do tipo de produto:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
213	CELSON MARTINS	VILSON CORDEIRO DO ROSARIO	YO-YO	516
214	CELSON MARTINS	VILSON CORDEIRO DO ROSARIO	NULL	3266
215	CELSON MARTINS	NULL	NULL	55912
216	DORINHA	3R (ARISTEU,ADALTON)	CANETA	750
217	DORINHA	3R (ARISTEU,ADALTON)	PORTA LAPIS	30

Na última linha do resultado, temos **NULL** nas três 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	PORTA LAPIS	110
1932	MARCELO	WALEU IND E COM DE PLASTICOS LTDA	NULL	805
1933	MARCELO	NULL	NULL	216732
1934	NULL	NULL	NULL	1022534

1.3.4. 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 TB_PEDIDO Pe
JOIN TB_CLIENTE C ON Pe.CODCLI = C.CODCLI
JOIN TB_VENDEDOR V ON Pe.CODVEN = V.CODVEN
JOIN TB_ITENSPEIDO I ON Pe.NUM_PEDIDO = I.NUM_PEDIDO
JOIN TB_PRODUTO Pr ON I.ID_PRODUTO = Pr.ID_PRODUTO
JOIN TB_TIPOPRODUTO T ON Pr.COD_TIPO = T.COD_TIPO
WHERE Pe.DATA_EMISSAO BETWEEN '2013.1.1' AND '2013.6.30'
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

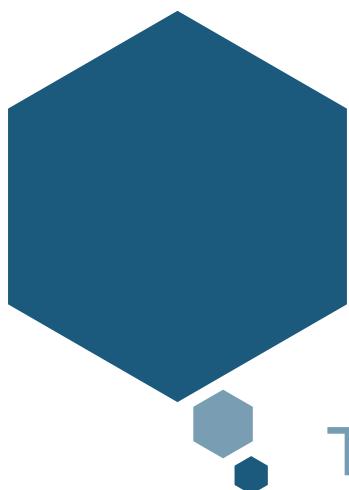
Seguindo a mesma regra, se apenas o **CLIENTE** não é nulo, o total corresponde ao total comprado por este cliente, independentemente de vendedor ou de produto:

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 da leitura.

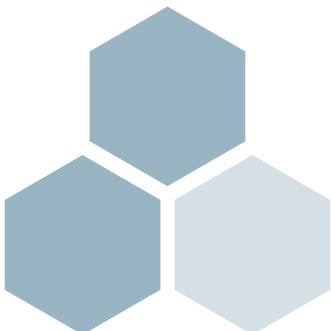
- As funções de agregação fornecidas pelo SQL Server permitem sumarizar dados. Por meio delas, é possível somar valores, calcular média e contar resultados. Os cálculos feitos pelas funções de agregação são feitos com base em um conjunto ou grupo de valores, porém retornam um único valor. Para obter 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.



Agrupando dados

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 26 e 27.



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 TB_EMPREGADO E  
JOIN TB_DEPARTAMENTO 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?

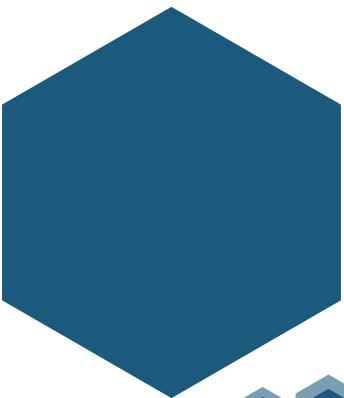
- a) Não há diferença entre a cláusula GROUP BY e a GROUP BY ALL.
- b) GROUP BY apresenta todas as linhas, mesmo não agrupadas.
- c) GROUP BY ALL deve ser utilizada com funções de agrupamento.
- d) Enquanto GROUP BY apresenta somente os grupos selecionados, GROUP BY ALL apresenta todos os grupos mesmo 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 CUBE
- e) Todas as alternativas anteriores estão corretas.



Agrupando dados



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 26 e 27.



