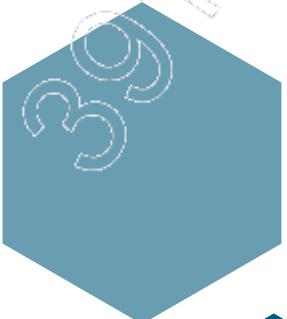
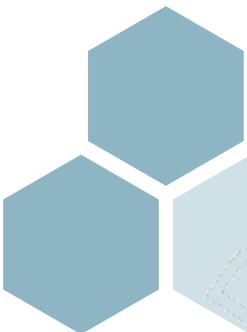


# SQL 2019 - Módulo I

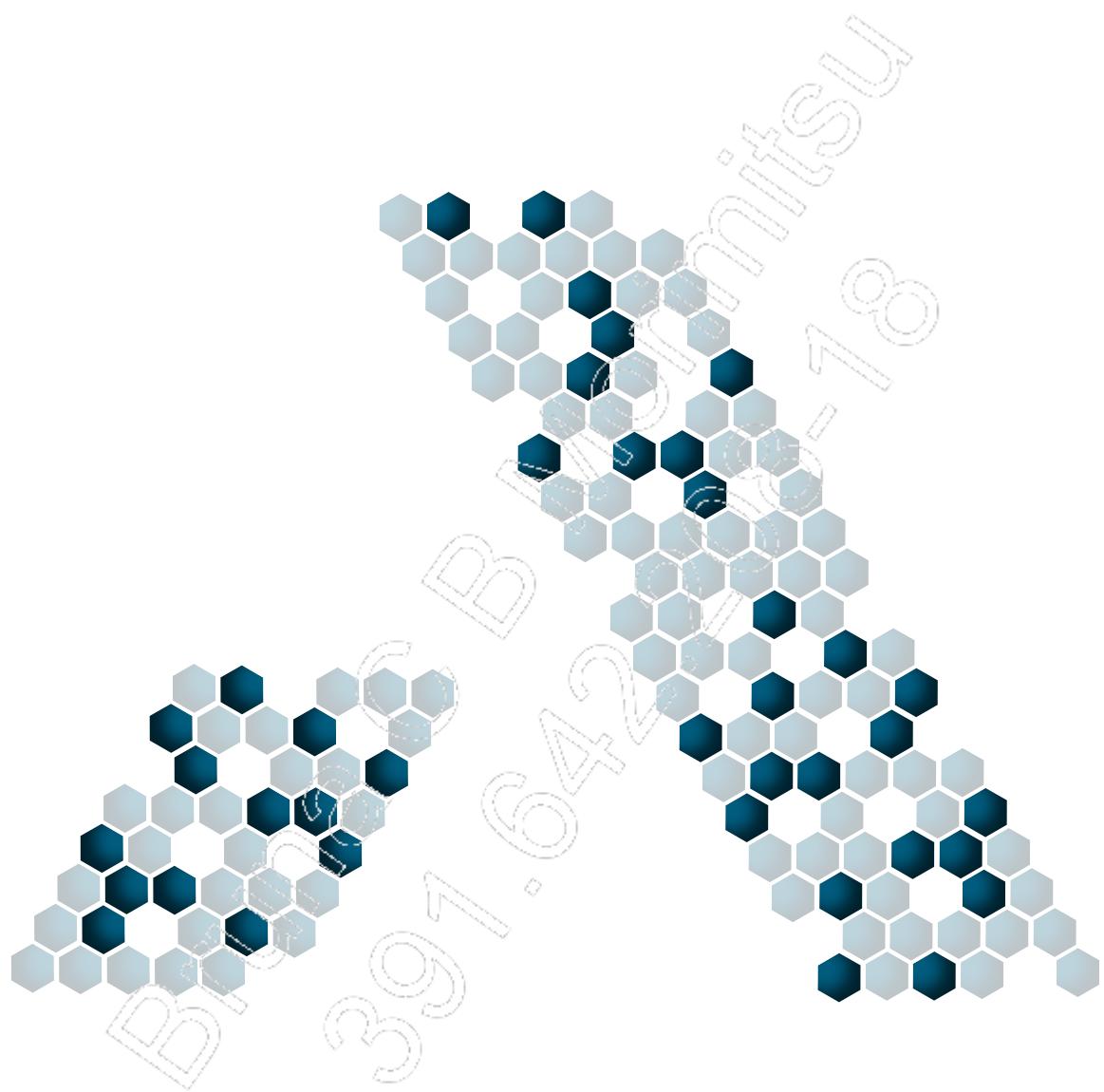


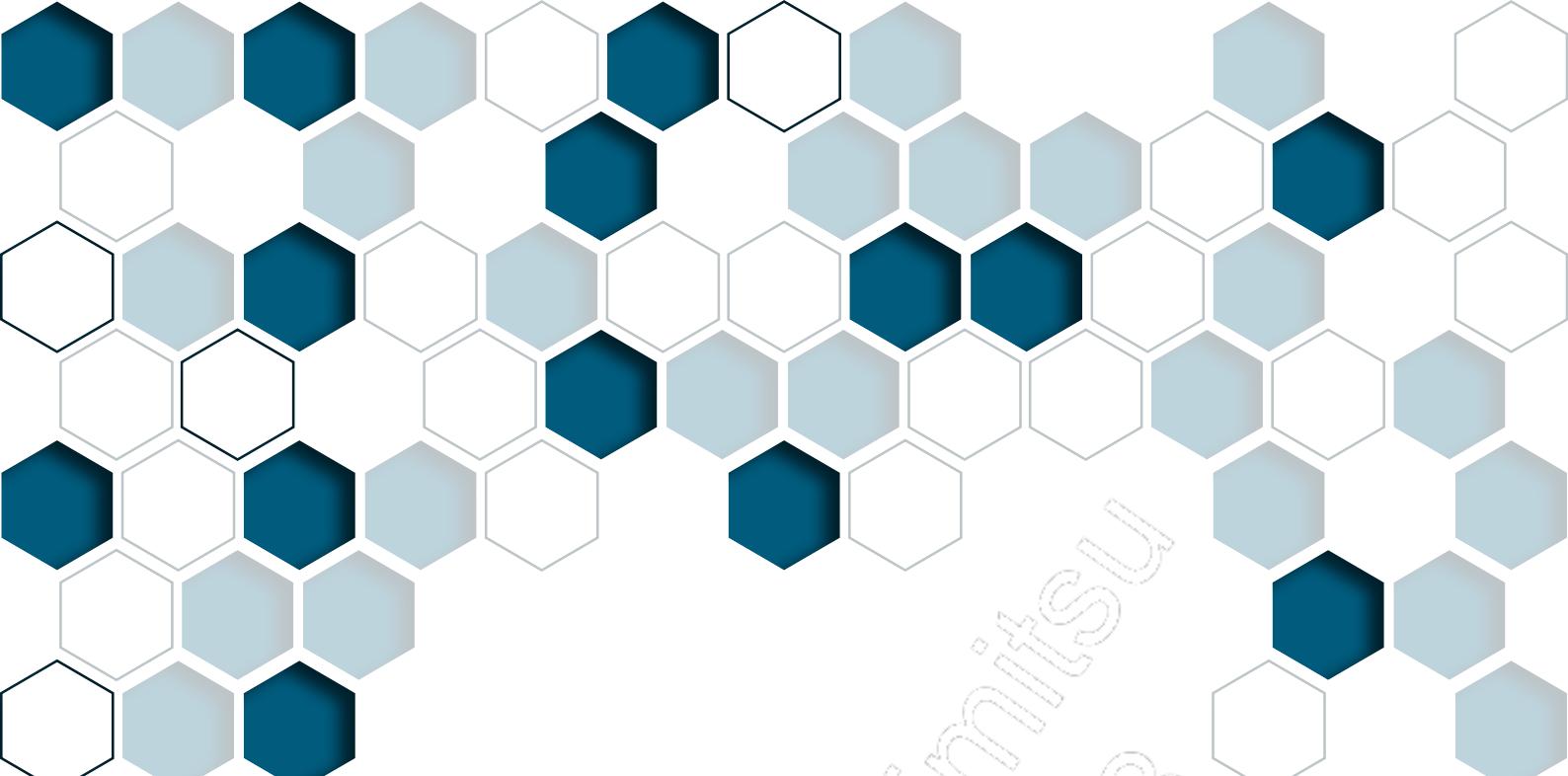
397



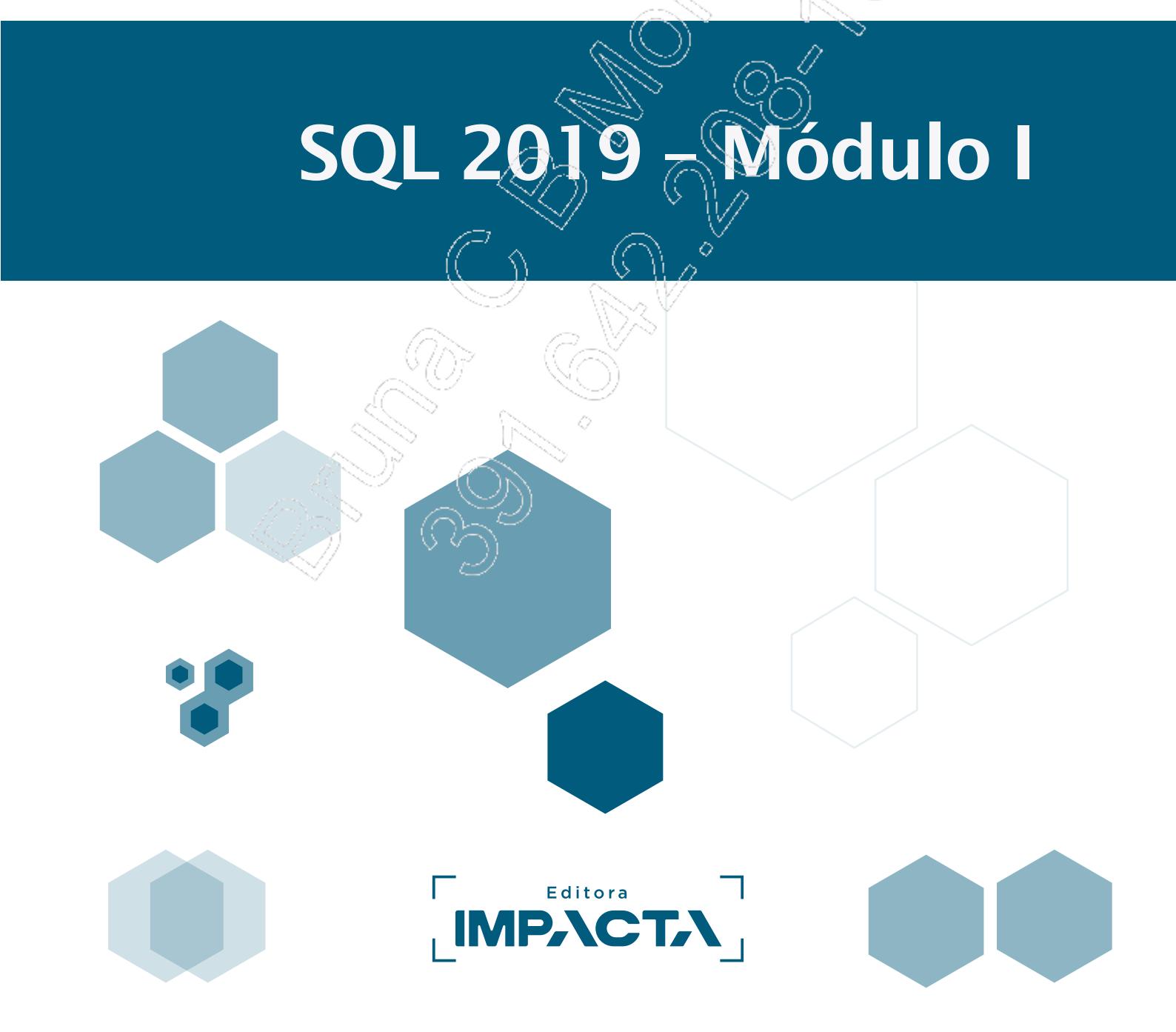
Editora  
**IMPACTA**







# SQL 2019 - Módulo I



397.642



## 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 2019 – Módulo I

## Coordenação Geral

Henrique Thomaz Bruscagin

## Autoria

Daniel Paulo Tamarosi Salvador

## Revisão Ortográfica e Gramatical

Marcos Cesar dos Santos Silva

## Diagramação

Bruno de Oliveira Santos

## Edição nº 1 | 1882\_1

Janeiro/ 2020

*Este material constitui uma nova obra e é uma derivação da seguinte obra original, produzida por Monte Everest Participações e Empreendimentos Ltda., em Ago/2016: SQL 2016 – Módulo I*

*Autoria: Daniel Paulo Tamarosi Salvador*

# Sumário

<b>Capítulo 1 - Introdução ao SQL Server 2019 .....</b>	<b>09</b>
1.1. Apresentação do curso.....	10
1.1.1. Próximos treinamentos .....	10
1.2. Arquitetura do banco de dados .....	11
1.3. Banco de dados.....	12
1.3.1. Características .....	12
1.4. Arquitetura SQL Server .....	12
1.5. Edições .....	12
1.6. Instância .....	13
1.6.1. Acesso à instância.....	13
1.7. A linguagem SQL.....	13
1.8. Componentes do SQL Server .....	15
1.9. Objetos de banco de dados.....	15
1.9.1. Tabelas .....	15
1.9.2. Índices .....	15
1.9.3. CONSTRAINT.....	15
1.9.4. VIEW (Visualização) .....	16
1.9.5. PROCEDURE (Procedimento armazenado) .....	16
1.9.6. FUNCTION (Função).....	16
1.9.7. TRIGGER (Gatilho) .....	16
1.10. Ferramentas de gerenciamento .....	17
1.11. SQL Server Management Studio (SSMS) .....	18
1.11.1. Inicializando o SSMS.....	18
1.11.2. Interface .....	20
1.11.3. Executando um comando .....	24
1.11.4. Opções .....	25
1.11.5. Salvando scripts .....	27
1.11.6. Soluções e Projetos .....	28
1.12. Comandos básicos .....	29
1.12.1. Nomenclatura .....	29
1.12.2. Comentários .....	30
1.12.3. Executando comandos .....	30
1.12.4. Resultado.....	31
1.12.4.1. Salvando resultados .....	32
1.12.5. Design Query in Editor .....	33
Pontos principais .....	36
<b>Teste seus conhecimentos.....</b>	<b>37</b>
<b>Mãos à obra!</b> .....	<b>41</b>
<b>Capítulo 2 - Banco de dados.....</b>	<b>45</b>
2.1. Introdução .....	46
2.2. Bancos de dados do sistema .....	46
2.2.1. Master .....	46
2.2.2. tempdb .....	46
2.2.3. Model .....	47
2.2.4. msdb .....	47
2.2.5. Resource.....	47
2.3. Bancos de dados snapshot .....	48
2.4. Criação do banco de dados .....	48
2.4.1. Comando T-SQL .....	49
2.4.2. Criação de banco de dados graficamente .....	50
2.5. Uso do banco de dados .....	52
2.6. Referenciando objetos .....	53
2.7. Objetos de catálogo .....	54
2.7.1. Metadados .....	54

# SQL 2019 – Módulo I

2.7.2.	Informações do banco de dados.....	56
2.7.2.1.	Catálogos do sistema.....	57
2.8.	Grupos de comandos T-SQL .....	62
Pontos principais .....	63	
<b>Teste seus conhecimentos.....</b>	<b>65</b>	
<b>Mãos à obra!.....</b>	<b>69</b>	

## **Capítulo 3 - Consultando dados ..... 73**

3.1.	Introdução .....	74
3.2.	SELECT.....	75
3.2.1.	Informando o nome das colunas .....	77
3.2.2.	Realizando cálculos.....	78
3.2.3.	Concatenando textos .....	80
3.2.4.	Renomeando o cabeçalho da coluna.....	80
3.3.	Ordenando dados .....	82
3.3.1.	Retornando linhas na ordem ascendente.....	82
3.3.2.	Retornando linhas na ordem descendente.....	83
3.3.3.	Ordenando por nome, alias ou posição .....	83
3.3.4.	Ranking em consulta.....	86
3.3.5.	ORDER BY com TOP WITH TIES .....	88
3.4.	Filtrando consultas.....	90
3.4.1.	Operadores relacionais .....	90
3.5.	Operadores lógicos .....	92
3.6.	Intervalos de valores .....	94
3.7.	Pesquisa em campo texto .....	95
3.8.	Lista de elementos .....	98
3.9.	Valores nulos .....	99
3.10.	Funções para tratamento de nulos .....	101
3.11.	Campos data e hora .....	102
Pontos principais .....	108	
<b>Teste seus conhecimentos.....</b>	<b>109</b>	
<b>Mãos à obra!.....</b>	<b>113</b>	

## **Capítulo 4 - Associando tabelas ..... 117**

4.1.	Introdução .....	118
4.2.	INNER JOIN.....	119
4.3.	SELF JOIN .....	123
4.4.	OUTER JOIN.....	123
4.5.	CROSS JOIN .....	125
Pontos principais .....	126	
<b>Teste seus conhecimentos.....</b>	<b>127</b>	
<b>Mãos à obra!.....</b>	<b>131</b>	

## **Capítulo 5 - Subconsultas ..... 135**

5.1.	Introdução .....	136
5.2.	Características .....	136
5.3.	Subconsulta com IN e NOT IN .....	140
5.4.	Operadores .....	141
5.5.	Subconsulta correlacionada.....	142
5.5.1.	Correlação com EXISTS .....	142
5.6.	Subconsultas e associações .....	143
Pontos principais .....	145	
<b>Teste seus conhecimentos.....</b>	<b>147</b>	
<b>Mãos à obra!.....</b>	<b>151</b>	

# Sumário

<b>Capítulo 6 - Agrupando dados .....</b>	<b>153</b>
6.1.    Introdução .....	154
6.2.    Funções de agregação .....	154
6.3.    GROUP BY .....	159
6.3.1.    Utilizando ALL .....	163
6.3.2.    Utilizando HAVING .....	164
6.3.3.    Utilizando WITH ROLLUP .....	166
6.3.4.    Utilizando WITH CUBE .....	168
Pontos principais .....	170
<b>Teste seus conhecimentos.....</b>	<b>171</b>
<b>Mãos à obra! .....</b>	<b>175</b>
<b>Capítulo 7 - Modelando um banco de dados.....</b>	<b>177</b>
7.1.    Design do banco de dados .....	178
7.1.1.    Modelo descritivo .....	178
7.1.2.    Modelo conceitual .....	179
7.1.3.    Modelo lógico .....	181
7.1.4.    Modelo físico .....	181
7.1.5.    Dicionário de dados .....	183
7.2.    Normalização de dados .....	184
7.2.1.    Regras de normalização .....	184
7.3.    Tipos de dados .....	188
7.3.1.    Numéricos exatos .....	189
7.3.2.    Numéricos aproximados .....	190
7.3.3.    Data e hora .....	190
7.3.4.    Caracteres texto .....	191
7.3.5.    Caracteres UNICODE .....	192
7.3.6.    Valores binários .....	192
7.3.7.    Outros tipos de dados .....	193
7.4.    Tabelas .....	193
7.4.1.    Tabelas regulares .....	194
7.4.2.    Tabelas temporárias locais .....	194
7.4.3.    Tabelas temporárias globais .....	195
7.4.4.    Tabelas baseadas em consultas .....	195
7.4.5.    Criando tabelas (CREATE TABLE) .....	196
7.4.6.    Auto numeração (IDENTITY) .....	198
7.5.    CONSTRAINTS .....	199
7.5.1.    Nulos .....	199
7.5.2.    Chave Primária (PRIMARY KEY) .....	200
7.5.3.    Chave única (UNIQUE) .....	202
7.5.4.    Checagem (CHECK) .....	204
7.5.5.    Valor padrão (DEFAULT) .....	205
7.5.6.    FOREIGN KEY (chave estrangeira) .....	206
7.6.    Apagando tabelas .....	208
7.7.    Alterando tabelas .....	208
Pontos principais .....	211
<b>Teste seus conhecimentos.....</b>	<b>213</b>
<b>Mãos à obra! .....</b>	<b>217</b>
<b>Capítulo 8 - Opções de definição de tabelas.....</b>	<b>221</b>
8.1.    Introdução .....	222
8.2.    Tipos de dados definidos pelo usuário .....	222
8.2.1.    CREATE TYPE .....	222
8.2.2.    DROP TYPE .....	223
8.2.3.    Regras e valores padrão .....	224

# SQL 2019 – Módulo I

8.2.4.	Criação de tipo de dados graficamente .....	224
8.2.5.	Trabalhando com UDDT .....	225
8.2.6.	Tipo tabular .....	227
8.2.6.1.	Tipo tabular otimizado em memória .....	228
8.3.	Tabelas de sistema .....	230
8.3.1.	Tabela SYSTYPES .....	230
8.3.2.	Tabela SYSOBJECTS .....	231
8.3.3.	Tabela SYSCOMMENTS .....	231
8.4.	Sequências.....	232
8.5.	Sinônimos .....	234
8.6.	Trabalhando com objetos binários .....	235
8.6.1.	Campos binários .....	235
8.7.	FILETABLE .....	237
8.8.	Colunas computadas.....	245
Pontos principais .....		246
<b>Teste seus conhecimentos.....</b>		<b>247</b>
<b>Mãos à obra!.....</b>		<b>251</b>
<b>Capítulo 9 - Inserção de dados.....</b>		<b>255</b>
9.1.	Constantes.....	256
9.2.	Inserindo dados .....	258
9.2.1.	INSERT posicional .....	260
9.2.2.	INSERT declarativo .....	261
9.2.3.	INSERT com CTE.....	261
9.2.4.	INSERT com tipo tabular.....	262
9.3.	Cláusula TOP .....	262
9.4.	OUTPUT .....	263
9.5.	Funções para campos autonomeáveis .....	264
Pontos principais .....		270
<b>Teste seus conhecimentos.....</b>		<b>271</b>
<b>Mãos à obra!.....</b>		<b>275</b>
<b>Capítulo 10 - Atualizando e excluindo dados.....</b>		<b>279</b>
10.1.	Introdução .....	280
10.2.	UPDATE .....	280
10.2.1.	Alterando dados de uma coluna .....	281
10.2.2.	Alterando dados de diversas colunas .....	282
10.2.3.	Utilizando TOP .....	282
10.2.4.	UPDATE com subconsulta .....	282
10.2.5.	UPDATE com JOIN .....	283
10.3.	DELETE .....	283
10.3.1.	Excluindo todas as linhas de uma tabela .....	284
10.3.2.	Utilizando TOP em uma instrução DELETE .....	285
10.3.3.	DELETE com subconsulta .....	285
10.3.4.	DELETE com JOIN .....	285
10.4.	OUTPUT para DELETE e UPDATE .....	286
10.5.	Transações .....	287
10.5.1.	Transações explícitas .....	287
10.6.	MERGE .....	289
10.6.1.	OUTPUT em uma instrução MERGE .....	292
Pontos principais .....		294
<b>Teste seus conhecimentos.....</b>		<b>295</b>
<b>Mãos à obra!.....</b>		<b>299</b>

# 1

# Introdução ao SQL Server 2019

- ◆ Apresentação do curso;
- ◆ Arquitetura do banco de dados;
- ◆ Banco de dados;
- ◆ Arquitetura SQL Server;
- ◆ Edições;
- ◆ Instância;
- ◆ A linguagem SQL;
- ◆ Componentes do SQL Server;
- ◆ Objetos de banco de dados;
- ◆ Ferramentas de gerenciamento;
- ◆ SQL Server Management Studio (SSMS);
- ◆ Comandos básicos.