Laboratório 1

A – Realizando consultas e ordenando dados

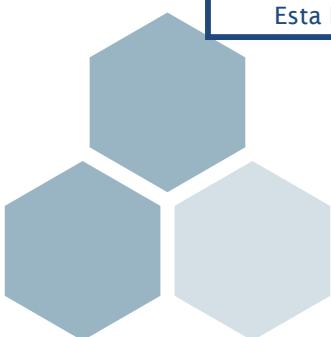
1. Coloque em uso o banco de dados **PEDIDOS**;
2. Calcule a média de preço de venda (**PRECO_VENDA**) do cadastro de **TB_PRODUTO**;
3. Calcule a quantidade de pedidos cadastrados em janeiro de 2014 (o maior e o menor valor total, **VLR_TOTAL**);
4. Calcule o valor total vendido (soma de **TB_PEDIDO.VLR_TOTAL**) em janeiro de 2014;
5. Calcule o valor total vendido pelo vendedor de código 1 em janeiro de 2014;
6. Calcule o valor total vendido pela vendedora **LEIA** em janeiro de 2014;
7. Calcule o valor total vendido pelo vendedor **MARCELO** em janeiro de 2014;
8. Calcule o valor da comissão (soma de **TB_PEDIDO.VLR_TOTAL * TB_VENDEDOR.PORC_COMISSAO/100**) que a vendedora **LEIA** recebeu em janeiro de 2014;
9. Calcule o valor da comissão que o vendedor **MARCELO** recebeu em janeiro de 2014;
10. Liste os totais vendidos por cada vendedor (mostrar **TB_VENDEDOR.NOME** e a soma de **TB_PEDIDO.VLR_TOTAL**) em janeiro de 2014. Deve-se exibir o nome do vendedor;
11. Liste o total comprado por cada cliente em janeiro de 2014. Deve-se mostrar o nome do cliente;
12. Liste o valor e a quantidade total vendida de cada produto em janeiro de 2014;
13. Liste os totais vendidos por cada vendedor em janeiro de 2014. Deve-se 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 2014. Deve-se 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 2014. Deve-se mostrar apenas os produtos que venderam mais de R\$16.000,00;
16. Liste o total comprado por cada cliente em janeiro de 2014. Deve-se mostrar o nome do cliente e somente os 10 primeiros do ranking;
17. Liste o total vendido de cada produto em janeiro de 2014. Deve-se mostrar os 10 produtos que mais venderam;
18. Liste o total vendido em cada um dos meses de 2013.



Comandos adicionais

- Funções de cadeia de caracteres;
- Função CASE;
- Manipulação de campos do tipo datetime;
- Alteração da configuração de idioma a partir do SSMS.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 28 a 30.



1.1. 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');
```

Results	
(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);
```

Results	
(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	
	(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:

d	Dia com 1 ou 2 dígitos.
dd	Dia com 2 dígitos.
ddd	Abreviação do dia da semana.
ddd	Nome do dia da semana.
M	Mês com 1 ou 2 dígitos.
MM	Mês com 2 dígitos.
MMM	Abreviação do nome do mês.
MMM	Nome do mês.
yy	Ano com 2 dígitos.
yyy	Ano com 4 dígitos.
hh	Hora de 1 a 12.
HH	Hora de 0 a 23.
mm	Minutos.
ss	Segundos.
fff	Milésimos de segundo.

Veja um exemplo:

```
SELECT FORMAT (GETDATE(), 'dd/MM/yyyy');
```

Results	
	(No column name)
1	23/09/2013

1.2. 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 **TB_EMPREGADO** 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 TB_EMPREGADO;
```

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 TB_EMPREGADO;
```

1.3. 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.

Vejamos 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
mdy	Mês, dia e ano (Formato padrão americano).
dmy	Dia, mês e ano.
ymd	Ano, mês e dia.
ydm	Ano, dia e mês.
myd	Mês, ano e dia.
dym	Dia, ano e mês.

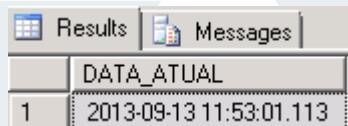
O exemplo a seguir determina o formato ano/mês/dia para o valor de data a ser retornado:

```
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
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 há quantos dias cada funcionário da tabela **TB_EMPREGADO** foi admitido:

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

- **DAY()**

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

```
DAY(data)
```

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

Vejamos 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 TB_EMPREGADO  
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.

Vejamos os exemplos:

```
-- Número do mês correspondente à data de hoje  
SELECT MONTH(GETDATE())  
  
-- Empregados admitidos em dezembro  
SELECT * FROM TB_EMPREGADO  
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 2006:

```
SELECT * FROM TB_EMPREGADO  
WHERE YEAR(DATA_ADMISSAO) = 2006;
```

Já o exemplo a seguir retorna os empregados admitidos no mês de janeiro de 2011:

```
SELECT * FROM TB_EMPREGADO  
WHERE YEAR(DATA_ADMISSAO) = 2011 AND  
MONTH(DATA_ADMISSAO) = 1;
```

- **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 tabela a seguir:

Valor	Parte retornada	Abreviação
year	Ano	yy, yyyy
quarter	Trimestre (1/4 de ano)	qq, q
month	Mês	mm, m
dayofyear	Dia do ano	dy, y
day	Dia	dd, d
week	Semana	wk, ww
weekday	Dia da semana	dw
hour	Hora	hh
minute	Minuto	mi, n
second	Segundo	ss, s
millisecond	Milissegundo	ms
microsecond	Microssegundo	mcs
nanosecond	Nanossegundo	ns
TZoffset	Diferença de fuso-horário	tz
ISO_WEEK	Retorna a numeração da semana associada a um ano	isowk, isoww

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 1996. Para isso, são especificadas as partes de ano (**YEAR**) e mês (**MONTH**):

```
SELECT * FROM TB_EMPREGADO  
WHERE DATEPART(YEAR, DATA ADMISSAO) = 1996 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 TB_EMPREGADO  
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(part, 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());
```

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. Vejamos:

```
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)	10:25:15
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 14:00:15.0000:

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

O resultado é mostrado a seguir:

Results	
	(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	Messages
	(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:

	Results	Messages
	(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]);
```

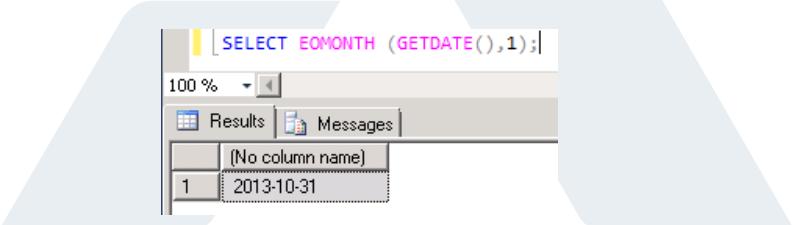
O valor retornado é do tipo data (DATE). No exemplo a seguir, vamos retornar o último dia do mês atual:

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

O resultado desse exemplo é mostrado a seguir:

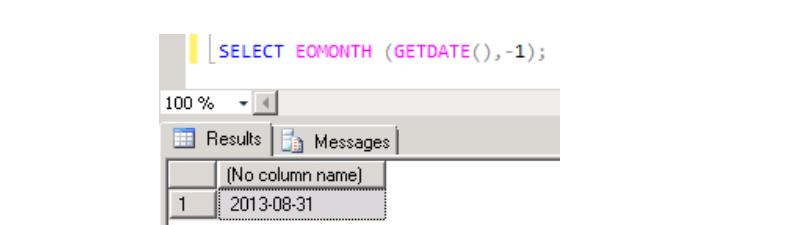
Results	
	(No column name)
1	2013-09-30

Para avançar 1 mês, adicione o valor 1 no parâmetro **adicionar_mês**:



Results	
	(No column name)
1	2013-10-31

Para voltar 1 mês adicione o valor -1 no parâmetro **adicionar_mês**:

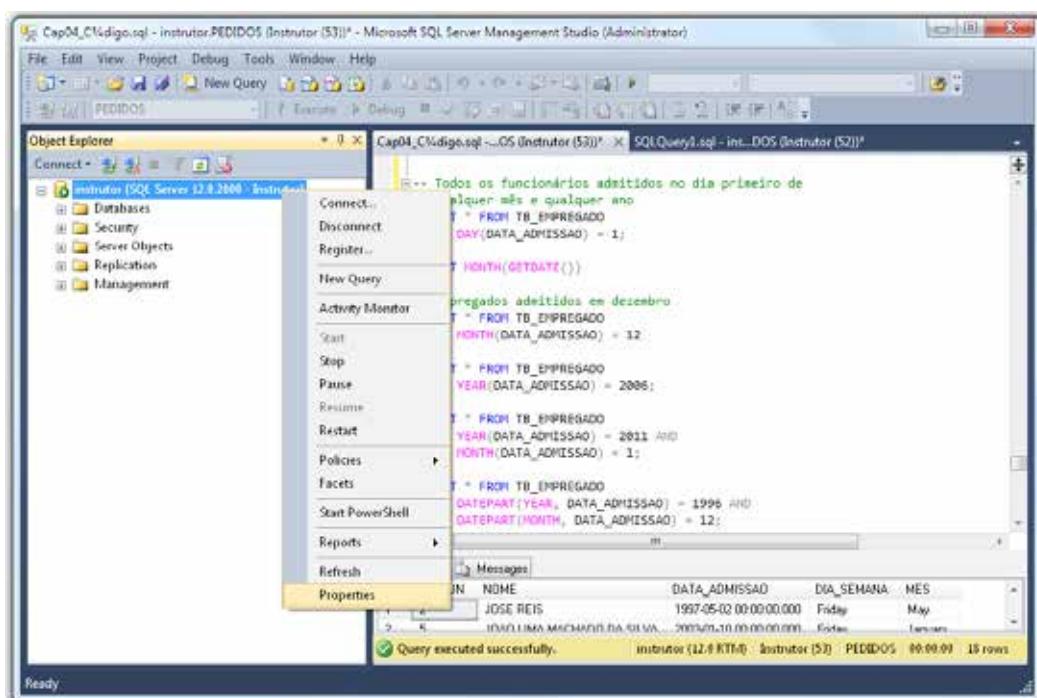


Results	
	(No column name)
1	2013-08-31

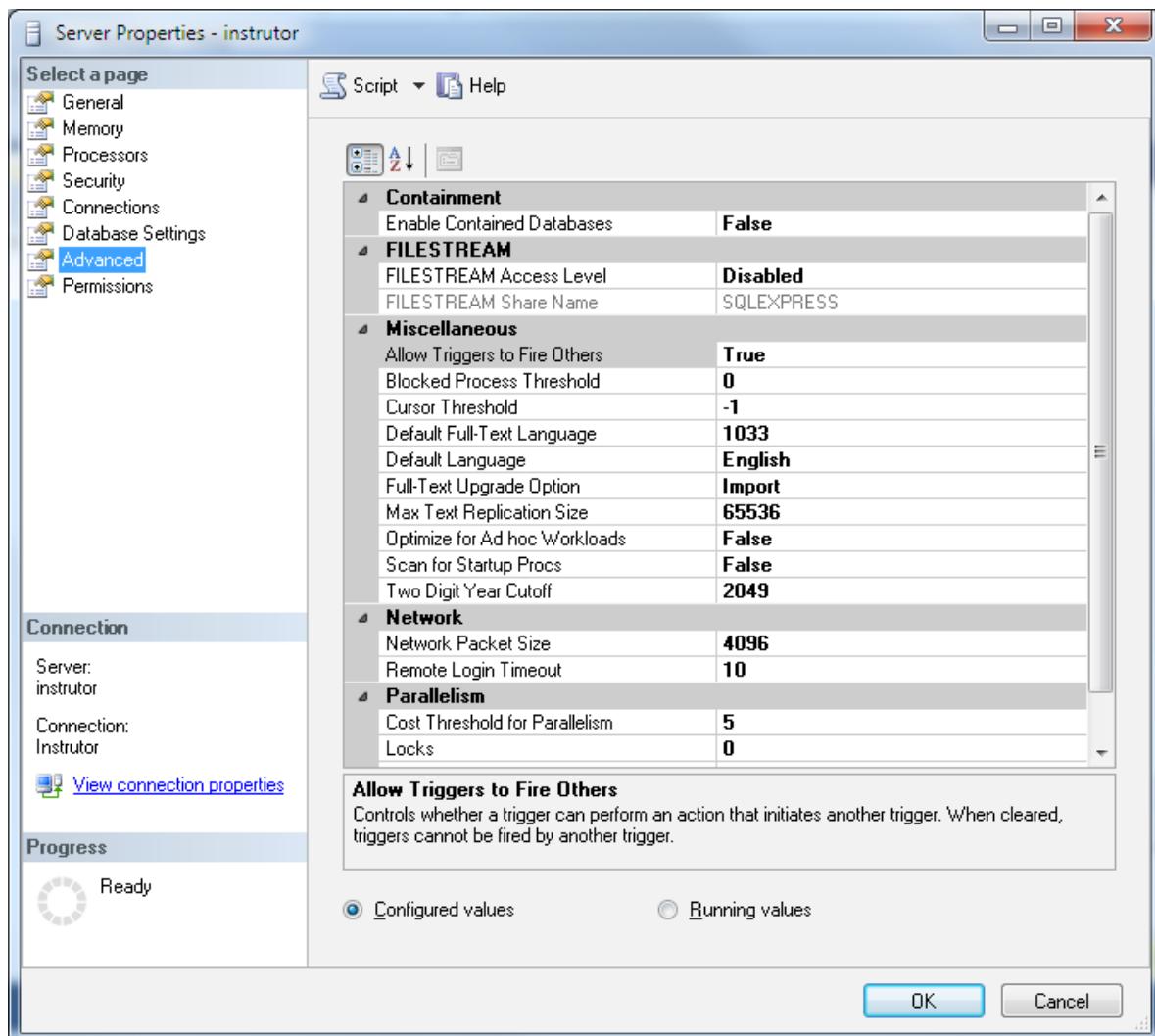
1.4. Alterando a configuração de idioma a partir do SSMS