## 1.1. Apresentação do curso

Este curso tem a finalidade de apresentar o software Microsoft SQL Server 2019, seus componentes, linguagem SQL e a extração dos dados com suas diversas variações.

O método que usaremos está diretamente relacionado com testes e exercícios práticos. A cada novo capítulo serão incluídos novos conceitos e aumento progressivo da dificuldade.

O que a Impacta deseja é que você, aluno, tenha um treinamento prático e interativo para alavancar sua carreira profissional.

Neste módulo você vai aprender sobre:

- Banco de dados;
- Consulta em tabelas;
- Agrupamento.

### 1.1.1. Próximos treinamentos

Dentro da carreira de banco de dados e treinamento voltado para banco de dados, segue a descrição dos próximos treinamentos:

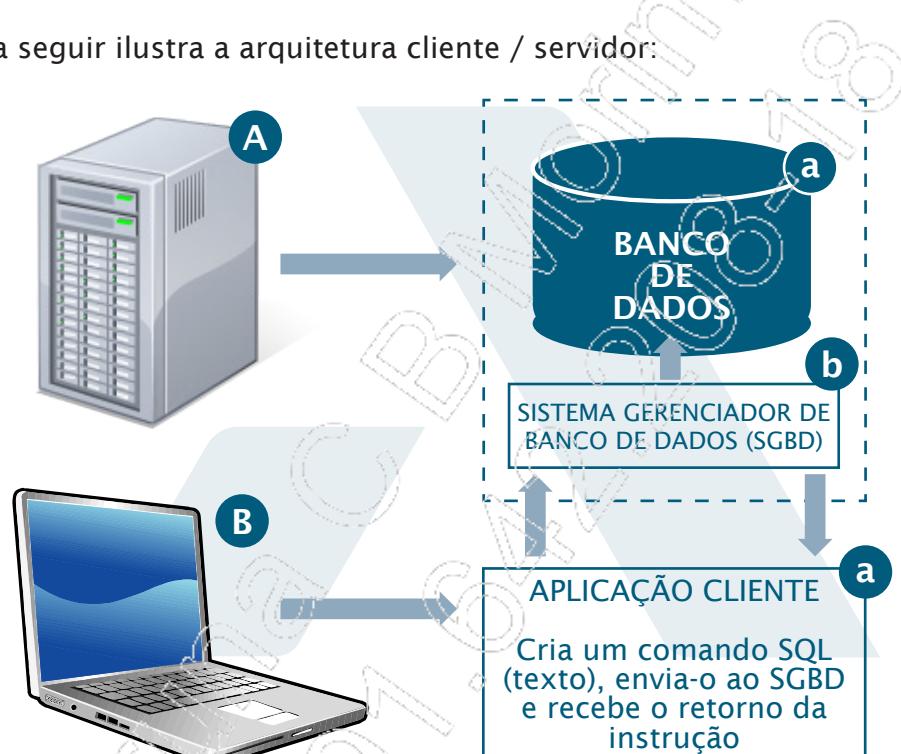
- **SQL Server 2019 Módulo II**
  - Views;
  - Programação em SQL;
  - Procedure;
  - Funções criadas pelo usuário;
  - Tratamento de erros;
  - Trigger.
- **SQL Server 2019 Módulo III**
  - Administração e manutenção do SQL Server;
  - Instalação do SQL Server;
  - Criação de banco de dados;
  - Criação e permissão de usuários de acesso;
  - Backup e restore;
  - Replicação;
  - Alta disponibilidade.
- **SQL Server 2019 – Business Intelligence**
  - Análise de dados;
  - Banco de dados analítico;
  - Extração, transformação e carga (ETL);
  - SSIS – SQL Server Integration Services;
  - SSAS – SQL Server Analysis Services;
  - SSRS – SQL Server Reporting Services.

## 1.2. Arquitetura do banco de dados

Essa arquitetura funciona da seguinte forma:

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

A imagem a seguir ilustra a arquitetura cliente / servidor:



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

### 1.3. Banco de dados

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.

#### 1.3.1. Características

As características são as seguintes:

- Independência dos programas e dos dados;
- Performance no acesso ao dado;
- Integração;
- Segurança;
- Acesso concorrente ao dado;
- Otimização do espaço de armazenamento.

### 1.4. Arquitetura SQL Server

O SQL Server é uma plataforma de banco de dados utilizada para armazenar dados e processá-los. Podemos utilizar para qualquer aplicação que necessite de um repositório seguro para dados e imagens.

Sistemas transacionais e analíticos são suportados pelo SQL oferecendo alto desempenho e segurança.

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

### 1.5. Edições

O SQL Server possuía as seguintes versões:

- **Enterprise**;
- **Standard**;
- **Express** (Versão gratuita com limite de banco até 10 GB);
- **Developer** (Versão gratuita para desenvolvimento, mesmas características do **Enterprise**, porém não são permitidos dados produtivos).

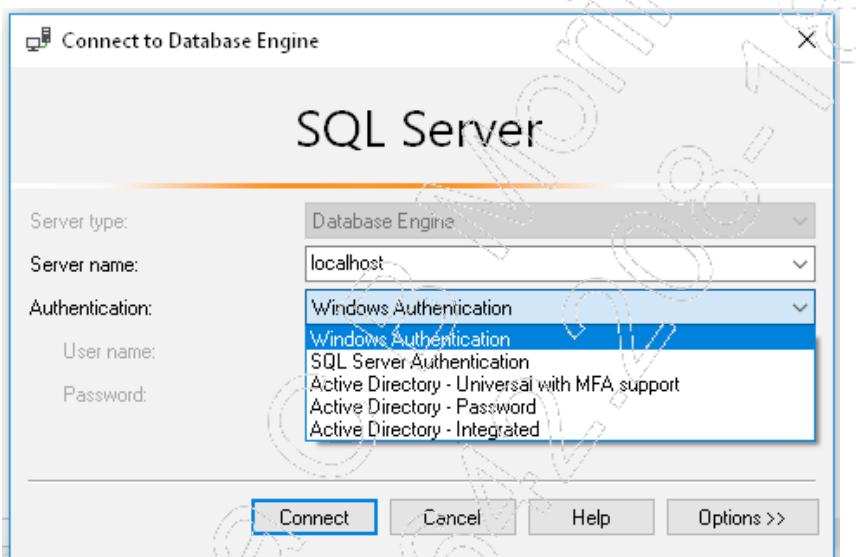
## 1.6. Instância

No SQL Server, cada instalação é considerada uma instância (Instance). É possível criar até 50 instalações em ambiente isolado, ou 25, em ambientes com alta disponibilidade.

A vantagem da utilização de instância é a separação total do serviço, banco de dados e usuários, dessa maneira é possível promover um isolamento do ambiente de banco de dados necessário para cada sistema ou área de negócio.

### 1.6.1. Acesso à instância

Para ter acesso à instância, é necessário que exista um login criado. Esse login pode estar associado ao login da máquina conectada.



## 1.7. A linguagem 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 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 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 2019, 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	COMPRAS
5	5	PRODUCAO
6	6	DIRETORIA
7	7	TELEMARKETING
8	8	FINANCEIRO
9	9	RECURSOS HUMANOS
10	10	TREINAMENTO

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

## 1.8. Componentes do SQL Server

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 (Online Analytical Processing);
- **Reporting Services:** Ferramenta para geração de relatórios;
- **Integration Services:** Facilita o processo de transferência de dados entre bancos de dados.

## 1.9. 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.

### 1.9.1. 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.

### 1.9.2. 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.

### 1.9.3. 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**.

### 1.9.4. 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.

### 1.9.5. 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.

### 1.9.6. 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.

### 1.9.7. 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**.

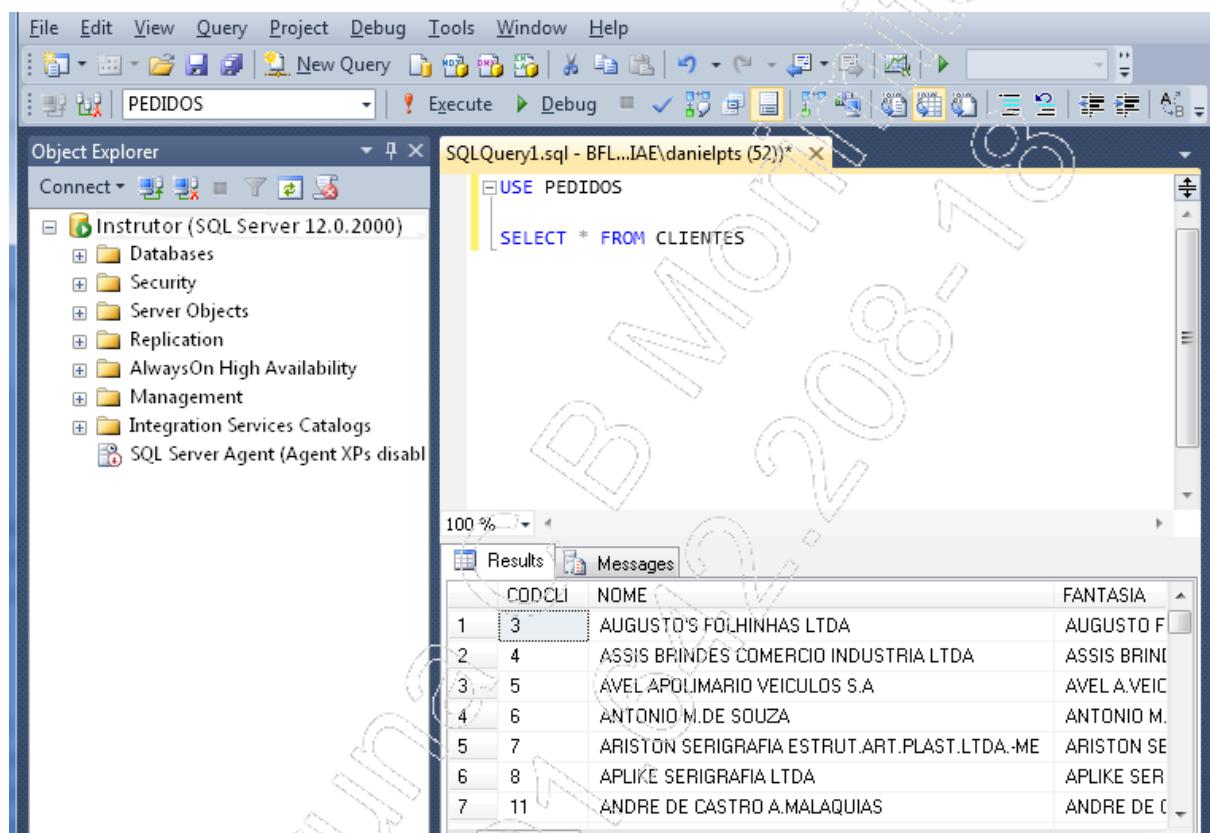
Quando atualizamos, inserimos ou excluímos dados em uma tabela, o TRIGGER automaticamente grava em uma tabela temporária os dados do registro atualizado, inserido ou excluído.

## 1.10. 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:

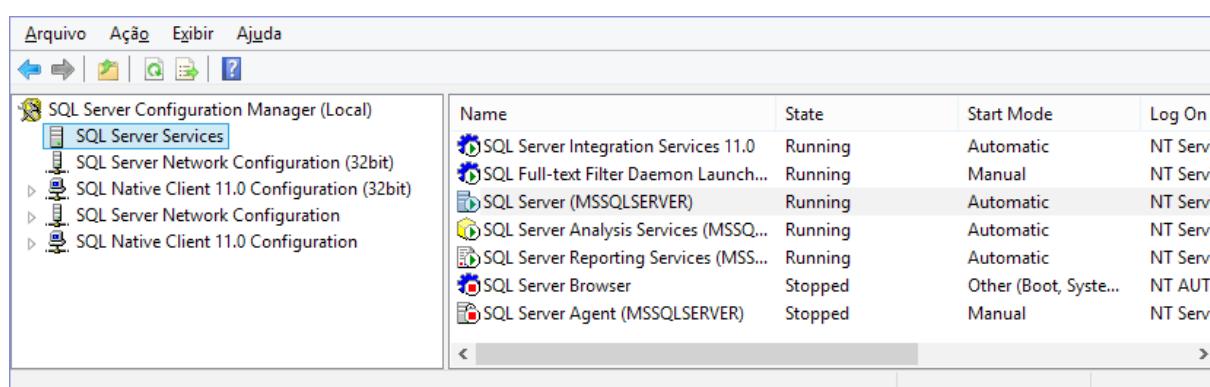
- **SQL Server Management Studio (SSMS)**

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:



- **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**.

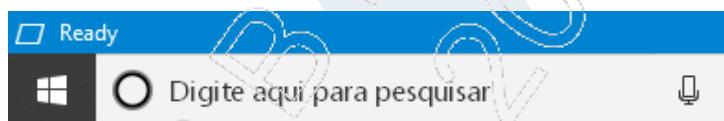
## 1.11.SQL Server Management Studio (SSMS)

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.

### 1.11.1.Inicializando o SSMS

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

É possível escrever o nome do SSMS na área de pesquisa:



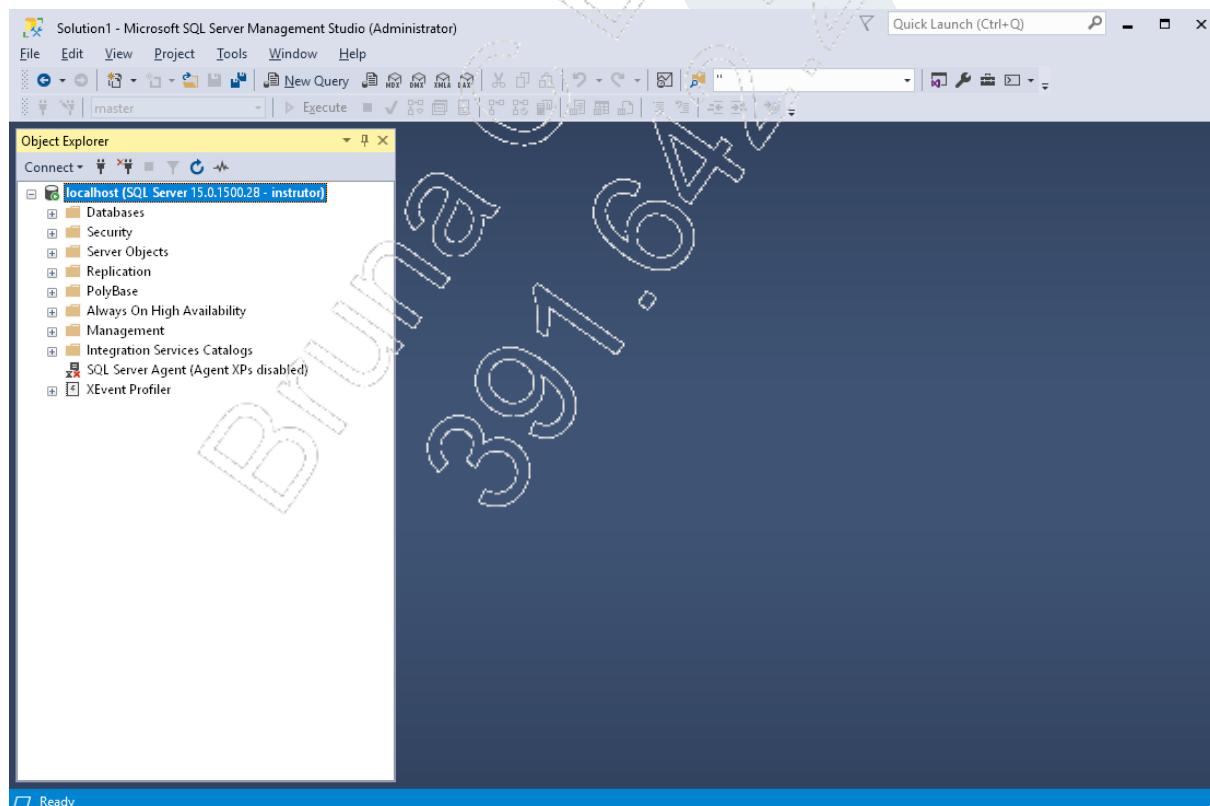
Ou por meio da lista de programas:



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:



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



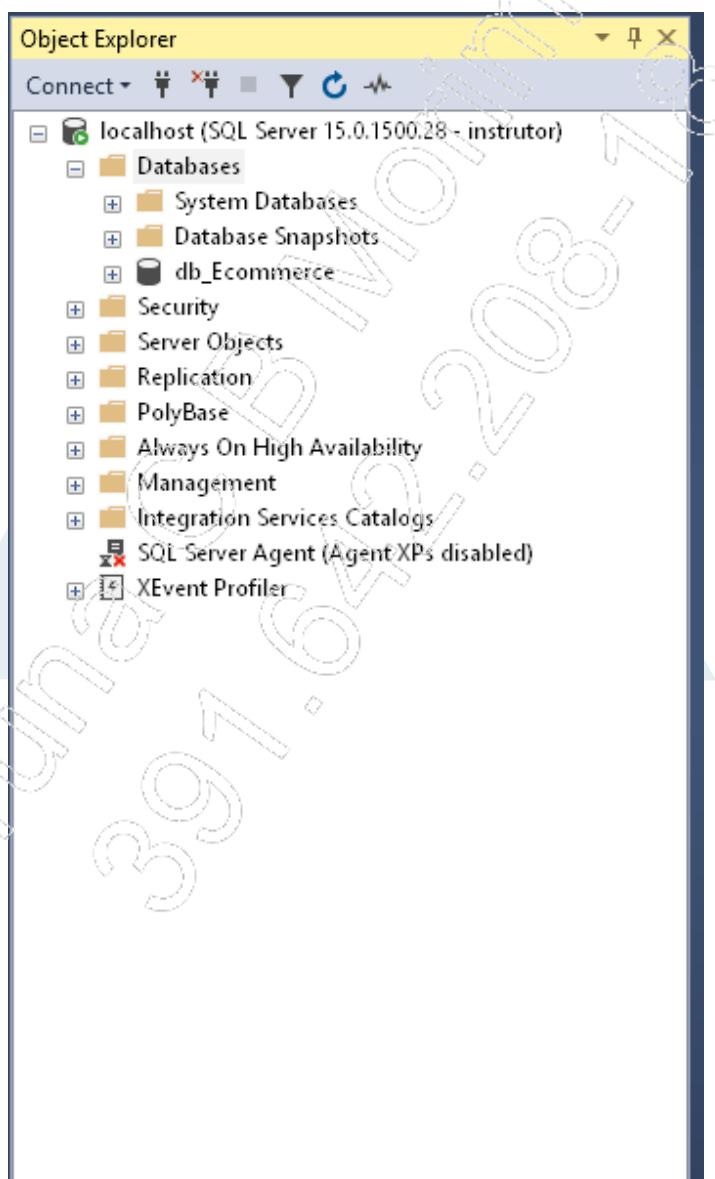
## 1.11.2. Interface

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

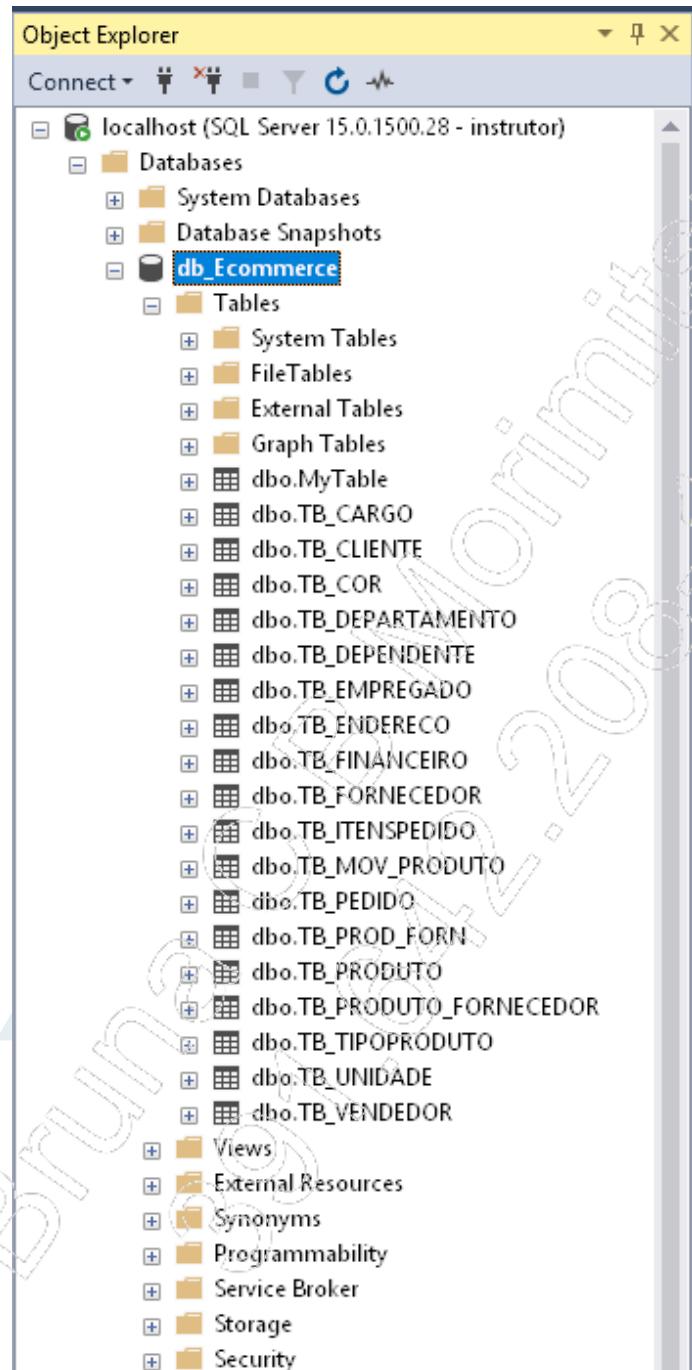
- **Object Explorer**

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

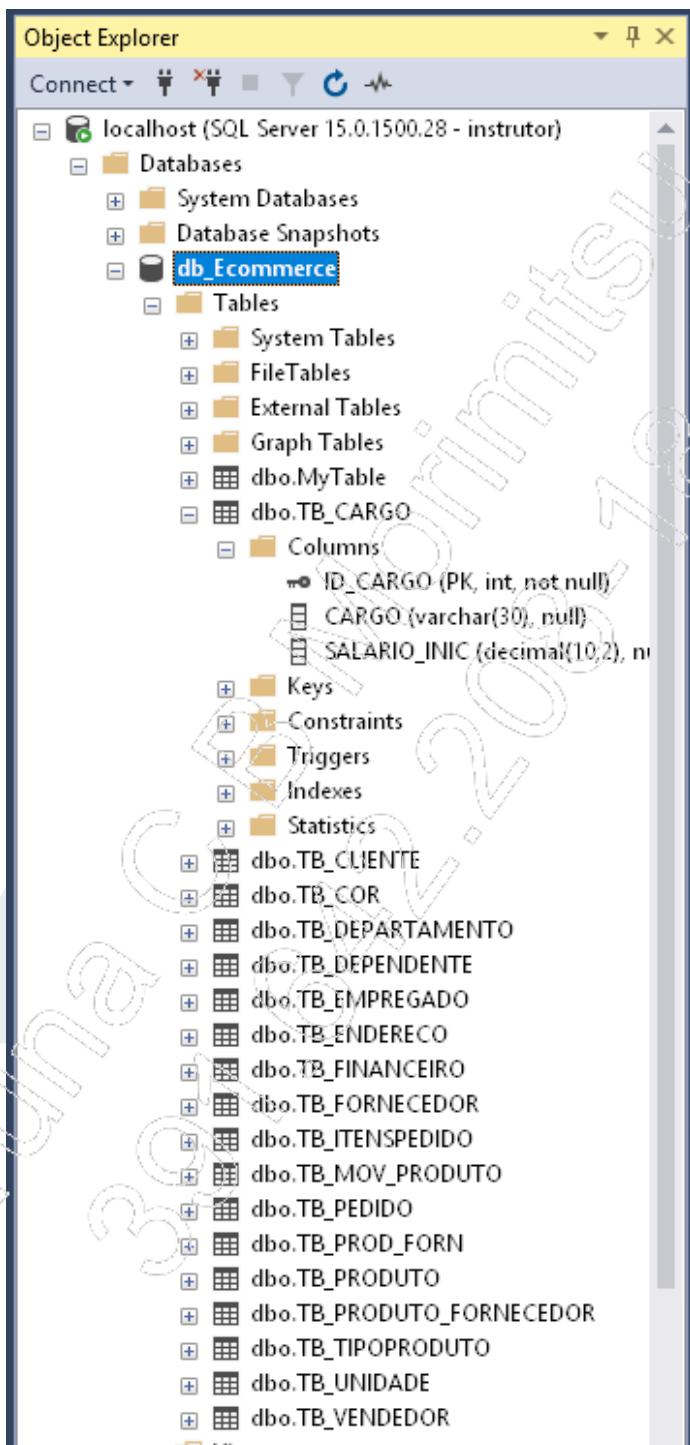
Observe que a pasta **Databases** mostra todos os bancos de dados existentes no servidor. No caso da imagem a seguir, **db\_Ecommerce** é um banco de dados:



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

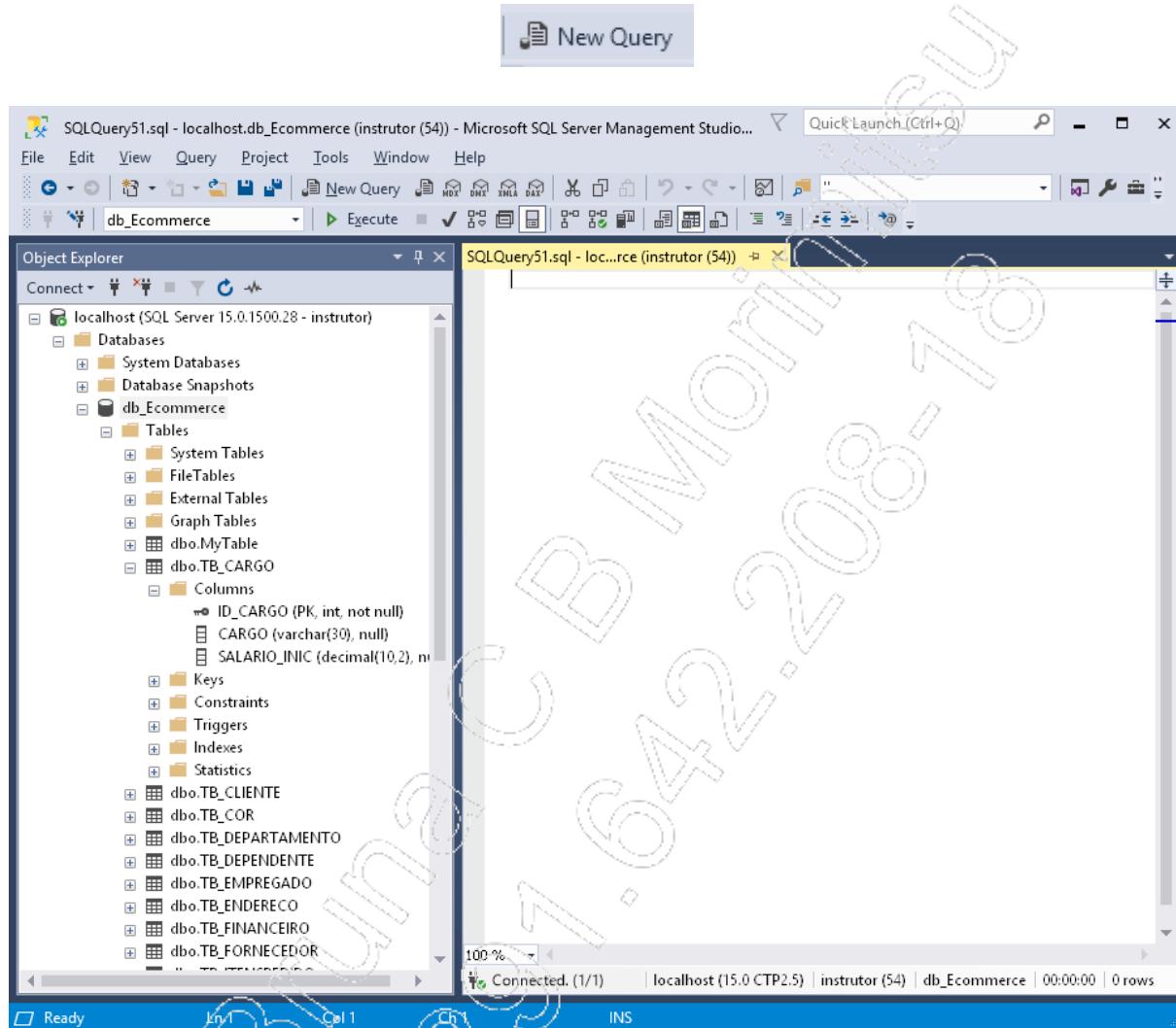


Expandindo uma das tabelas, veremos características da sua estrutura, bem como as suas colunas:



- **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 vazia será criada:



Cada uma dessas abas representa uma conexão com o banco de dados e recebe um **número de sessão**. Na imagem anterior, a sessão é a 54.

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.

### 1.11.3. Executando um comando

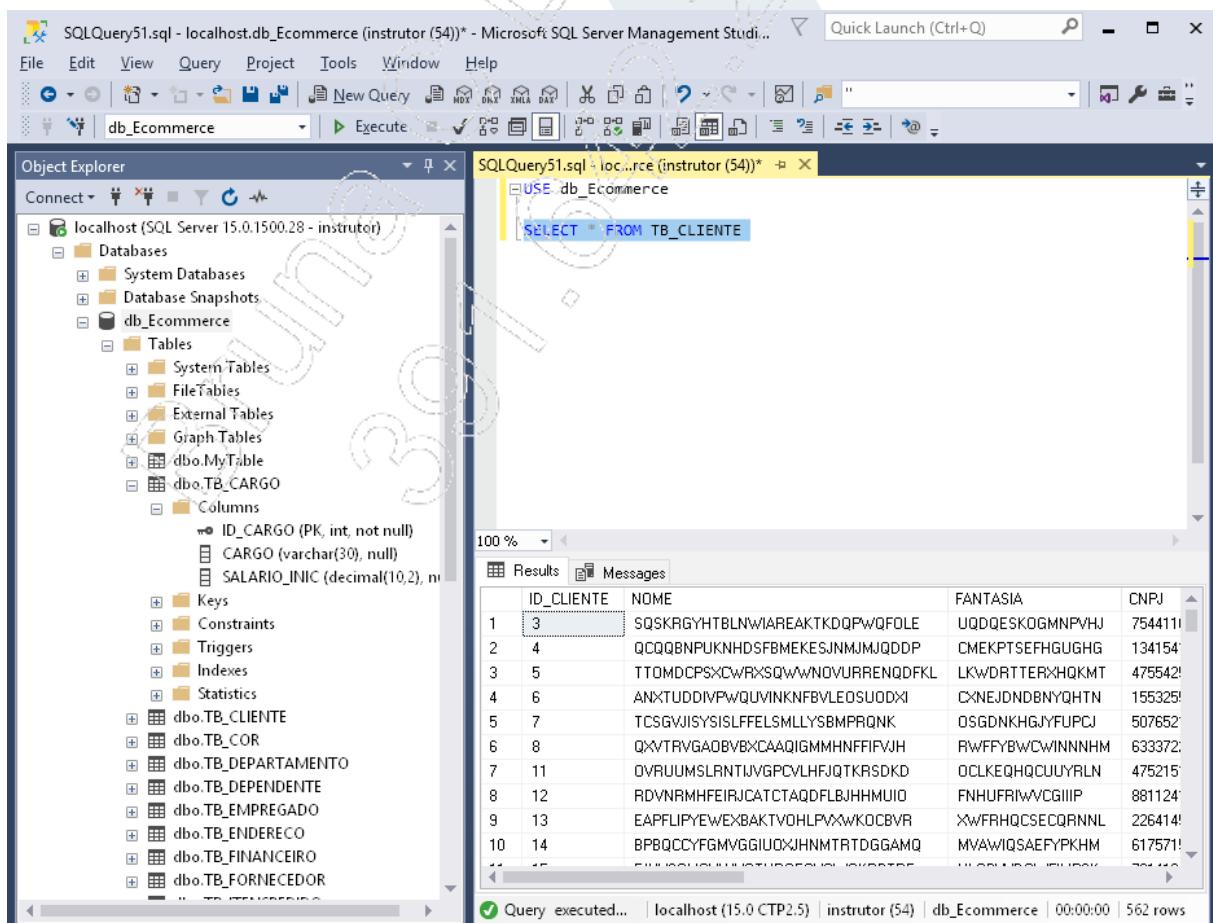
Para executar um comando a partir do SQL Server Management Studio, adote o seguinte procedimento:

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:

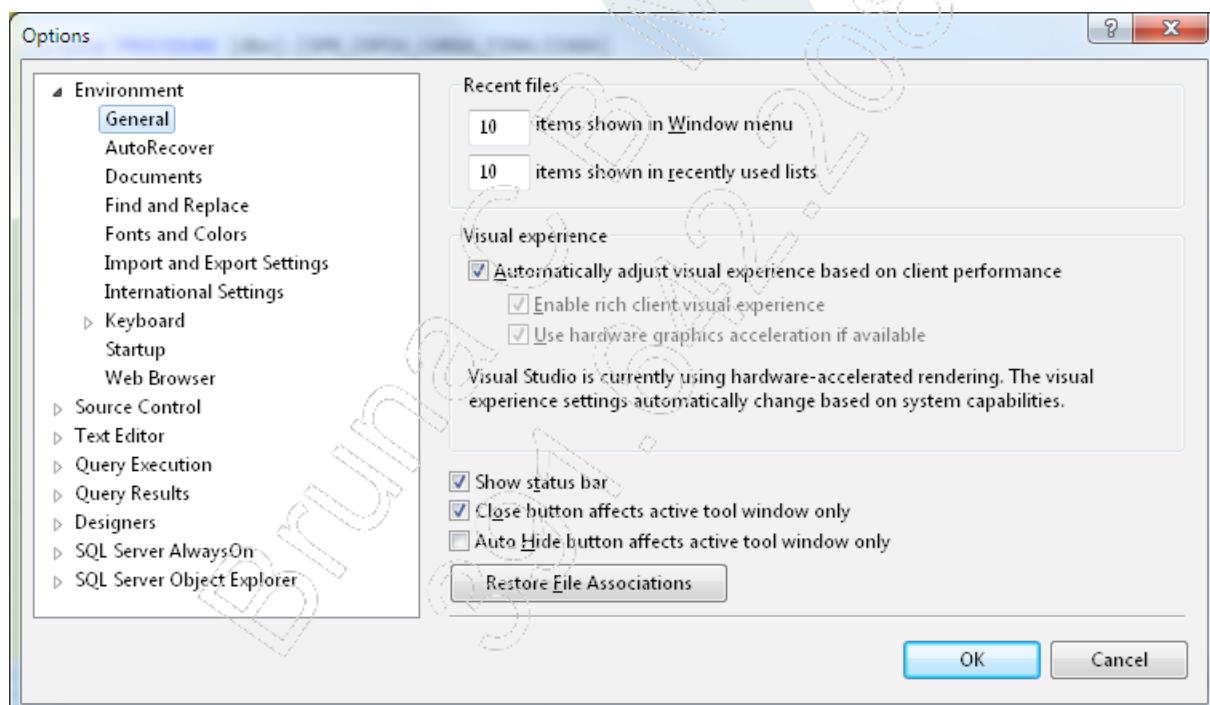


Com relação ao procedimento anterior, é importante considerar as seguintes informações:

- 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**.

## 1.11.4. Opções

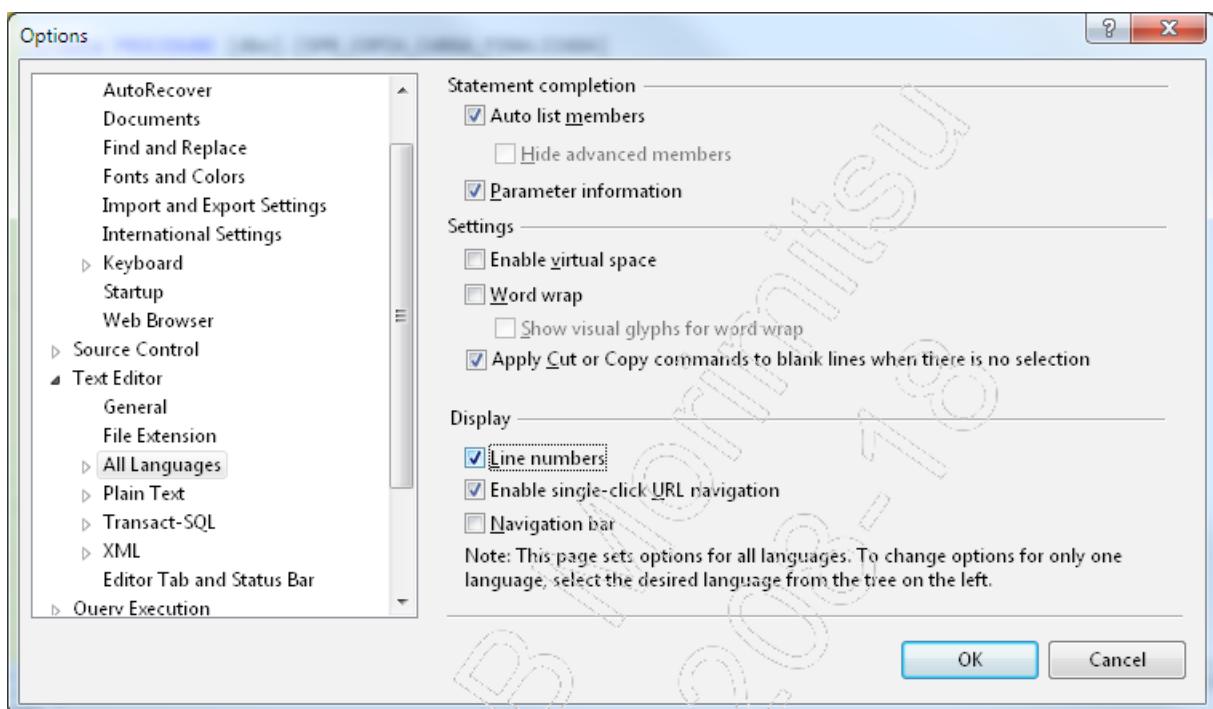
Várias opções de customização do SSMS podem ser ajustadas conforme a preferência do usuário. Para isso, no menu **Tools**, clique em **Options**, abrindo a seguinte janela:



Vejamos, a seguir, alguns exemplos de customização:

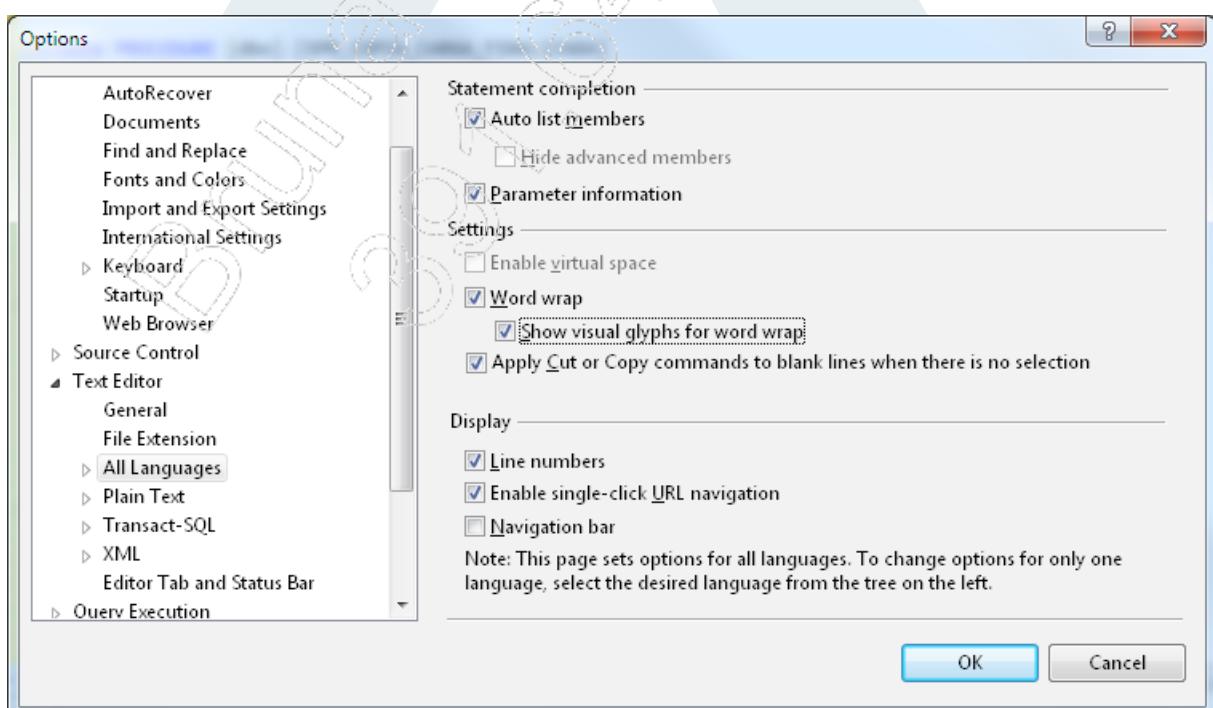
- Numeração automática da linha

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



- Quebra de linha de automática

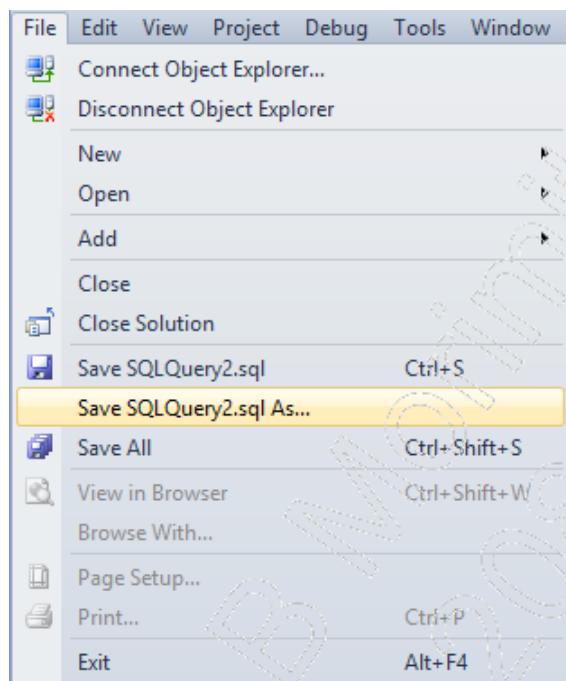
Na mesma guia, selecione **Word wrap** e **Show visual glyphs for word wrap**:



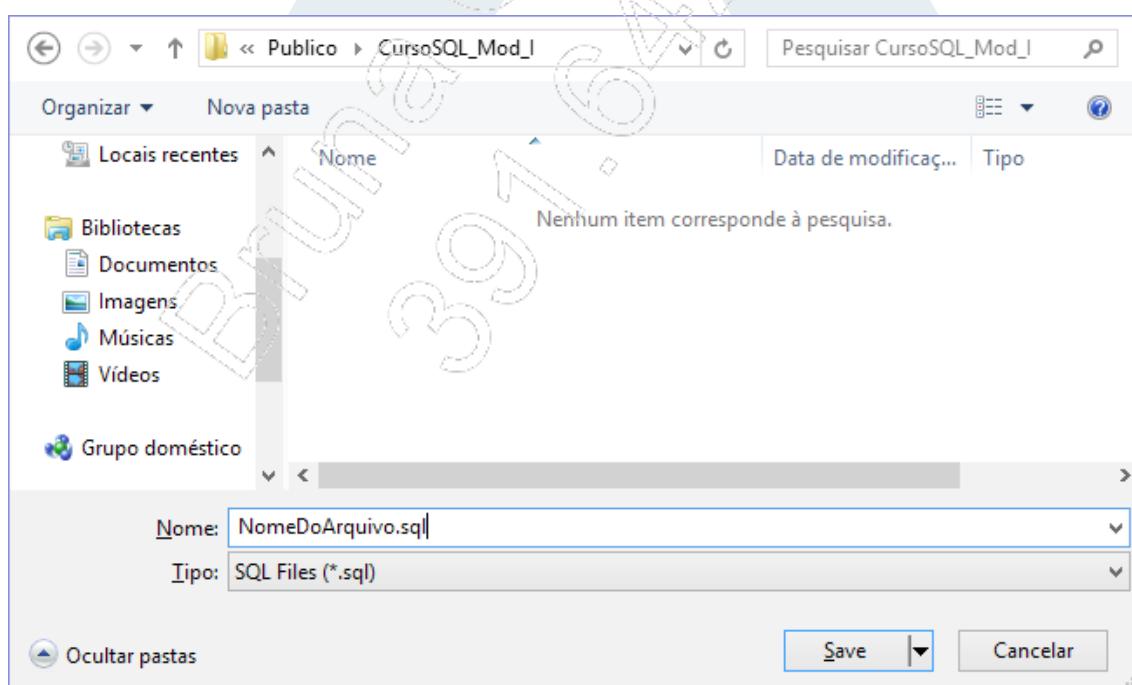
## 1.11.5. Salvando scripts

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**.

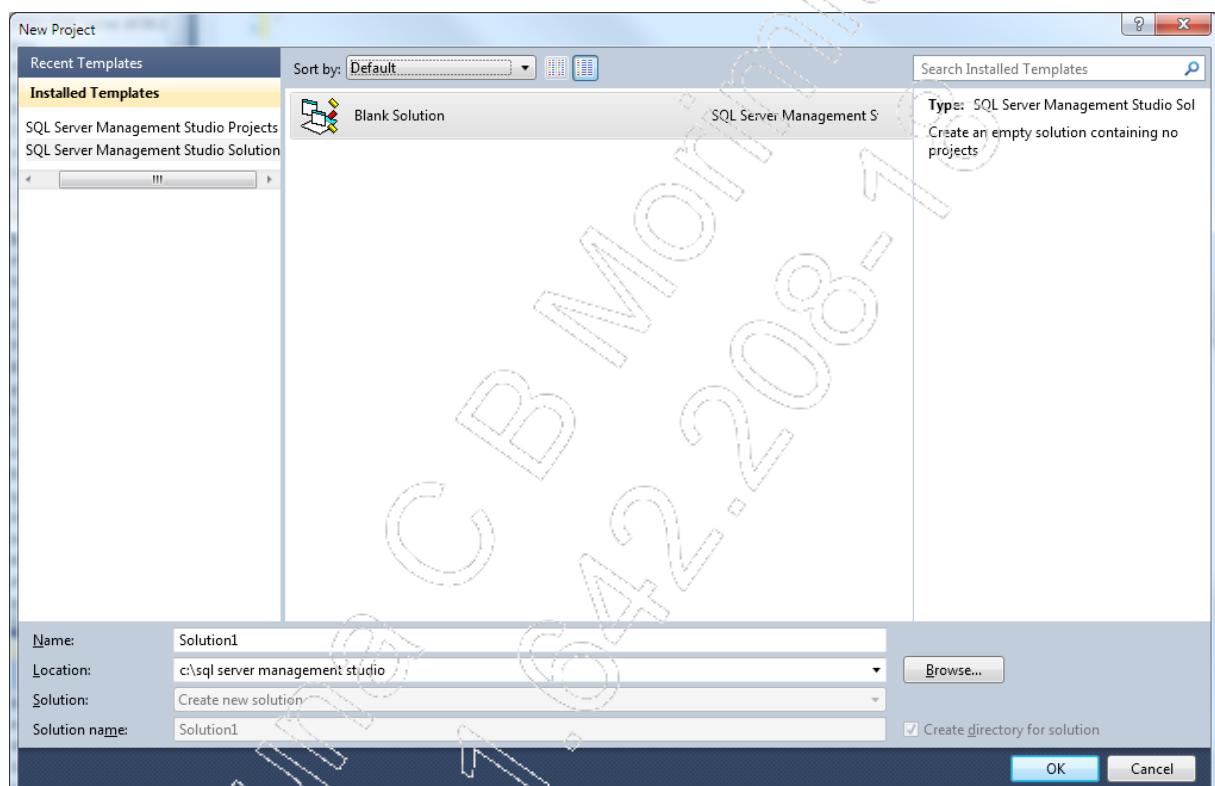
## 1.11.6. Soluções e Projetos

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:

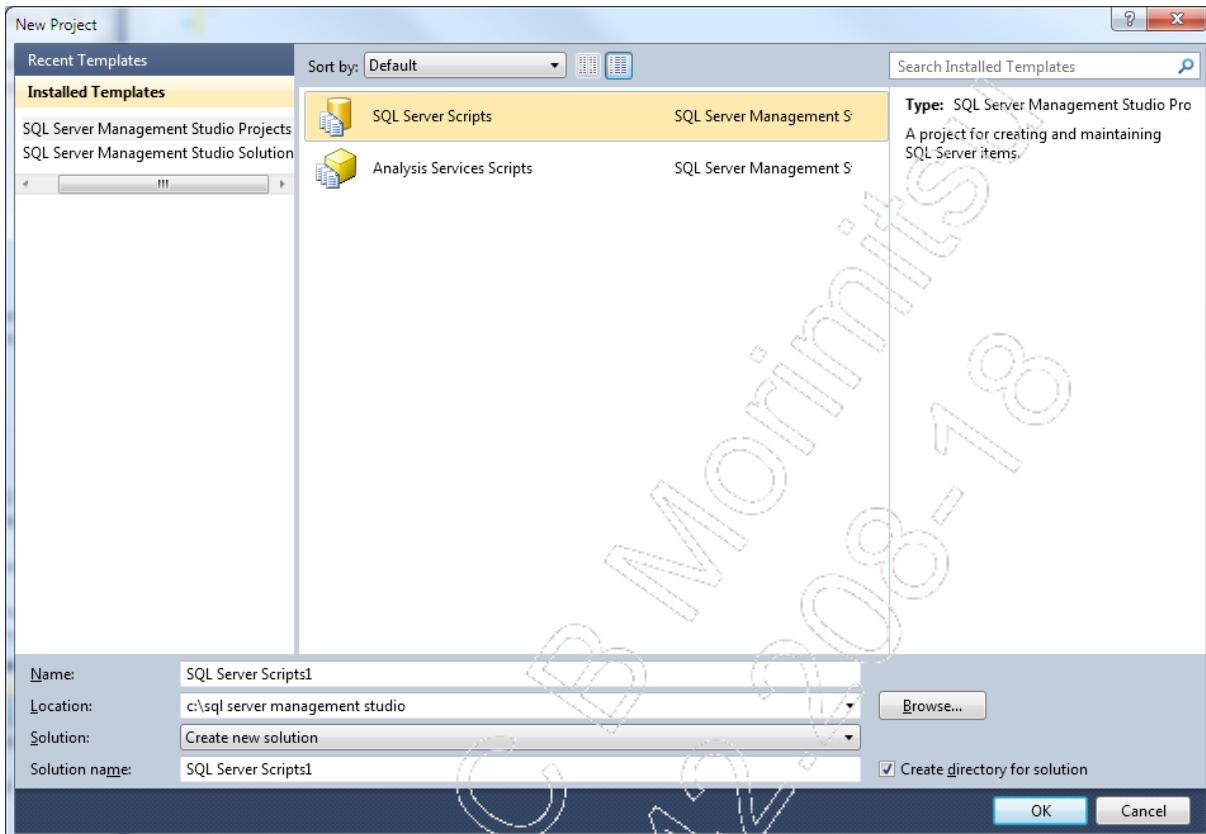
- Criando uma solução

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:



## 1.12. Comandos básicos

A seguir, conheceremos os comandos básicos de nomenclatura, comentários, execução de comandos, resultado e Design Query in Editor.

### 1.12.1. Nomenclatura

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.

### 1.12.2. Comentários

Todo script deve possuir comentários que ajudarão a entender e documentar as instruções. A cor do comentário é 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  
*/
```

### 1.12.3. Executando comandos

Para executar um comando, você pode utilizar a tecla F5 ou o botão mostrado a seguir:



Execute

```
SELECT GETDATE();
```

## 1.12.4. Resultado

O resultado será apresentado abaixo da área de escrita dos comandos:

Results	Messages
(No column name)	
1	2019-05-17 09:22:07.140

Na aba **Results** é apresentado o resultado das consultas (SELECT), e na **Messages** são apresentadas as mensagens e erros.

```
PRINT 'Olá/'
```

Messages
Olá

```
SELECT GETDATE;
```

Messages
Msg 207, Level 16, State 1, Line 27 Invalid column name 'GETDATE'.

O resultado pode ser apresentado de três maneiras: texto, grid ou salvar em arquivo. Selecione botões para a escolha do tipo de resultado:



- Em formato texto:

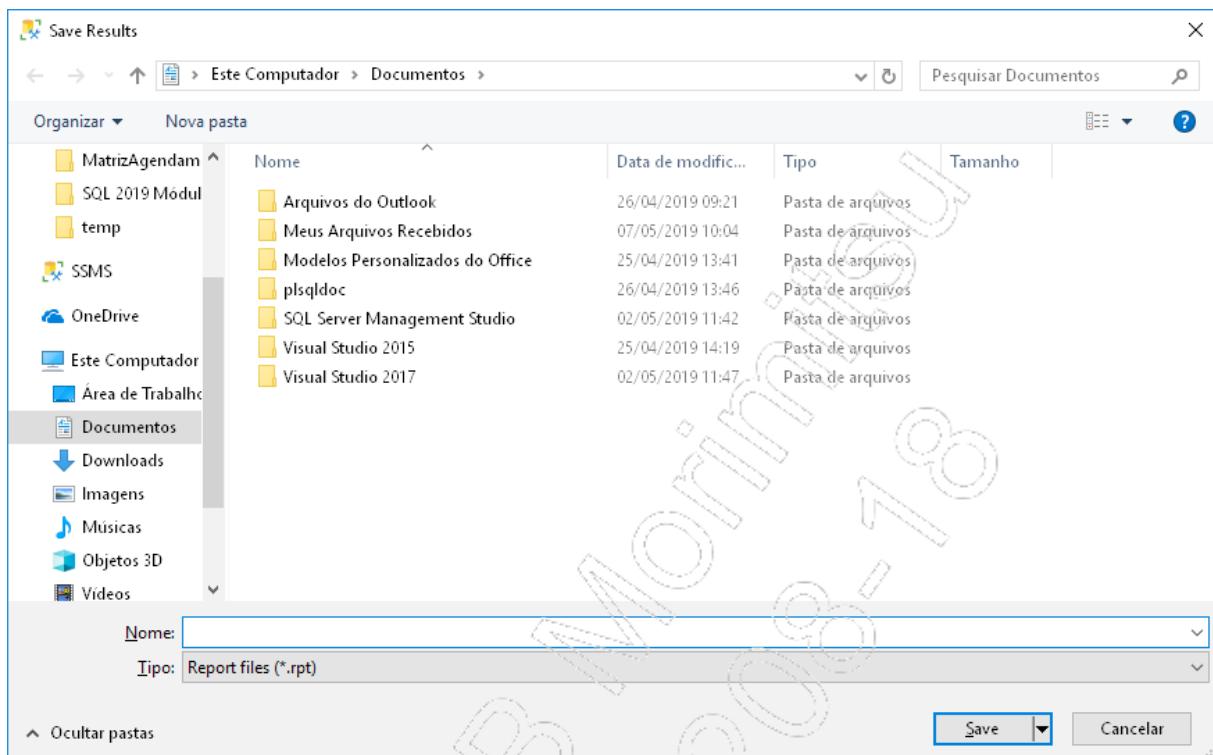
```
SELECT GETDATE();
```

Results
----- 2019-05-17 09:34:44.527 (1 row affected)

- Em formato grid:

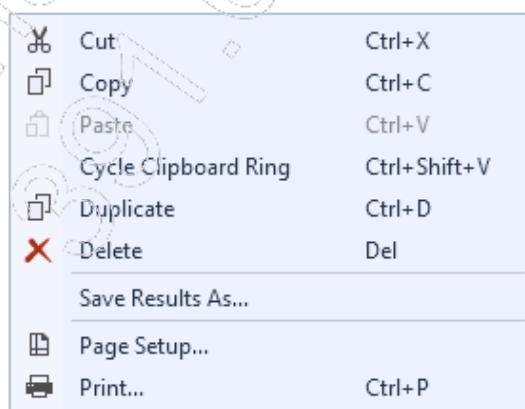
Results	Messages
(No column name)	
1	2019-05-17 09:35:24.967

- Com saída em arquivo, será apresentada uma tela para a gravação do resultado em formato RPT:



## 1.12.4.1. Salvando resultados

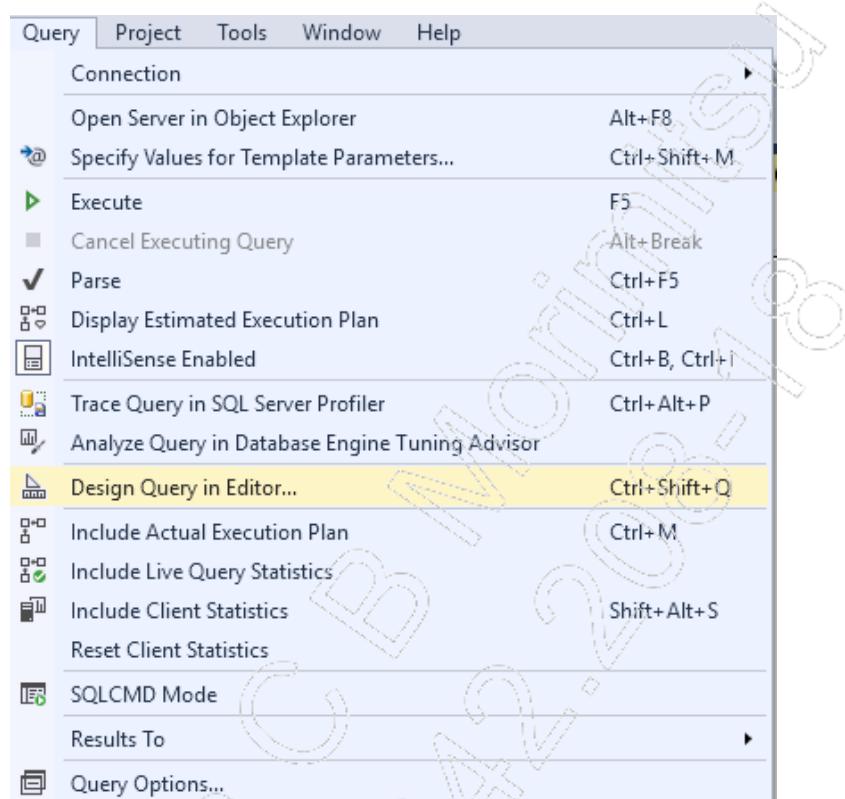
É possível copiar ou salvar os resultados. Para isso é necessário selecionar com o botão direito:



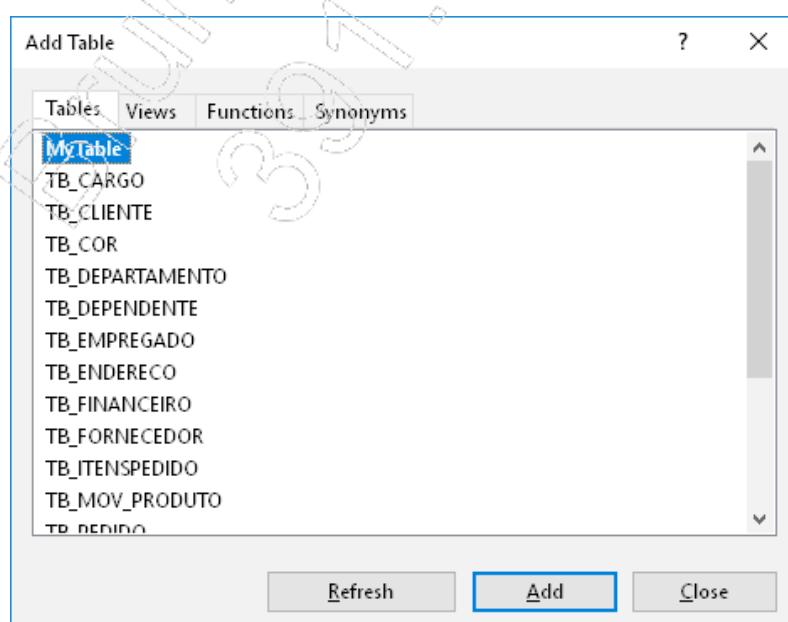
## 1.12.5. Design Query in Editor

O **Design Query in Editor** habilita uma interface gráfica amigável para a construção de consultas no SQL Server.

Para utilizá-lo, é necessário selecionar o menu **Query / Design Query in Editor**:

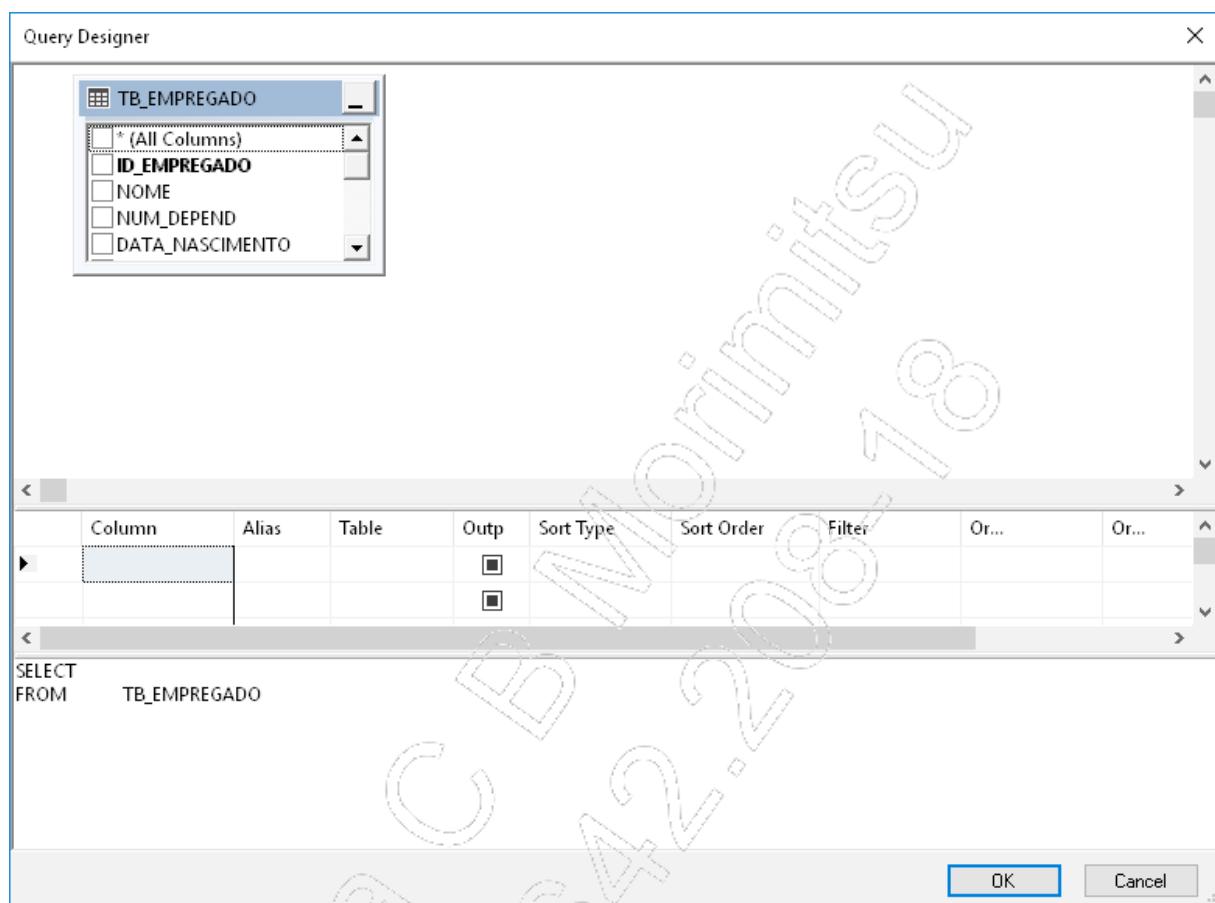


Na primeira tela são apresentadas as listas de tabelas, views, funções e sinônimos:

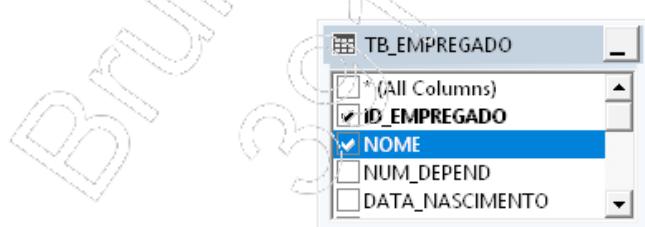


Selecione a(s) tabela(s) que deseja utilizar para a consulta. No exemplo podemos usar a tabela **TB\_EMPREGADO**.

Marque a tabela, selecione os botões **Add** e, depois, **Close**.



Selecione os campos que deseja apresentar. Marque **ID\_EMPREGADO** e **NOME**:



Na coluna **Sort Type** é possível escolher qual será a ordenação da consulta.

Column	Alias	Table	Outp	Sort Type	
ID_EMPREGA		TB_EMPR...	<input checked="" type="checkbox"/>		
NOME		TB_EMPR...	<input checked="" type="checkbox"/>	Descending	

Os dados podem ser filtrados a partir da coluna **Filter**.

A consulta está montada na parte inferior.