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:

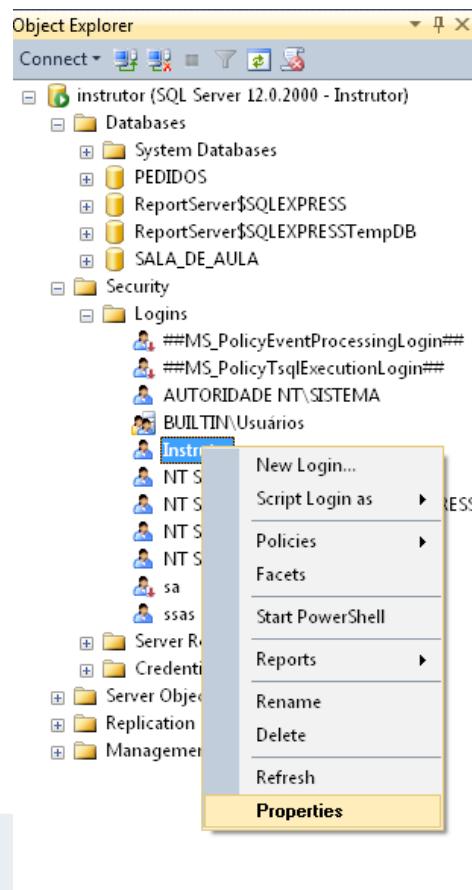
- **Brazilian:** Pode utilizar os formatos dd/mm/yy ou dd/mm/yyyy;
- **English:** Pode utilizar os formatos mm/dd/yy, mm/dd/yyyy ou yyyy.mm.dd.

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

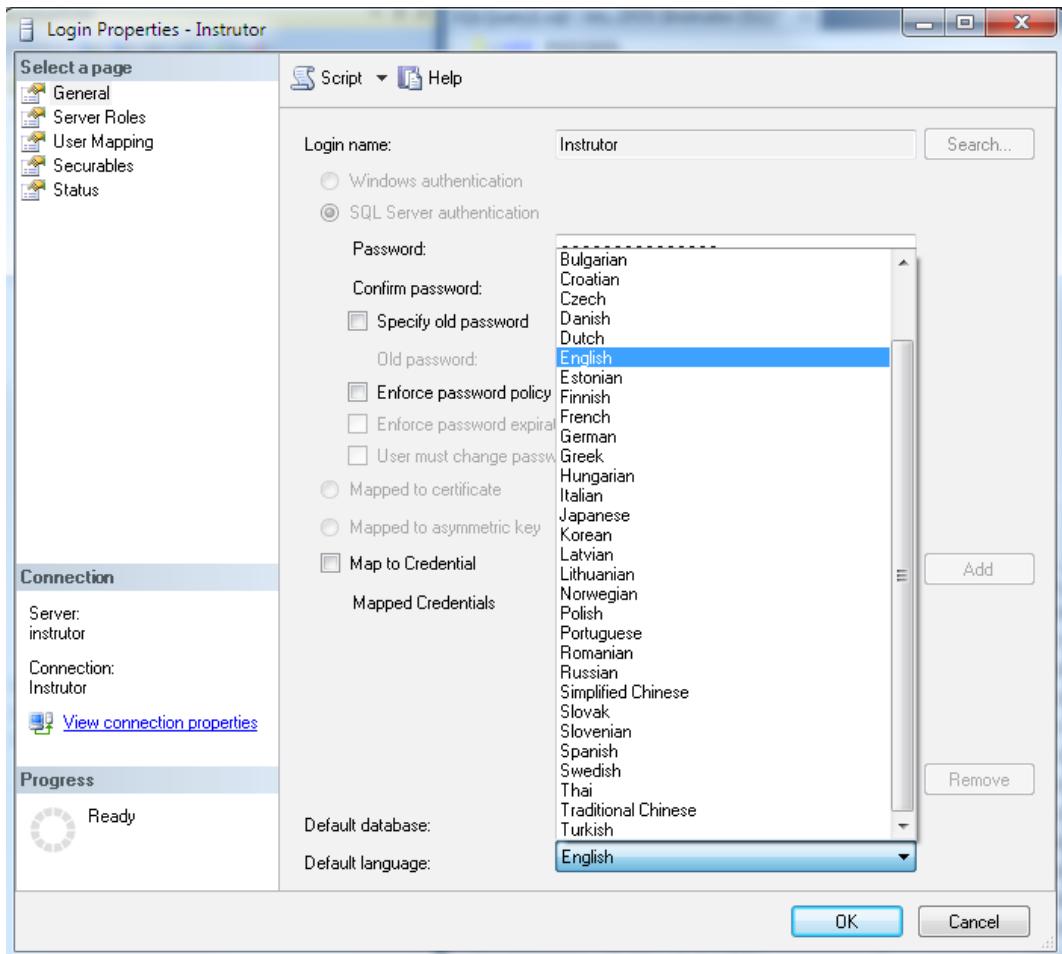
Para alterar a configuração de logins existentes, siga os passos adiante:

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:



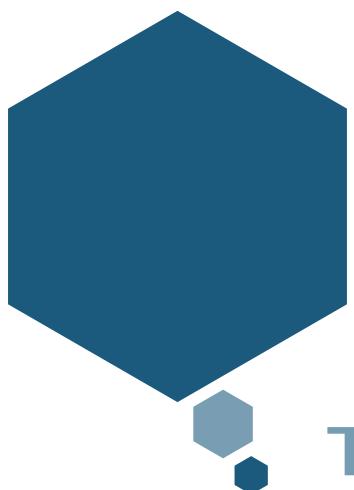
3. Na caixa de seleção **Default Language**, escolha o idioma desejado e clique em **OK**.

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 da leitura.

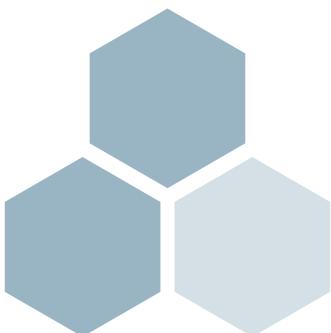
- Existem várias funções que auxiliam com valores de entrada de cadeia de caracteres, como LEN, REVERSE, CONCAT etc.;
- A função CASE permite que selecionemos valores conforme uma ou mais cláusulas específicas;
- O tipo de dado **datetime** é utilizado para definir valores de data e hora. Esse tipo é 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()**.



Comandos adicionais

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 28 a 30.



1. Qual destas funções não é uma função de texto?

- a) LEN
- b) STR
- c) COUNT
- d) CHARINDEX
- e) REPLICATE

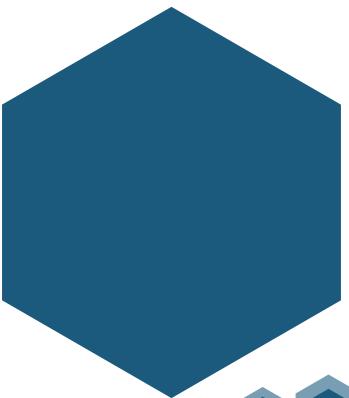
2. O que está faltando no comando a seguir?

```
SELECT NOME, SALARIO, CASE SINDICALIZADO
    WHEN 'S' THEN 'Sim'
    WHEN 'N' THEN 'Não'
    ELSE 'N/C'
    AS [Sindicato?] ,
    DATA_ADMISSAO
FROM TB_EMPREGADO;
```

- a) ORDER BY
- b) END
- c) GROUP BY
- d) O comando está correto.
- e) WHERE

3. Com relação ao uso de um filtro por ano em uma consulta, qual opção está incorreta?

- 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.



Comandos adicionais



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 28 a 30.



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 **TB_PRODUTO** e **TBPROD**, apresentando os registros que são iguais;
7. Compare as tabelas **TB_PRODUTO** e **TBPROD**, apresentando os registros que aparecem somente em **TB_PRODUTO**.

Laboratório 2

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/2014;
3. Liste todos os pedidos com data de emissão no primeiro semestre de 2014;
4. Liste todos os pedidos com data de emissão em janeiro e junho de 2014;
5. Liste todos os pedidos do Vendedor Código 1 em Jan/2014;
6. Liste os pedidos emitidos em Jan/2014 em uma sexta-feira.