```
SELECT ID_EMPREGADO, NOME  
FROM TB_EMPREGADO  
ORDER BY NOME DESC
```

Ao selecionar **OK** a consulta é copiada para sua área de codificação.

```
SQLQuery51.sql - loc...rce (instrutor (54)) *  
USE db_Ecommerce  
SELECT * FROM TB_CLIENTE  
SELECT ID_EMPREGADO, NOME  
FROM TB_EMPREGADO  
ORDER BY NOME DESC
```

### Pontos principais

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

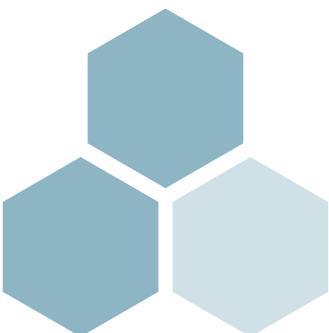
- Um banco de dados armazena informações e seu principal objeto são as tabelas;
- Uma Instance é uma instalação completa de um SQL Server que possui bancos, serviços e usuários independentes;
- A linguagem T-SQL (Transact-SQL) é baseada na linguagem SQL ANSI, desenvolvida pela IBM na década de 1970;
- 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;
- É possível utilizar o **Design Query in Editor** para facilitar a construção de consultas.



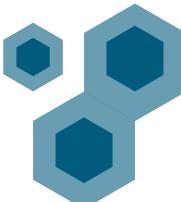
1

# Introdução ao SQL Server 2019

Teste seus conhecimentos



Editora  
**IMPACTA**



## 1. Qual a vantagem de utilizar um banco de dados?

- a) Armazenamento mais lento.
- b) Segurança.
- c) Acesso único no banco de dados.
- d) O espaço não é bem utilizado no banco.
- e) A integração de dados não deve ser realizada.

## 2. O que é uma Instance de SQL Server?

- a) Instance é um banco de dados.
- b) Um banco de dados SQL é uma Instance.
- c) A Instance é uma instalação completa do SQL Server.
- d) Podemos instalar um SQL Server que possui o nome de banco de dados.
- e) Banco de dados e Instance possuem o mesmo significado.

## 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 2019 não utiliza a linguagem SQL.
- c) O SQL Server 2019 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, qual das seguintes afirmações está correta?

- a) É uma plataforma de banco de dados que armazena dados.
- 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





1

# Introdução ao SQL Server 2019

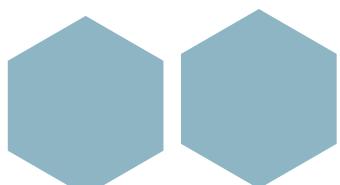


Mãos à obra!

Bruna Morimoto  
397.422-78



Editora  
**IMPACTA**



## Laboratório 1

### 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 passos adiante:

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 por meio do botão **Execute**;
10. Salve o script.

## Laboratório 2

### A – Trabalhando com a tabela TB\_PEDIDO

1. Coloque o banco de dados **db\_Ecommerce** em uso;
2. Selecione o menu **Query**;
3. Selecione **Design Query in Editor**;
4. Escolha a tabela **TB\_PEDIDO** e selecione o botão **Add**;
5. Marque as colunas **ID\_PEDIDO**, **DATA\_EMISSAO** e **VLR\_TOTAL**;
6. Selecione **OK**;
7. Execute a consulta gerada.

## Laboratório 3

### A – Trabalhando com as tabelas TB\_PEDIDO e TB\_CLIENTE

1. Coloque o banco de dados **db\_Ecommerce** em uso;
2. Selecione o menu **Query**;
3. Selecione **Design Query in Editor**;
4. Selecione as tabelas **TB\_PEDIDO** e **TB\_CLIENTE**;
5. Selecione as colunas **Nome**, **ID\_PEDIDO** e **VLR\_TOTAL**;
6. Selecione **OK**;
7. Execute a consulta.



### Laboratório 4

#### A – Trabalhando com as tabelas TB\_PEDIDO, TB\_CLIENTE e TB\_EMPREGADO

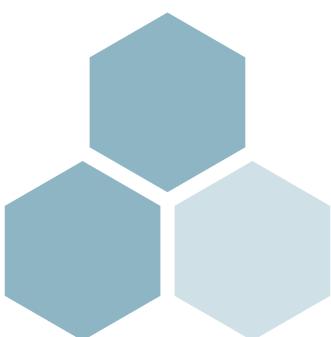
1. Coloque o banco de dados **db\_Ecommerce** em uso;
2. Selecione o menu **Query**;
3. Selecione **Design Query in Editor**;
4. Selecione as tabelas **TB\_PEDIDO**, **TB\_CLIENTE** e **TB\_EMPREGADO**;
5. Selecione as colunas **Nome** da tabela **TB\_CLIENTE**, **Nome** da tabela **TB\_EMPREGADO** e **ID\_PEDIDO**;
6. Selecione **OK**;
7. Execute a consulta.

Bruna@C-B-Morinitsu  
397.642.208-78

# 2

## Banco de dados

- Bancos de dados do sistema;
- Bancos de dados snapshot;
- Criação de um banco de dados;
- Uso do banco de dados;
- Referenciando objetos;
- Objetos de catálogo;
- Grupos de comandos T-SQL.



## 2.1. Introdução

O SQL utiliza o conceito de banco de dados (databases) para organização dos objetos e acessos aos recursos.

Podemos classificar os databases em três tipos:

- Sistemas;
- Snapshot;
- Criado pelo usuário.

## 2.2. Bancos de dados do sistema

Os bancos de dados de gerenciamento são aqueles que permitem ao software gerenciar o ambiente da instance, os databases e usuários. Existem diferentes tipos de bancos de dados de gerenciamento, cada qual com uma finalidade.

No momento em que o SQL Server é instalado, são criados alguns bancos de dados que permitem ao software gerenciar os sistemas de usuários, fazendo a integridade e a segurança dos dados serem mantidas. Esses bancos de dados são chamados de **bancos de dados gerenciais**. Dentro deles são criadas tabelas pelo setup de instalação do SQL Server, as quais recebem o nome de **system tables** (tabelas do sistema).

A seguir, descreveremos cada um dos bancos de dados do sistema.

### 2.2.1. Master

É responsável por registrar todas as informações de sistema, as mensagens de erro do SQL Server e contas de acesso (logins). Também controla qualquer processo em execução no SQL Server.

Caso esse banco esteja indisponível, torna-se inviável iniciar o SQL Server.

O setup de instalação do SQL Server cria objetos utilizados por administradores e pelo próprio SQL Server. Esses objetos compreendem funções, stored procedures estendidas e stored procedures do sistema. A SP\_ADDLOGIN é uma das stored procedures do sistema que podem ser utilizadas pelo administrador e, neste caso, adiciona um login ao sistema.

### 2.2.2. tempdb

Este banco de dados é um recurso global responsável por armazenar qualquer objeto temporário. Exemplos de objetos armazenados neste banco de dados são as tabelas temporárias e as stored procedures temporárias.

O tempdb é considerado um recurso global, visto que armazena tabelas e stored procedures temporárias que podem ser acessadas por todos os usuários que se encontram conectados na(s) instância(s) do SQL Server.

Em um banco de dados tempdb, são alocados os seguintes itens: objetos temporários que foram criados de forma explícita, tabelas internas criadas pelo SQL Server database, versões atualizadas de registros e resultados de ordenações temporárias.

Todas as vezes que o SQL Server é iniciado, o banco de dados tempdb é recriado, o que significa que sempre há uma cópia limpa do banco de dados quando o sistema é iniciado. Dessa forma, quando o sistema é encerrado, não existem conexões ativas, sendo que, entre uma sessão e outra, não há itens a serem salvos em um banco de dados tempdb. Vale destacar que não podemos fazer o backup ou restaurar esse banco de dados.

### 2.2.3. Model

O SQL Server utiliza este banco de dados como modelo para criar qualquer outro banco de dados. Dessa forma, quando solicitamos a criação de um novo banco de dados ao SQL Server, este copia todos os objetos de Model para o novo banco de dados.

A criação de um banco de dados é iniciada pela cópia do conteúdo presente no banco de dados Model. Caso sejam feitas alterações nesse banco de dados, todos aqueles criados posteriormente também refletirão tais modificações.

### 2.2.4. msdb

Este banco de dados registra informações como configurações de replicação e históricos dos jobs. O SQL Server Agent utiliza o msdb para programar a execução de jobs e alertas.

O msdb é utilizado para realizar o armazenamento de dados não apenas pelo SQL Server Agent, mas também pelo SQL Server e SQL Server Management Studio. Um histórico completo de backup e restauração on-line é mantido pelo SQL Server no banco de dados msdb. Esses dados são utilizados pelo SQL Server Management Studio a fim de criar um planejamento de restauração do banco de dados e aplicar backups de log de transação. Mesmo os bancos de dados que são criados com ferramentas de outras empresas ou com aplicações personalizadas têm seus eventos de backup registrados.

Recomenda-se realizar o backup do msdb com frequência, caso esse banco de dados sofra constantes alterações. Caso o msdb seja danificado de alguma forma, as informações referentes ao backup e à restauração serão perdidas, assim como as informações utilizadas pelo SQL Server Agent.

### 2.2.5. Resource

Este é um banco de dados do sistema. Trata-se de um banco de dados apenas de leitura, que possui todos os objetos do sistema e é utilizado pelo próprio SQL Server para fazer upgrade de uma aplicação para a nova versão do software.

O Resource não permanece visível junto com os outros bancos de dados no SQL Server Management Studio. Seus arquivos de dados e de transações estão normalmente disponíveis em: C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data.

## 2.3. Bancos de dados snapshot

Um banco de dados Snapshot é uma cópia de um banco de dados e é somente leitura, não permitindo atualizações diretas.

A sua utilização pode variar conforme a necessidade de cada implementação, podendo ser destacados alguns benefícios:

- Banco estático com informações de um determinado período;
- Banco para extração de consultas e relatórios;
- Estado preservado antes de atualizações.

**! Lembre-se de analisar o espaço disponível, pois esta solução consome muito recurso de disco.**

## 2.4. Criação do banco de dados

Um banco de dados criado pelo usuário é o agrupamento dos objetos que compõem uma solução de negócio. Podemos criar diversos bancos dentro de um servidor SQL e definirmos permissões de acesso distintas para cada um deles.

Nessa estrutura podemos criar objetos e armazenar os dados necessários para atender à necessidade operacional.

Os arquivos que vão ser criados para o banco de dados são estes:

Tipo	Descrição	Extensão
Dados	Armazena os dados e índices das tabelas. O SQL precisa que um arquivo seja o principal. Nesse arquivo, além dos dados, são armazenadas as informações da localização dos arquivos, usuários e mensagens.	MDF
LOG	Arquivo sequencial que armazena as transações que serão encaminhadas para as páginas de dados.	LDF
Criados pelo usuário	Criados para melhoria de performance e divisão dos dados em discos diferentes.	NDF

Para a criação de um banco de dados, é necessário ter permissão específica (sysadmin ou dbcreator).

**Atenção!** A criação de banco de dados e a distribuição de arquivos serão detalhadas no treinamento de SQL Server Módulo III.

### 2.4.1. Comando T-SQL

O comando CREATE DATABASE tem a funcionalidade de criação de um banco de dados. Sua 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 DB_SALAS;
```

Essa instrução criará dois arquivos:

- **DB\_SALAS.MDF**: Este arquivo armazenará os dados de tabelas, índices e as informações da localização dos arquivos e usuários;
- **DB\_SALAS\_LOG.LDF**: Armazena os logs de transações.

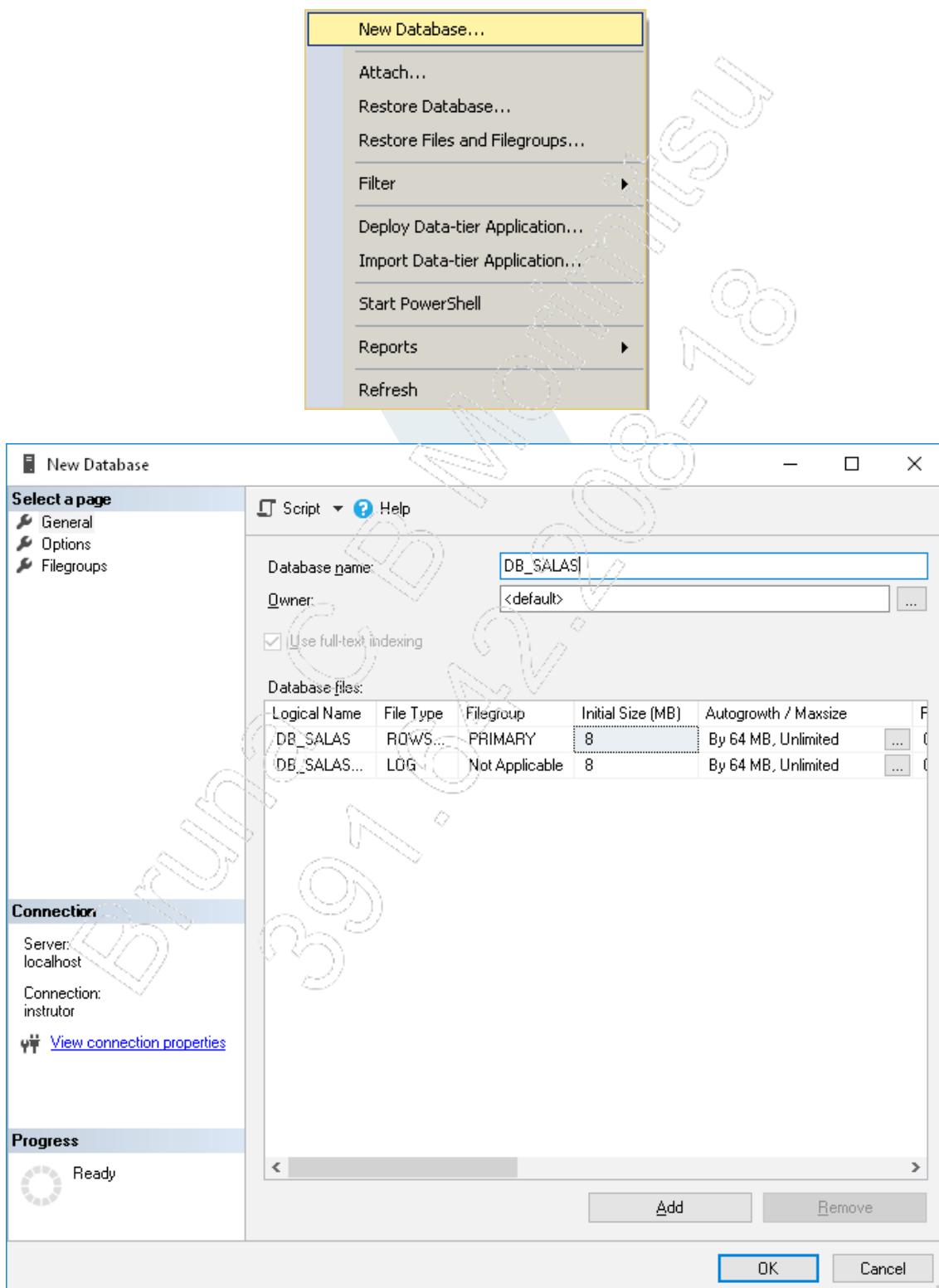
Normalmente, esses arquivos estão localizados na pasta **SQL\_DATA** dentro do diretório de instalação do SQL Server.

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

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

## 2.4.2. Criação de banco de dados graficamente

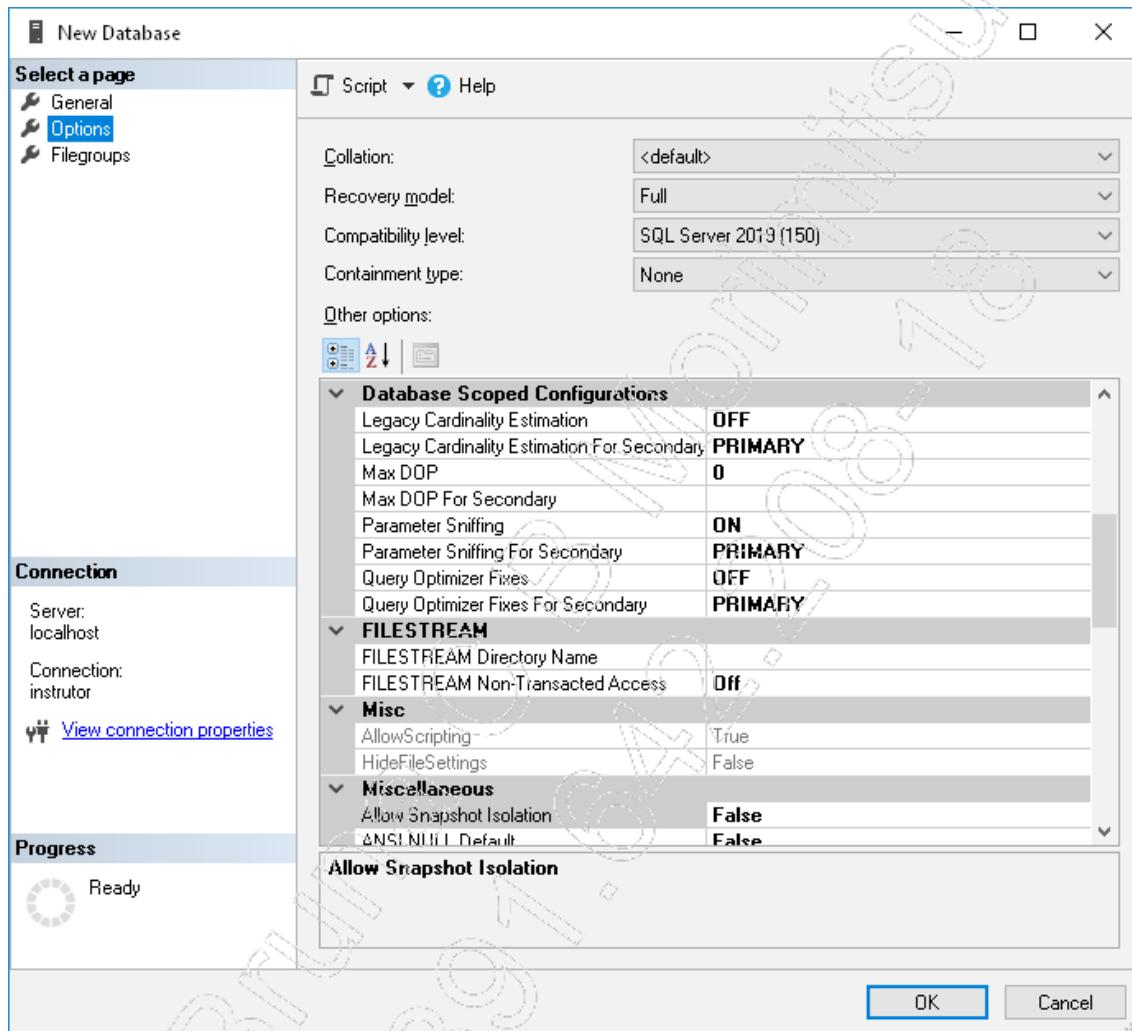
Também é possível criar um banco de dados graficamente. Por meio do SSMS, em **Databases**, com o botão direito, selecione **New Database**:



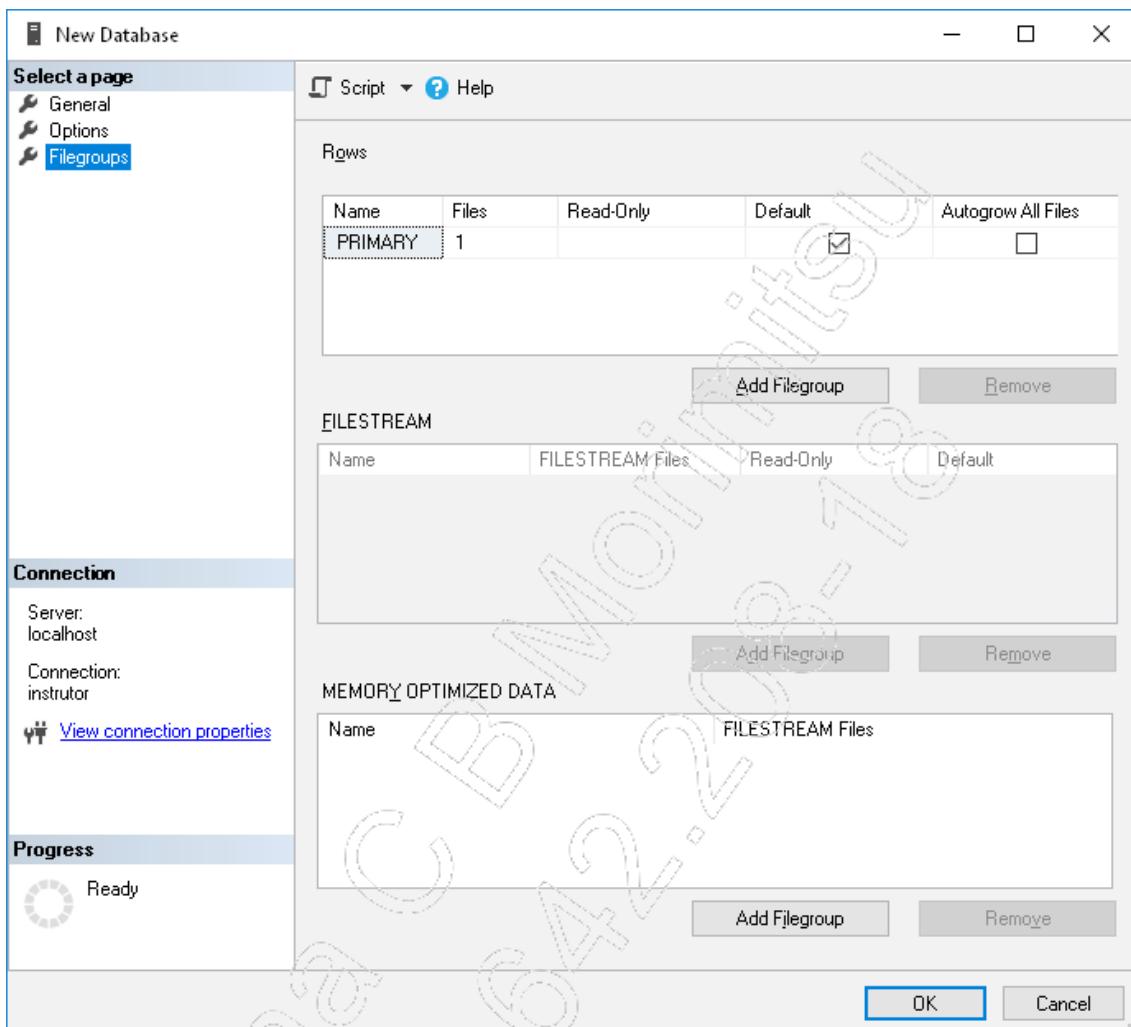
É importante que seja definida a localização dos arquivos que compõem o banco para o seu melhor gerenciamento.

No exemplo anterior, os arquivos serão gravados no caminho: E:\SQL\_DATA\ e E:\SQL\_LOG\.

Na aba **Options** são definidas as informações de configurações do banco:



Em **Filegroups** serão criados os nomes lógicos (Filegroups) que são utilizados para associar os arquivos de dados com os objetos tabelas:



O detalhamento será apresentado no treinamento de SQL Server Módulo III.

## 2.5. Uso do banco de dados

Para acessar os objetos contidos no banco de dados **DB\_SALAS** é necessário informar o **nome do banco + schema** (proprietário do objeto) + **objeto**.

A seguir, um exemplo de consulta realizada no banco **DB\_SALAS**, com o schema **SYS** e na view **ALL\_COLUMNS**:

```
SELECT * FROM DB_SALAS.SYS.ALL_COLUMNS;
```

Caso esteja conectado no banco, não é obrigatório informar o nome do banco.

Colocar o banco em uso é possível utilizando o comando **USE** ou graficamente:

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

Para o banco criado anteriormente:

```
USE DB_SALAS;
```

Por meio do SSMS também é possível alterar o banco corrente da consulta. Observe que, na parte superior esquerda do SSMS, existe um combo box que mostra o nome do banco de dados que está em uso:



## 2.6. Referenciando objetos

Um objeto do SQL Server pode ser referenciado com seu nome inteiro, que o qualifica totalmente, ou com parte de seu nome. Nesta situação, o SQL Server determina o restante do nome de acordo com o contexto em que estiver trabalhando.

- **Nomes totalmente qualificados**

São quatro os identificadores que compõem o nome completo de um objeto, a destacar: nome do servidor, nome do banco de dados do objeto, nome do schema do objeto e nome do objeto em si. O nome composto por todos esses itens é denominado **nome totalmente qualificado**.

Cada um dos objetos que é criado no SQL Server deve apresentar somente um nome totalmente qualificado, ou seja, o SQL Server não permite a duplicação de nomes totalmente qualificados. Além disso, os nomes das colunas presentes em uma mesma tabela, ou views presentes em um mesmo banco de dados, devem ser especificados de maneira única.

- **Nomes parcialmente qualificados**

Embora tenhamos os nomes totalmente qualificados para os objetos do SQL Server, podemos omitir identificadores intermediários desses nomes, desde que eles sejam indicados por pontos, conforme podemos observar nas formas descritas a seguir:

- **Servidor.banco\_de\_dados.schema.objeto;**
- **Servidor.banco\_de\_dados.objeto;**
- **Banco\_de\_dados.schema.objeto;**

- Banco\_de\_dados.objeto;
- Schema.objeto;
- Objeto.

Quando criamos um objeto e omitimos alguns identificadores de seu nome, o SQL Server utiliza alguns padrões para determinar as partes que foram omitidas. Vejamos, a seguir, quais são esses padrões:

- Servidor padrão = Servidor local;
- Banco de dados padrão = Banco de dados atual;
- Schema = Trata-se de um objeto cuja finalidade é representar um contexto de posse para um objeto de banco de dados do SQL Server.

Normalmente, as quatro partes que compõem o nome do objeto são utilizadas para queries distribuídas ou chamadas stored procedures remotas. Grande parte da referência a objetos utiliza apenas três partes do nome e o padrão determinado para o nome do servidor.

## 2.7. Objetos de catálogo

O SQL possui diversos objetos (view, procedures e funções) que possibilitam a extração de informações de metadados, que são as informações da definição dos objetos do banco de dados.

### 2.7.1. Metadados

**Metadados** são os dados que compõem toda a estrutura de um banco de dados. Eles ficam armazenados nas tabelas de sistema, como SYSOBJECTS e SYSCOLUMNS. Podem ser definidos como sendo informações que retratam os objetos referenciados. A palavra **metadados** possui o seguinte significado: informação estruturada a respeito de recursos de informação. Em outras palavras, são dados a respeito de dados.

No SQL Server, os metadados podem ser obtidos por meio de:

- Leitura dos dados das tabelas do sistema;
- Execução de determinadas views, procedures ou funções.

Todo metadado é disponibilizado como uma view de catálogo do sistema. Para isso, usamos o seguinte código:

```
SELECT * FROM SYS.OBJECTS;
```

Na coluna TYPE é apresentado o tipo do objeto. Vejamos a seguir uma lista com esses tipos:

Tipo	Descrição
AF	Função de agregação
C	CONSTRAINT: CHECK
D	CONSTRAINT: DEFAULT
F	CONSTRAINT: FOREIGN KEY
FN	Função escalar
FS	Função escalar: Assembly (CLR)
FT	Função tabular: Assembly (CLR)
IF	Função IN-LINED
IT	Tabela Interna
L	LOG
P	Procedimento armazenado
PC	Procedimento armazenado: Assembly (CLR)
PK	CONSTRAINT: PRIMARY KEY
RF	Procedimento armazenado: Replication filter
S	Tabela de Sistema
SN	Sinônimo
SQ	Service queue
TA	DML trigger: Assembly (CLR)
TF	Função tabular
TR	DML Trigger
TT	Tipo Table
U	Tabela de usuário
UQ	CONSTRAINT: UNIQUE
V	View
X	Procedimento armazenado estendido

A seguir, alguns objetos para extração das definições do banco de dados.

## 2.7.2. Informações do banco de dados

A procedure que retorna as informações dos bancos na instância:

```
EXEC SP_HELPDB;
```

	name	db_size	owner	dbid	created	status	compatibility_level
1	db_Ecommerce	71.13 MB	usr_impacta	6	Apr 30 2019	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150
2	DB_SALAS	16.00 MB	instrutor	5	May 17 2019	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150
3	master	6.69 MB	sa	1	Apr 8 2003	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150
4	model	16.00 MB	sa	3	Apr 8 2003	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150
5	msdb	15.50 MB	sa	4	Apr 15 2019	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150
6	tempdb	40.00 MB	sa	2	May 7 2019	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150

Informando o nome do banco após o comando, serão apresentadas somente as informações do banco solicitado:

```
EXEC SP_HELPDB DB_SALAS;
```

	name	db_size	owner	dbid	created	status	compatibility_level
1	DB_SALAS	16.00 MB	instrutor	5	May 17 2019	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	150

	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	DB_SALAS	1	C:\Program Files\Microsoft SQL Server\MSSQL15.MSS...	PRIMARY	8192 KB	Unlimited	65536 KB	data only
2	DB_SALAS_log	2	C:\Program Files\Microsoft SQL Server\MSSQL15.MSS...	NULL	8192 KB	2147483648 KB	65536 KB	log only

A função que retorna o nome do banco de dados é esta:

```
SELECT DB_NAME();
```

	(No column name)
1	db_Ecommerce

A procedure que retorna os arquivos do banco de dados é a seguinte:

```
EXEC SP_HELPFILE;
```

	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	DB_SALAS	1	C:\Program Files\Microsoft SQL Server\MSSQL15.MSS...	PRIMARY	8192 KB	Unlimited	65536 KB	data only
2	DB_SALAS_log	2	C:\Program Files\Microsoft SQL Server\MSSQL15.MSS...	NULL	8192 KB	2147483648 KB	65536 KB	log only

Por meio da consulta também é possível:

```
SELECT * FROM SYSFILES;
```

	fileid	groupid	size	maxsize	growth	status	perf	name	filename
1	1	1	1024	-1	8192	2	0	DB_SALAS	C:\Program Files\Microsoft SQL Server\MSSQL15.MSS...
2	2	0	1024	268435456	8192	66	0	DB_SALAS_log	C:\Program Files\Microsoft SQL Server\MSSQL15.MSS...

Apresentando o tamanho do banco atual:

```
EXEC SP_SPACEUSED;
```

	database_name	database_size	unallocated space
1	DB_SALAS	16.00 MB	5.51 MB

	reserved	data	index_size	unused
1	2552 KB	968 KB	1208 KB	376 KB

### 2.7.2.1. Catálogos do sistema

O catálogo do sistema fornece as seguintes informações para os bancos de dados SQL Server:

- Nome e número das tabelas e exibições em um banco de dados;
- Número de colunas, além do nome, tipo de dados, escala e precisão de cada coluna;
- Restrições definidas para uma tabela;
- Índices e chaves definidos para uma tabela;
- Um conjunto de visualizações que exibem os metadados e estes, por sua vez, descrevem um objeto em uma instância do SQL Server.

Vejamos alguns exemplos:

- Exemplo de tabelas de sistema

```
USE db_Ecommerce

-- TABELAS DE SISTEMA
-- Armazena todos os objetos do banco de dados
SELECT * FROM SYSOBJECTS
-- Tabelas de usuários
SELECT * FROM SYSOBJECTS WHERE XTYPE = 'U'
-- Chaves primárias
SELECT * FROM SYSOBJECTS WHERE XTYPE = 'PK'
-- Tabelas e suas chaves primárias
SELECT T.NAME AS TABELA, PK.NAME AS CHAVE_PRIMARIA
FROM SYSOBJECTS T JOIN SYSOBJECTS PK ON T.id = PK.parent_obj
WHERE T.XTYPE = 'U'

-- Colunas existentes nas tabelas do banco de dados
SELECT * FROM SYSCOLUMNS -- Observe a coluna ID
-- Colunas das tabelas do banco de dados
SELECT T.NAME AS TABELA, C.NAME AS COLUNAS
FROM SYSOBJECTS T JOIN SYSCOLUMNS C ON T.id = C.id
WHERE T.XTYPE = 'U'

-- Colunas de uma tabela do banco de dados
SELECT T.NAME AS TABELA, C.NAME AS COLUNAS
FROM SYSOBJECTS T JOIN SYSCOLUMNS C ON T.id = C.id
WHERE T.XTYPE = 'U' AND T.name = 'TB_PEDIDO'

-- Mostrar também o tipo de cada coluna
SELECT T.NAME AS TABELA, C.NAME AS COLUNAS, C.XTYPE, DT.NAME
FROM SYSOBJECTS T JOIN SYSCOLUMNS C ON T.id = C.id
JOIN SYSTYPES DT ON C.XTYPE = DT.XTYPE
WHERE T.XTYPE = 'U' AND T.name = 'TB_PEDIDO'

-- Mais informações sobre a estrutura da tabela
SELECT T.NAME AS TABELA, C.NAME AS COLUNAS, C.XTYPE, DT.NAME,
C.LENGTH AS BYTES, C.XPREC AS PRECISAO, C.XSCALE AS
DECIMAIS
FROM SYSOBJECTS T JOIN SYSCOLUMNS C ON T.id = C.id
JOIN SYSTYPES DT ON C.XTYPE = DT.XTYPE
WHERE T.XTYPE = 'U' AND T.name = 'TB_PEDIDO'
```

- Exemplo de views de catálogo

```
-- VIEWS DE CATÁLOGO
-- TABELAS DE USUÁRIO
SELECT * FROM SYS.TABLES
-- COLUMNAS DE CADA TABELA
SELECT * FROM SYS.COLUMNS
-- TIPOS DE DADOS
SELECT * FROM SYS.TYPES
-- ÍNDICES
SELECT * FROM SYS.INDEXES
-- CAMPOS IDENTIDADE
SELECT * FROM SYS.IDENTITY_COLUMNS
-- CHAVES ÚNICAS E CHAVES PRIMÁRIAS
SELECT * FROM SYS.KEY_CONSTRAINTS
```

- Procedures que retornam metadados

Procedure	Descrição
<b>SP_HELPDEVICE</b>	Dispositivos de backup.
<b>SP_HELP</b>	Objetos do banco de dados.
<b>SP_HELPCONSTRAINT</b>	CONSTRAINTS.
<b>sp_helpextendedproc</b>	Procedimentos armazenados estendidos atualmente definidos, bem como o nome da biblioteca de vínculo dinâmico (DLL).
<b>sp_helpfile</b>	Os nomes físicos e os atributos de arquivos do banco de dados.
<b>sp_helpdb</b>	Banco de dados.
<b>sp_helpdbfixedrole</b>	Roles do banco de dados.
<b>sp_helpfilegroup</b>	Nomes e os atributos de grupos de arquivos.
<b>sp_helpsrvrole</b>	Roles do servidor de banco de dados.
<b>sp_helptrigger</b>	Trigger DML definidos para a tabela.
<b>sp_helpuser</b>	Usuários do banco de dados.
<b>sp_helpstats</b>	Estatísticas sobre colunas e índices.
<b>sp_helptext</b>	A definição de procedures, funções, views etc.
<b>sp_helplogins</b>	Logons e usuários associados em cada banco de dados.
<b>sp_helpmember</b>	Membros de uma função no banco de dados atual.
<b>sp_helpntgroup</b>	Grupos do Windows com contas no banco de dados atual.

Procedure	Descrição
<code>sp_helprotect</code>	Permissões para um objeto ou permissões de instrução, no banco de dados atual.
<code>sp_helpremotelogin</code>	Logons remotos para um determinado servidor remoto, ou para todos os servidores, definido no servidor local.
<code>sp_helpserver</code>	Servidor remoto ou réplica, ou sobre todos os servidores de ambos os tipos.
<code>sp_helprole</code>	Funções no banco de dados atual.
<code>sp_helpsort</code>	COLLATION.
<code>sp_helpindex</code>	Índices em uma tabela ou view.
<code>sp_HELPLANGUAGE</code>	Idioma alternativo específico ou sobre todos os idiomas.
<code>sp_helplinkedsrvlogin</code>	Mapeamentos de logon definidos em um servidor vinculado e procedimentos armazenados remotos.

- Exemplo de procedures de catálogo

```
-- TABELAS DO BANCO DE DADOS
EXEC SP_TABLES
-- COLUNAS DE UMA TABELA
EXEC SP_COLUMNS TB_PEDIDO
-- CAMPOS QUE FORMAM A CHAVE PRIMÁRIA DE UMA TABELA
EXEC SP_PKEYS TB_ITENSPEDEIDO
-- ESTRUTURA DE UMA TABELA
EXEC SP_HELP TB_PEDIDO
```

- Funções que retornam metadados

Função	Descrição
<b>@@PROCID</b>	ID do módulo atual do T-SQL.
<b>COL_LENGTH</b>	Tamanho definido para uma coluna.
<b>COL_NAME</b>	Nome da coluna a partir de um ID.
<b>COLUMNPROPERTY</b>	Informações sobre uma coluna.
<b>DATABASE_PRINCIPAL_ID</b>	ID de um principal do banco de dados.
<b>DATABASEPROPERTY</b>	Propriedades do banco de dados.
<b>DB_ID</b>	ID do banco de dados.
<b>DB_NAME</b>	Nome do banco de dados.
<b>FILE_NAME</b>	Nome lógico de um arquivo.
<b>FILEGROUP_ID</b>	ID de um filegroup.
<b>FILEGROUP_NAME</b>	Nome de um filegroup.
<b>FILEGROUOPROPERTY</b>	Propriedades de um filegroup.
<b>FILEPROPERTY</b>	Propriedades de um arquivo.
<b>INDEX_COL</b>	Nome da coluna indexada.
<b>Key_ID</b>	ID da chave simétrica do banco corrente.
<b>KEY_NAME</b>	Nome da chave simétrica do banco corrente.
<b>OBJECT_ID</b>	ID de um objeto.
<b>OBJECT_NAME</b>	Nome de um objeto.
<b>OBJECT_SCHEMA_NAME</b>	Nome do schema do objeto.
<b>OBJECTPROPERTY</b>	Propriedades de um objeto.
<b>SCHEMA_ID</b>	ID do schema.
<b>TYPE_ID</b>	ID de um tipo de dados.
<b>TYPE_NAME</b>	Nome de um tipo de dados.

- Exemplo de funções

```
--FUNÇÕES QUE RETORNAM DADOS DO SISTEMAS
--NOME DO BANCO
SELECT DB_NAME()
--LISTA O NOME DOS OBJETOS
SELECT OBJECT_NAME(ID) FROM SYSOBJECTS
```

## 2.8. Grupos de comandos T-SQL

Os comandos da linguagem T-SQL são divididos em quatro grupos, a destacar:

- **DCL (Data Control Language)**
  - **GRANT**: Este comando é utilizado para conceder permissões;
  - **REVOKE**: Este comando é utilizado para revogar a concessão ou a negação de permissões;
  - **DENY**: Este comando é utilizado para negar permissões.
- **DDL (Data Definition Language)**
  - **ALTER**: Este comando é utilizado para alterar a estrutura que os objetos apresentam no sistema;
  - **CREATE**: Este comando é utilizado para criar objetos no sistema;
  - **DROP**: Este comando é utilizado para excluir objetos do sistema;
  - **TRUNCATE TABLE**: Este comando é utilizado para excluir todas as linhas de uma tabela e não registra como elas foram removidas individualmente.
- **DML (Data Manipulation Language)**
  - **BACKUP**: Utilizado para fazer o backup dos dados;
  - **BULK INSERT**: Comando utilizado para incluir uma grande quantidade de dados na tabela;
  - **DELETE**: Exclui os dados presentes na tabela;
  - **INSERT**: Insere dados em uma tabela;
  - **RESTORE**: Comando utilizado para restaurar os dados de um backup;
  - **SELECT**: Realiza a leitura de views e dados de tabelas;
  - **UPDATE**: Altera os dados de uma tabela.
- **DTL (Data Transaction Language)**
  - **BEGIN TRANSACTION**: Abre uma transação explícita;
  - **COMMIT**: Confirma uma transação;
  - **ROLLBACK**: Cancela uma transação.

## Pontos principais

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

- Os objetos que fazem parte de um sistema são criados dentro de um objeto denominado **database**, ou seja, uma estrutura lógica formada por dois tipos de arquivo: um responsável pelo armazenamento de dados, e outro que armazena as transações feitas. Para que um banco de dados seja criado no SQL Server, é necessário utilizar a instrução **CREATE DATABASE**;
- Os dados de um sistema são armazenados em objetos denominados tabelas (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;
- Os **bancos de dados de gerenciamento** são aqueles que permitem ao software gerenciar os sistemas dos usuários. Existem cinco diferentes tipos de bancos de dados de gerenciamento, cada qual com uma finalidade: **master**, **tempdb**, **model**, **msdb** e **resource**;
- São chamados de **catálogos** os recursos existentes para extraírmos metadados do banco de dados, como as tabelas de catálogo, as views de catálogo e as procedures de catálogo. Os catálogos do sistema oferecem um conjunto de visualizações que exibem os metadados e estes, por sua vez, descrevem um objeto em uma instância do SQL Server;
- Os processos de desenvolvimento e gerenciamento de um aplicativo são afetados pela arquitetura utilizada com a finalidade de implementar um sistema. Os tipos de arquiteturas utilizados são: **servidor inteligente**, **cliente inteligente**, **Internet** e **sistema de N camadas**.

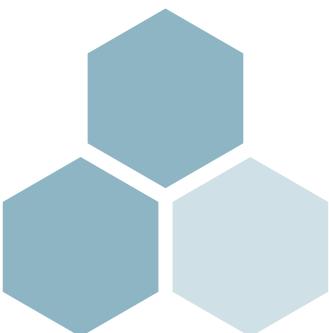




2

# Banco de dados

Teste seus conhecimentos



**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 banco de dados não é sistema?**

- a) MASTER
- b) INTERNAL
- c) TEMPDB
- d) MODEL
- e) RESOURCE

**3. A que grupo de comando do SQL pertence o SELECT?**

- a) DML
- b) DDL
- c) DCL
- d) DTL
- e) DSL

## 4. Qual a vantagem da utilização de catálogos de sistemas?

- a) Não possui vantagem, pois podemos usar o OBJECT EXPLORER.
- b) É mais rápido que o modo gráfico.
- c) Apresenta dados básicos.
- d) Permite o acesso ao metadado do objeto.
- e) Podemos consultar dados.

## 5. Qual a vantagem de um banco Snapshot?

- a) Não existe vantagem.
- b) O melhor é usar um Backup/Restore.
- c) É um banco estático.
- d) É recomendado usar somente o banco original.
- e) É um banco estático que pode ser usado para consultas.





2

# Banco de dados

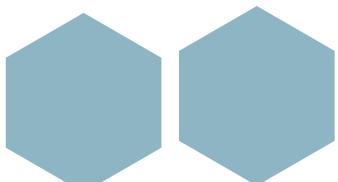


Mãos à obra!

Bruna C. Morimoto  
397.642-008-78



Editora  
**IMPACTA**



## Laboratório 1

### A – Criando um banco de dados

1. Acesse o SSMS e realize a conexão;
2. Selecione o botão **New Query**;
3. Crie um banco de nome **DB\_SALA**;
4. Coloque em uso o banco **DB\_SALA**;
5. Por meio de funções, procedures, views ou tabelas, retorne as informações adiante:
  - Nome do banco;
  - Lista dos arquivos do banco de dados;
  - Lista dos objetos do banco de dados;
  - Lista dos logins.
6. Realize o mesmo procedimento utilizando o Object Explorer do SSMS.

## Laboratório 2

### A – Criando outro banco de dados

1. Acesse o SSMS e realize a conexão;
2. No Object Explorer do SSMS, sob **Databases**, selecione com o botão direito **New Database**. A tela de criação de banco de dados será apresentada;
3. No campo **Database Name** informe o nome do banco: **DB\_SALA1**;
4. Selecione **OK**;
5. Verifique no Object Explorer do SSMS se foi criado o banco de dados;
6. Por meio de funções, procedures, views ou tabelas, retorne as informações adiante:
  - Nome do banco;
  - Lista dos arquivos do banco de dados;
  - Lista dos objetos do banco de dados;
  - Lista dos logins.
7. Realize o mesmo procedimento utilizando o Object Explorer do SSMS.

## Laboratório 3

### A – Utilizando os objetos de catálogo

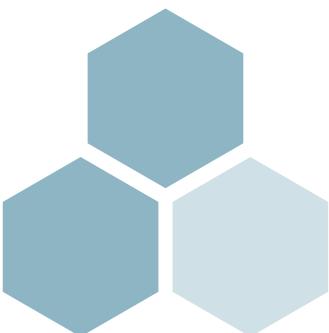
1. Coloque o banco **db\_Ecommerce** em uso;
2. Por meio de funções, procedures, views ou tabelas, retorne as informações a seguir:
  - Liste as tabelas de usuário do banco de dados;
  - Liste os campos da tabela **TB\_CLIENTE**;
  - Apresente os objetos do tipo = 'V';
  - Verifique a estrutura da tabela **TB\_Pedido**.



# 3

## Consultando dados

- ◆ SELECT;
- ◆ Ordenando dados;
- ◆ Filtrando consultas;
- ◆ Operadores lógicos;
- ◆ Intervalos de valores;
- ◆ Pesquisa em campo texto;
- ◆ Lista de elementos;
- ◆ Valores nulos;
- ◆ Funções para tratamento de nulos;
- ◆ Campos data e hora.

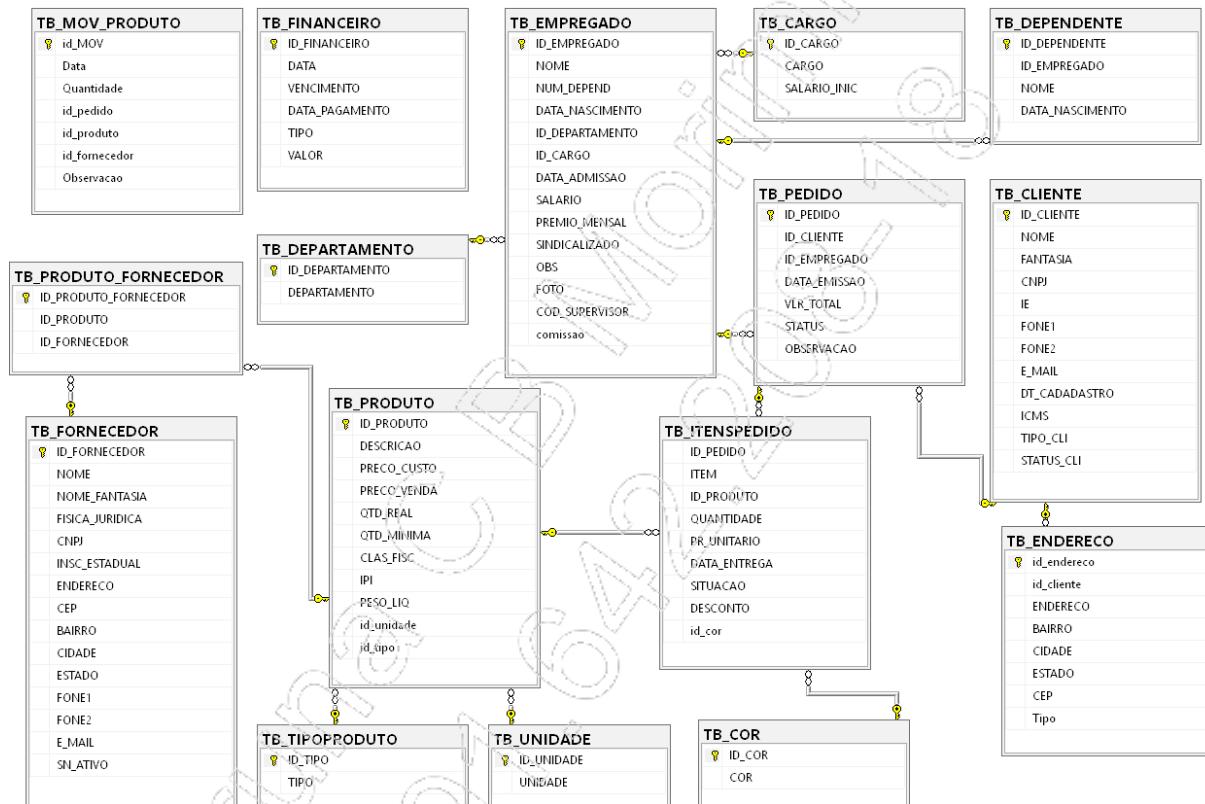


## 3.1. Introdução

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

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

- **Diagrama do banco db\_Ecommerce**



## 3.2. 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**).

O comando possui as seguintes cláusulas:

- **SELECT**: Informa para o SQL que o comando será de consulta. Após a palavra **SELECT** será necessário informar as colunas que serão apresentadas nas colunas:
  - **<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.
- **FROM**: Esta cláusula define qual será a fonte de dados para a extração das informações. Pode ser tabelas, tabelas temporárias, views, conjunto de dados externos, entre outros;
- **WHERE**: A cláusula **WHERE** aplica uma condição de filtro que determinará quais linhas farão parte do resultado;
- **GROUP BY**: 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;
- **HAVING**: A cláusula **HAVING** define uma condição de busca para o grupo de linhas a ser retornado por **GROUP BY**;
- **ORDER BY**: A cláusula **ORDER BY** é utilizada para determinar a ordem em que os resultados são retornados. A ordenação pode ser crescente ou decrescente.

O processamento que o SQL Server vai fazer segue esta ordem:

1. FROM;
2. WHERE;
3. GROUP BY;
4. HAVING;
5. SELECT;
6. ORDER BY.

Além das cláusulas mencionadas, também podemos usar as seguintes cláusulas:

- **[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.

- Exemplos:
  - Consulta simples retornando um texto:

```
SELECT 'IMPACTA' AS NOME_EMPRESA
```

Results	
	Messages
1	NOME_EMPRESA IMPACTA

- Consulta simples retornando uma data:

```
SELECT GETDATE() AS DATA_HORA_SERVIDOR
```

Results	
	Messages
1	DATA_HORA_SERVIDOR 2019-04-24 09:46:06.187

- Consulta simples retornando um número:

```
SELECT 123 AS NUMERO
```

Results	
	Messages
1	NUMERO 123

- Consulta buscando informações de uma tabela:

```
SELECT * FROM TB_COR
```

Results	
	Messages
1	ID_COR COR 1 BRANCO
2	2 PRETO
3	3 VERDE
4	4 AZUL

- Consulta ordenando dados:

```
SELECT * FROM TB_COR ORDER BY COR
```

	ID_COR	COR
1	22	AMARELO
2	11	AMARELO CITRICO
3	4	AZUL
4	10	BEGE

### 3.2.1. Informando o nome das colunas

Ao executar um comando de SELECT, é possível informar o asterisco (\*) para retornar todas as colunas da(s) fonte(s) de dados definida(s) na cláusula FROM. Este recurso é prático, porém traz um efeito indesejado, que é o retorno de mais informações do que o necessário, com isso sobrecarregando a transmissão de dados por meio da rede.

Neste exemplo podemos utilizar o asterisco (\*), que apresenta todas as colunas da tabela **TB\_EMPREGADO**:

```
SELECT * FROM TB_EMPREGADO;
```

ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL	SINDICALIZADO
1	OLAVO	1	1960-04-14 00:00:00.000	4	5	2012-02-03 00:00:00.000	3000.00	1200.00	S
2	JOSE	6	1962-08-18 00:00:00.000	2	5	2012-02-03 00:00:00.000	600.00	1250.00	S
3	MARCELO	1	1960-04-14 00:00:00.000	5	5	2012-02-03 00:00:00.000	2400.00	1200.00	S
4	PAULO	2	1962-01-25 00:00:00.000	8	5	2012-02-03 00:00:00.000	600.00	1250.00	S
5	JOAO	2	1965-09-07 00:00:00.000	4	5	2012-02-03 00:00:00.000	1200.00	900.00	S
6	CARLOS	0	1971-05-15 00:00:00.000	11	5	2012-02-03 00:00:00.000	4500.00	1199.53	S
7	ELIANE	0	1964-11-22 00:00:00.000	6	5	2012-02-03 00:00:00.000	1200.00	1237.20	S
8	RUDGE	3	1971-05-31 00:00:00.000	2	4	2012-02-03 00:00:00.000	800.00	1255.28	N

Veja o seguinte exemplo, em que é feita a consulta nas colunas **ID\_EMPREGADO**, **NOME** e **DATA\_NASCIMENTO** da tabela **TB\_EMPREGADO**:

```
SELECT ID_EMPREGADO, NOME, DATA_NASCIMENTO FROM TB_EMPREGADO;
```

Confira o resultado:

ID_EMPREGADO	NOME	DATA_NASCIMENTO
1	OLAVO	1960-04-14 00:00:00.000
2	JOSE	1962-08-18 00:00:00.000
3	MARCELO	1960-04-14 00:00:00.000
4	PAULO	1962-01-26 00:00:00.000
5	JOAO	1965-09-07 00:00:00.000
6	CARLOS	1971-05-15 00:00:00.000
7	ELIANE	1964-11-22 00:00:00.000
8	RUDGE	1971-05-31 00:00:00.000

No próximo exemplo faremos uma consulta que retorne o número do pedido, a data da emissão e o valor total do pedido:

```
SELECT ID_PEDIDO, DATA_EMISSAO, VLR_TOTAL FROM TB_PEDIDO;
```

	ID_PEDIDO	DATA_EMISSAO	VLR_TOTAL
1	1	2016-03-17 01:53:13.497	2.00
2	2	2016-12-17 01:53:13.497	30.70
3	3	2016-12-17 01:53:13.497	59.76
4	4	2016-12-17 01:53:13.497	603.25
5	5	2016-12-17 01:53:13.497	72.39
6	6	2016-12-17 01:53:13.497	970.57
7	7	2016-12-17 01:53:13.497	153.90
8	8	2016-12-17 01:53:13.497	874.00
9	9	2016-12-17 01:53:13.497	3163.50

## 3.2.2. Realizando cálculos

No SQL Server é possível realizar cálculos com os valores das linhas. A vantagem é a utilização de recursos do servidor.

O cálculo é realizado utilizando os seguintes operadores aritméticos:

Operador	Funcionalidade
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

Verifique os exemplos a seguir:

```
SELECT 10 + 4 AS SOMA
```

Results	Messages
<pre>1    14</pre>	
SOMA	

```
SELECT 3 - 4 AS SUBTRACAO
```

Results	Messages
<pre>1    -1</pre>	
SUBTRACAO	

## Consultando dados

```
SELECT 8 * 8 AS MULTIPLICACAO
```

Results		Messages	
	MULTIPLICACAO		
1	64		

```
SELECT 70 / 10 AS DIVISAO
```

Results		Messages	
	DIVISAO		
1	7		

```
SELECT 7 % 6 AS RESTO_DIVISAO
```

Results		Messages	
	RESTO_DIVISAO		
1	1		

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

```
SELECT ID_EMPREGADO, NOME, SALARIO, SALARIO * 1.10 FROM TB_EMPREGADO;
```

Veja o resultado:

Results		Messages		
	ID_EMPREGADO	NOME	SALARIO	(No column name)
1	1	OLAVO	3000.00	3300.0000
2	2	JOSE	600.00	660.0000
3	3	MARCELO	2400.00	2640.0000
4	4	PAULO	600.00	660.0000
5	5	JOAO	1200.00	1320.0000
6	7	CARLOS	4500.00	4950.0000
7	8	ELIANE	1200.00	1320.0000
8	9	RUDGE	800.00	880.0000
9	10	MARIA	1200.00	1320.0000
10	11	FERNANDO	1200.00	1320.0000

Observe que não existe identificação para a coluna calculada.

### 3.2.3. Concatenando textos

Para concatenar um texto ou coluna, utilizamos o sinal de soma (+). Verifique os exemplos:

```
SELECT 'TEXT01' + 'TEXT02' AS TEXTO
```

Results	
	TEXTO
1	TEXT01TEXT02

```
SELECT ENDERECO + ' - ' + BAIRRO + ' - ' + CIDADE + ' / ' +  
ESTADO AS ENDERECO FROM TB_ENDERECHO;
```

	ENDERECO
1	NULL
2	R.COMENDADOR JOSE ZILLO,401 - PQ. DOS OCACIS - ASSI...
3	NULL
4	R.RONDINIA,71 - JD.PROGRESSO - ASSIS CHATEAUBRIAN...
5	AV. MIGUEL ROSA,6296 - SUL MACAUBA - TERESINA/PI
6	AV.PRES.VARGAS,364 - CENTRO - GARIBALDI/RS
7	R.MARIA BENJAMIN, 280 - PILARES - RIO DE JANEIRO/RJ
8	NULL
9	R.ONZE DE JUNHO,791 - CASA BRANCA - SANTO ANDRE/SP
10	R.GUIAN, 393 - VL CAMPESTRE - SAO PAULO/SP

Na consulta apareceram valores nulos. Esses valores serão tratados posteriormente.

### 3.2.4. Renomeando o cabeçalho da coluna

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

Costuma-se utilizar a cláusula AS a fim de facilitar a identificação do alias, no entanto, não é uma obrigatoriedade.

## Consultando dados

Vejamos os seguintes exemplos de consulta com uso de alias:

```
SELECT ID_EMPREGADO AS CODIGO,
       NOME AS EMPREGADO,
       SALARIO AS SALARIO
  FROM TB_EMPREGADO;
```

	CODIGO	EMPREGADO	SALARIO
1	1	OLAVO	3000.00
2	2	JOSE	600.00
3	3	MARCELO	2400.00
4	4	PAULO	600.00
5	5	JOAO	1200.00
6	7	CARLOS ALBERTO	4500.00
7	8	ELIANE	1200.00
8	9	RUDGE	800.00
9	10	MARIA APARECIDA	1200.00

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:

- Usando colchetes:

```
SELECT ID_EMPREGADO      AS Código,
       NOME                AS Nome,
       SALARIO              AS Salário,
       DATA ADMISSÃO        AS [Data de Admissão]
  FROM TB_EMPREGADO;
```

	Código	Nome	Salário	Data de Admissão
1	1	OLAVO	3000.00	2012-02-03 00:00:00.000
2	2	JOSE	600.00	2012-02-03 00:00:00.000
3	3	MARCELO	2400.00	2012-02-03 00:00:00.000
4	4	PAULO	600.00	2012-02-03 00:00:00.000
5	5	JOAO	1200.00	2012-02-03 00:00:00.000
6	7	CARLOS ALBERTO	4500.00	2012-02-03 00:00:00.000
7	8	ELIANE	1200.00	2012-02-03 00:00:00.000
8	9	RUDGE	800.00	2012-02-03 00:00:00.000
9	10	MARIA APARECIDA	1200.00	2012-02-03 00:00:00.000
10	11	FERNANDO	1200.00	2012-02-03 00:00:00.000

- Usando aspas simples:

```
SELECT ID_EMPREGADO      AS Código,  
       NOME                AS Nome,  
       SALARIO              AS Salário,  
       DATA_ADMISSAO        AS 'Data de Admissão'  
  FROM TB_EMPREGADO;
```

- Usando aspas duplas:

```
SELECT ID_EMPREGADO      AS Código,  
       NOME                AS Nome,  
       SALARIO              AS Salário,  
       DATA_ADMISSAO        AS "Data de Admissão"  
  FROM TB_EMPREGADO;
```

- Uma coluna calculada também pode receber um alias:

```
SELECT ID_EMPREGADO,  
       NOME,  
       SALARIO, SALARIO * 1.10 AS SALARIO_MAIS_10_POR_CENTO  
  FROM TB_EMPREGADO;
```

	ID_EMPREGADO	NOME	SALARIO	SALARIO_MAIS_10_POR_CENTO
1	1	OLAVO	3000,00	3300,0000
2	2	JOSE	600,00	660,0000
3	3	MARCELO	2400,00	2640,0000
4	4	PAULO	600,00	660,0000
5	5	JOAO	1200,00	1320,0000
6	7	CARLOS ALBERTO	4500,00	4950,0000
7	8	ELIANE	1200,00	1320,0000
8	9	RUDGE	800,00	880,0000
9	10	MARIA APARECIDA	1200,00	1320,0000

## 3.3. Ordenando dados

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

### 3.3.1. 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.

A próxima consulta retorna as informações dos empregados ordenados pelo campo **NOME**:

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

ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL
58	ALBERTO	0	1988-12-11 00:00:00.000	9	9	2012-02-03 00:00:00.000	3330.00	519.12
16	ALTAMIR	2	1973-03-23 00:00:00.000	6	12	2012-02-03 00:00:00.000	3300.00	1153.50
26	ANA	0	1968-09-03 00:00:00.000	5	10	2012-02-03 00:00:00.000	5000.00	898.30
47	ANA DO CARMO	0	1989-11-09 00:00:00.000	3	4	2012-02-03 00:00:00.000	800.00	253.37
17	ANA MARIA	2	1972-01-05 00:00:00.000	3	3	2012-02-03 00:00:00.000	1200.00	1207.89
57	ANTONIO	0	1999-02-10 00:00:00.000	5	5	2012-02-03 00:00:00.000	500.00	336.65
44	ANTONIO JORGE	0	1990-05-10 00:00:00.000	NULL	NULL	2012-02-03 00:00:00.000	NULL	528.71
--	--	--	--	--	--	--	--	--

A mesma consulta pode ser realizada sem a palavra **ASC**:

```
SELECT * FROM TB_EMPREGADO ORDER BY NOME;
```

A seguir, outros exemplos:

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

### 3.3.2. 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.

Realizando a mesma consulta anterior, porém usando a cláusula **DESC**:

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

ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL
21	SILVIO	0	1959-12-28 00:00:00.000	3	5	2012-02-03 00:00:00.000	500.00	865.15
68	SEVERINO	0	1998-12-20 00:00:00.000	NULL	NULL	2012-02-03 00:00:00.000	NULL	528.71
19	SEBASTIÃO	0	1961-10-27 00:00:00.000	2	1	2012-02-03 00:00:00.000	8300.00	1140.90
9	RUDGE	3	1971-05-31 00:00:00.000	2	4	2012-02-03 00:00:00.000	800.00	1255.28
61	RONALDO	0	1989-11-09 00:00:00.000	1	12	2012-02-03 00:00:00.000	3300.00	1053.91

Outros exemplos:

```
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.

### 3.3.3. 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 SALÁRIO
SELECT ID_EMPREGADO 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 ID_EMPREGADO AS Código,
       NOME AS Nome,
       SALARIO AS Salário,
       SALARIO * 1.10 [Salário com 10% de aumento]
  FROM TB_EMPREGADO
 ORDER BY 3;

-- Pela coluna calculada
SELECT ID_EMPREGADO 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 ID_EMPREGADO 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 ID_EMPREGADO, NOME, DATA_ADMISSAO, SALARIO
  FROM TB_EMPREGADO
 ORDER BY SALARIO;
--

SELECT ID_EMPREGADO, 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 ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO;
```

	ID_DEPARTAMENTO	NOME	DATA_ADMISSAO	SALARIO
1	NULL	ANTONIO JORGE	2012-02-03 00:00:00.000	NULL
2	NULL	SEVERINO	2012-02-03 00:00:00.000	NULL
3	NULL	JOSE	2012-02-03 00:00:00.000	NULL
4	1	PEDRO	2012-02-03 00:00:00.000	890.00
5	1	ROBERTO	2012-02-03 00:00:00.000	4500.00
6	1	ROBERTO ALEXANDRO	2012-02-03 00:00:00.000	800.00
7	1	JORGE	2012-02-03 00:00:00.000	4500.00
8	1	ARNALDO	2012-02-03 00:00:00.000	890.00
9	1	ROGÉRIO	2012-02-03 00:00:00.000	4500.00
10	1	RONALDO	2012-02-03 00:00:00.000	3300.00

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

```
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO, NOME;
```

	ID_DEPARTAMENTO	NOME	DATA_ADMISSAO	SALARIO
1	NULL	ANTONIO JORGE	2012-02-03 00:00:00.000	NULL
2	NULL	JOSE	2012-02-03 00:00:00.000	NULL
3	NULL	SEVERINO	2012-02-03 00:00:00.000	NULL
4	1	ARNALDO	2012-02-03 00:00:00.000	890.00
5	1	CARLOS FERNANDO	2012-02-03 00:00:00.000	8300.00
6	1	CASSIANO	2012-02-03 00:00:00.000	1200.00
7	1	JOÃO	2012-02-03 00:00:00.000	5000.00
8	1	JORGE	2012-02-03 00:00:00.000	4500.00
9	1	JOSÉ	2012-02-03 00:00:00.000	8300.00
10	1	JOSÉ	2012-02-03 00:00:00.000	3300.00
11	1	PEDRO	2012-02-03 00:00:00.000	890.00
12	1	ROBERTO	2012-02-03 00:00:00.000	4500.00
13	1	ROBERTO ALEXANDRO	2012-02-03 00:00:00.000	800.00

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

```
--EMPREGADOS ORDENADOS PELO ID_DEPARTAMENTO E SALÁRIO
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO, SALARIO;
--EMPREGADOS ORDENADOS PELO ID_DEPARTAMENTO E DATA DE ADMISSÃO
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO, DATA_ADMISSAO;
-- CONTINUA VALENDO O USO DO "ALIAS" OU DA POSIÇÃO DA COLUNA
SELECT ID_DEPARTAMENTO, 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**:

```
--EMPREGADOS ORDENADOS PELO ID_DEPARTAMENTO DECREScente E
SALÁRIO CRESCENTE
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO DESC, SALARIO;
--EMPREGADOS ORDENADOS PELO ID_DEPARTAMENTO CRESCENTE E
SALÁRIO DECREScente
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO, SALARIO DESC;
--EMPREGADOS ORDENADOS PELO ID_DEPARTAMENTO DECREScente E
SALÁRIO DECREScente
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO, SALARIO
FROM TB_EMPREGADO
ORDER BY ID_DEPARTAMENTO DESC, SALARIO DESC;
```

## 3.3.4. Ranking em consulta

É possível realizar um ranking (retornar um número de registros a partir de uma classificação) utilizando as cláusulas **TOP** e **ORDER BY**.

Vamos ver alguns exemplos:

## Consultando dados

Realize a consulta dos cinco clientes que mais gastaram:

```
SELECT TOP 5
    ID_CLIENTE , VLR_TOTAL
FROM TB_PEDIDO
ORDER BY VLR_TOTAL DESC;
```

	ID_CLIENTE	VLR_TOTAL
1	34	22871.25
2	236	19807.36
3	449	18261.96
4	34	17998.70
5	41	16711.33

Apresente os cinco maiores salários:

```
SELECT TOP 5
    NOME, SALARIO
FROM TB_EMPREGADO
ORDER BY SALARIO DESC;
```

	NOME	SALARIO
1	CARLOS FERNANDO	8300.00
2	SEBASTIÃO	8300.00
3	JOSÉ	8300.00
4	ANA	5000.00
5	RICARDO	5000.00

```
-- 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;
```

## 3.3.5. 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 ID_EMPREGADO, NOME, SALARIO  
FROM TB_EMPREGADO  
ORDER BY SALARIO DESC;
```

ID_EMPREGADO	NOME	SALARIO
1	CARLOS FERNANDO	8300.00
2	SEBASTIÃO	8300.00
3	JOSÉ	8300.00
4	JOÃO	5000.00
5	ANA	5000.00
6	RICARDO	5000.00
7	ROBERTO MARIA	4500.00
8	CARLOS ALBERTO	4500.00
9	ROGÉRIO	4500.00
10	JORGE	4500.00

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 ID_EMPREGADO, NOME, SALARIO  
FROM TB_EMPREGADO  
ORDER BY SALARIO DESC;
```

ID_EMPREGADO	NOME	SALARIO
1	CARLOS FERNANDO	8300.00
2	SEBASTIÃO	8300.00
3	JOSÉ	8300.00
4	ANA	5000.00
5	RICARDO	5000.00
6	JOÃO	5000.00
7	CARLOS ALBERTO	4500.00

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

## Consultando dados

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 ID_EMPREGADO, NOME, SALARIO
FROM TB_EMPREGADO
ORDER BY SALARIO DESC;
```

	ID_EMPREGADO	NOME	SALARIO
1	18	CARLOS FERNANDO	8300.00
2	19	SEBASTIÃO	8300.00
3	66	JOSÉ	8300.00
4	43	JOÃO	5000.00
5	26	ANA	5000.00
6	27	RICARDO	5000.00
7	25	ROBERTO MARIA	4500.00
8	7	CARLOS ALBERTO	4500.00
9	53	ROGÉRIO	4500.00
10	51	JORGE	4500.00
11	59	MANOEL ELIAS	4500.00
12	72	ROBERTO	4500.00

Também podemos usar a cláusula **TOP** com percentual. A tabela **TB\_EMPREGADO** possui 61 linhas, então, se pedirmos pra ver 10% das linhas, deverão aparecer 7 linhas devido ao arredondamento.

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

São exibidas as seguintes linhas:

	ID_EMPREGADO	NOME	SALARIO
1	18	CARLOS FERNANDO	8300.00
2	19	SEBASTIÃO	8300.00
3	66	JOSÉ	8300.00
4	43	JOÃO	5000.00
5	26	ANA	5000.00
6	27	RICARDO	5000.00
7	25	ROBERTO MARIA	4500.00

## 3.4. Filtrando consultas

Ao executar qualquer consulta, raramente precisamos apresentar todos os registros contidos na tabela. Para filtrar uma consulta, necessitamos utilizar a cláusula **WHERE**.

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).

### 3.4.1. 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:

- **Empregados com SALÁRIO abaixo de 1000**

```
SELECT ID_EMPREGADO, NOME, ID_CARGO, SALARIO
FROM TB_EMPREGADO
WHERE SALARIO < 1000
ORDER BY SALARIO;
```

ID_EMPREGADO	NOME	ID_CARGO	SALARIO
21	SILVIO	5	500.00
23	JOAQUIM EDUARDO	5	500.00
35	MARIANA	16	500.00
40	JOAQUIM	5	500.00
45	MARIA	5	500.00
55	ARLINDO	5	500.00
57	ANTONIO	5	500.00
2	JOSE	5	600.00
4	PAULO	5	600.00
38	LUIS	14	600.00

## Consultando dados

- Empregados com SALÁRIO acima de 5000

```
SELECT ID_EMPREGADO, NOME, ID_CARGO, SALARIO
FROM TB_EMPREGADO
WHERE SALARIO > 5000
ORDER BY SALARIO;
```

- Empregados com campo ID\_DEPTO menor ou igual a 3

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

- Empregados com campo ID\_CARGO igual a 2

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

- Empregados com campo ID\_CARGO diferente de 2

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

- Fazendo a filtragem com campos do tipo **texto**.

No caso de campo de texto, é necessário colocar entre aspas simples:

- Empregados que possuem o nome ELIANE

```
SELECT *
FROM TB_EMPREGADO
WHERE NOME = 'ELIANE';
```

	ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL
1	8	ELIANE	0	1964-11-22 00:00:00.000	6	5	2012-02-03 00:00:00.000	1200.00	1237.20

- Empregados que possuem o nome iniciado em M

```
SELECT ID_EMPREGADO, NOME, SALARIO
FROM TB_EMPREGADO
WHERE NOME >= 'M'
ORDER BY NOME;
```

	ID_EMPREGADO	NOME	SALARIO
1	31	MANOEL	3300.00
2	59	MANOEL ELIAS	4500.00
3	3	MARCELO	2400.00
4	15	MARCO	2400.00
5	45	MARIA	500.00
6	10	MARIA APARECIDA	1200.00

## 3.5. 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:

Operador	Descrição
<b>AND</b>	Combina duas expressões e exige que sejam verdadeiras para que o registro seja apresentado.
<b>OR</b>	Combina duas expressões e exige que pelo menos uma seja verdadeira para que o registro seja apresentado.
<b>NOT</b>	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:

- Empregados do departamento 2 e que ganhem mais de 5000

```
SELECT *
FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 2 AND SALARIO > 5000;
```

Verifique que somente o empregado de código 19 foi retornado.

	ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL
1	19	SEBASTIÃO	0	1961-10-27 00:00:00.000	2	1	2012-02-03 00:00:00.000	8300.00	1140.90

- Empregados do departamento 2 ou aqueles que ganhem mais de 5000

```
SELECT *
FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 2 OR SALARIO > 5000;
```

	ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_M
1	2	JOSE	6	1962-08-18 00:00:00.000	2	5	2012-02-03 00:00:00.000	600.00	1250.00
2	9	RUDGE	3	1971-05-31 00:00:00.000	2	4	2012-02-03 00:00:00.000	800.00	1255.28
3	18	CARLOS FERNANDO	4	1960-02-19 00:00:00.000	1	1	2012-02-03 00:00:00.000	8300.00	1456.93
4	19	SEBASTIÃO	0	1961-10-27 00:00:00.000	2	1	2012-02-03 00:00:00.000	8300.00	1140.90
5	20	EURICO	0	1960-02-26 00:00:00.000	2	4	2012-02-03 00:00:00.000	800.00	1152.95
6	25	ROBERTO MARIA	0	1967-12-22 00:00:00.000	2	11	2012-02-03 00:00:00.000	4500.00	1377.47
7	28	MARIANO	3	1963-11-27 00:00:00.000	2	9	2012-02-03 00:00:00.000	3330.00	891.17
8	38	LUIS	0	1988-05-31 00:00:00.000	2	14	2012-02-03 00:00:00.000	600.00	303.37
9	40	JOAQUIM	0	1987-11-20 00:00:00.000	2	5	2012-02-03 00:00:00.000	500.00	390.63
10	66	JOSÉ	0	1990-10-11 00:00:00.000	1	1	2012-02-03 00:00:00.000	8300.00	553.64

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, digitariam o seguinte:

```
SELECT *
FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 2 AND ID_DEPARTAMENTO = 5;
```

	ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL	SINDICALIZA

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 ID_DEPARTAMENTO = 2 OR ID_DEPARTAMENTO = 5;
```

	ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_M
1	2	JOSE	6	1962-08-18 00:00:00.000	2	5	2012-02-03 00:00:00.000	600.00	1250.00
2	3	MARCELO	1	1960-04-14 00:00:00.000	5	5	2012-02-03 00:00:00.000	2400.00	1200.00
3	9	RUDGE	3	1971-05-31 00:00:00.000	2	4	2012-02-03 00:00:00.000	800.00	1255.28
4	10	MARIA APARECIDA	0	1964-01-21 00:00:00.000	5	3	2012-02-03 00:00:00.000	1200.00	1181.58
5	13	OSMAR	0	1963-09-05 00:00:00.000	5	2	2012-02-03 00:00:00.000	2400.00	1253.37
6	19	SEBASTIÃO	0	1961-10-27 00:00:00.000	2	1	2012-02-03 00:00:00.000	8300.00	1140.90
7	20	EURICO	0	1960-02-26 00:00:00.000	2	4	2012-02-03 00:00:00.000	800.00	1152.95
8	24	ITAMAR	0	1962-09-05 00:00:00.000	5	3	2012-02-03 00:00:00.000	1200.00	915.83
9	25	ROBERTO MARIA	0	1967-12-22 00:00:00.000	2	11	2012-02-03 00:00:00.000	4500.00	1377.47
10	26	ANA	0	1968-09-03 00:00:00.000	5	10	2012-02-03 00:00:00.000	5000.00	898.30
11	27	RICARDO	0	1963-03-12 00:00:00.000	5	10	2012-02-03 00:00:00.000	5000.00	735.97
12	28	MARIANO	3	1963-11-27 00:00:00.000	2	9	2012-02-03 00:00:00.000	3330.00	891.17
13	38	LUIS	0	1988-05-31 00:00:00.000	2	14	2012-02-03 00:00:00.000	600.00	303.37

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

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

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

- **Empregados com SALARIO abaixo de 3000 ou acima de 5000**

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

Uma opção seria a utilização do operando **NOT**. Porém cuidado ao utilizar esse operando, pois pode gerar lentidão na consulta:

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

## 3.6. Intervalos de valores

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:

- **Empregados na faixa de salário entre 3000 e 5000**

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

--OU

```
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 alfanuméricicos:

```
--Consulta com Texto
SELECT * FROM TB_EMPREGADO
WHERE NOME BETWEEN 'C' AND 'E'

--Consulta com Data
SELECT * FROM TB_EMPREGADO
WHERE DATA_ADMISSAO BETWEEN '2012.1.1' AND '2012.12.31'
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;
```

## 3.7. Pesquisa em campo texto

As pesquisas em campos texto utilizam os mesmos operandos igual (**=**) e diferente (**<>**), porém nesse caso o texto deve ser exato.

No próximo exemplo, é realizada a pesquisa de **EMPREGADO** com nome **JOAO**:

```
SELECT * FROM TB_EMPREGADO
WHERE NOME = 'JOAO'
```

	Results	Messages								
	ID_EMPREGADO	NOME	NUM_DEPEND	DATA_NASCIMENTO	ID_DEPARTAMENTO	ID_CARGO	DATA_ADMISSAO	SALARIO	PREMIO_MENSAL	SINDIC
1	5	JOAO	2	1965-09-07 00:00:00.000	4	5	2012-02-03 00:00:00.000	1200.00	900.00	S
2	12	JOAO	0	1963-03-27 00:00:00.000	4	3	2012-02-03 00:00:00.000	1200.00	1304.05	S

Verifique o resultado quando colocarmos um espaço antes do nome:

```
SELECT * FROM TB_EMPREGADO  
WHERE NOME = ' JOAO'
```

Isso ocorre porque o nome não é o mesmo. Para resolver esse problema, podemos usar o operador **LIKE**. Esse comando é usado para fazer pesquisas em dados do tipo texto (CHAR, VARCHAR, NCHAR e NVARCHAR). É útil quando não sabemos de forma exata o dado que queremos pesquisar.

LIKE	Descrição
% antes do nome '%nome'	Pesquisa o texto que termina com o nome.
% depois do nome 'nome%'	Pesquisa o texto que começa com o nome.
Entre % '%nome%'	Pesquisa o texto que contenha o nome.
Subscrito (_)	Equivale a um único caractere.
Entre colchetes []	Ao colocar os colchetes podemos escolher mais de uma letra para o nome.
NOT LIKE	Pesquisa dos registros que não possuam o nome.

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%';
```

- Nomes que terminam com MARIA

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

- Nomes que terminam com ALBERTO

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

- 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 ELIAS

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

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 **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%' ;
```

## 3.8. Lista de elementos

Operando	Descrição
<b>IN</b>	Pesquisa em uma lista de elementos.
<b>NOT IN</b>	Pesquisa os registros que estão fora da 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 (**ID\_DEPARTAMENTO**) seja 1, 3, 4 ou 7:

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

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

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

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

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

## 3.9. Valores nulos

Operando	Descrição
<b>IS NULL</b>	Pesquisa valores que são nulos.
<b>IS NOT NULL</b>	Pesquisa valores que não são nulos.

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

- Não é um valor zero, ou texto em branco. **NULO** é um valor não inicializado;
- 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 quando fazemos a consulta a seguir:

```
SELECT ID_EMPREGADO, NOME, SALARIO FROM TB_EMPREGADO
WHERE SALARIO < 800
ORDER BY SALARIO;
```

ID_EMPREGADO	NOME	SALARIO
1	SILVIO	500.00
2	JOAQUIM EDUARDO	500.00
3	MARIANA	500.00
4	JOAQUIM	500.00
5	MARIA	500.00
6	ARLINDO	500.00
7	ANTONIO	500.00
8	LUIS	600.00
9	JOSE	600.00
10	PAULO	600.00

Verifique que existem salários nulos:

```
SELECT ID_EMPREGADO, NOME, SALARIO FROM TB_EMPREGADO  
ORDER BY SALARIO;
```

ID_EMPREGADO	NOME	SALARIO
1	ANTONIO JORGE	NULL
2	SEVERINO	NULL
3	JOSE	NULL
4	MARIA	500.00
5	ARLINDO	500.00

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

```
SELECT ID_EMPREGADO, NOME, SALARIO, PREMIO_MENSAL,  
       SALARIO + PREMIO_MENSAL AS RENDA_TOTAL  
FROM TB_EMPREGADO  
WHERE SALARIO IS NULL;
```

ID_EMPREGADO	NOME	SALARIO	PREMIO_MENSAL	RENDATOTAL
1	ANTONIO JORGE	NULL	528.71	NULL
2	SEVERINO	NULL	528.71	NULL
3	JOSE	NULL	NULL	NULL

Para lidar com valores nulos, evitando problemas no resultado final da consulta, pode-se empregar os operadores: **IS NULL** e **IS NOT NULL**.

O exemplo a seguir busca, na tabela **TB\_EMPREGADO**, os registros cujo salário é nulo:

```
SELECT * FROM TB_EMPREGADO  
WHERE SALARIO 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;
```

### 3.10. Funções para tratamento de nulos

Função	Descrição
ISNULL	Esta função permite definir um valor alternativo que será retornado, caso o valor de argumento seja nulo.
COALESCE	Esta função é responsável por retornar o primeiro argumento não nulo em uma lista de argumentos testados.
NULLIF	Retorna um valor nulo caso os dois valores sejam iguais.

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.

- Na próxima consulta é apresentada uma coluna com o valor tratado com ISNULL e outra sem o tratamento

```
SELECT
    ID_EMPREGADO, NOME, SALARIO, PREMIO_MENSAL,
    ISNULL(SALARIO, 0) + ISNULL(PREMIO_MENSAL, 0) AS RENDA_TOTAL,
    SALARIO + PREMIO_MENSAL AS RENDA_SEM_FUNCAO
FROM TB_EMPREGADO
WHERE SALARIO IS NULL;
```

ID_EMPREGADO	NOME	SALARIO	PREMIO_MENSAL	RENDATOTAL	RENDASEMFUNCAO
1	ANTONIO JORGE	NULL	528.71	528.71	NULL
2	SEVERINO	NULL	528.71	528.71	NULL
3	JOSE	NULL	NULL	0.00	NULL

- Exemplo 2

```
SELECT
    ID_EMPREGADO, NOME,
    ISNULL(DATA_NASCIMENTO, '1900.1.1') AS DATA_NASC
FROM TB_EMPREGADO
WHERE DATA_NASCIMENTO IS NULL;
```

ID_EMPREGADO	NOME	DATA_NASC
1	JOSE	1900-01-01 00:00:00.000

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

```
SELECT  
    ID_EMPREGADO,  
    SALARIO,  
    PREMIO_MENSAL,  
    COALESCE (SALARIO,PREMIO_MENSAL, 0) AS COLUNA_COALESCE  
FROM TB_EMPREGADO;
```

## 3.11. Campos data e hora

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;
```

		Results	Messages
		DATA_ATUAL	
1		2019-05-08 17:35:09.277	

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. A função **CAST** tem a finalidade de converter um tipo de dados em outro. Essa função será tratada posteriormente:

```
SELECT ID_EMPREGADO, 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**. O argumento **data** pode ser uma expressão, literal de texto, 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**. O argumento **data** pode ser uma expressão, literal de texto, 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**. O argumento **data** pode ser uma expressão, literal de texto, variável definida pelo usuário ou expressão de coluna.

O exemplo a seguir retorna os empregados admitidos no ano 2010:

```
SELECT * FROM TB_EMPREGADO  
WHERE YEAR(DATA_ADMISSAO) = 2010;
```

Já o exemplo a seguir retorna os empregados admitidos no mês de janeiro de 2015:

```
SELECT * FROM TB_EMPREGADO  
WHERE YEAR(DATA_ADMISSAO) = 2015 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

Valor	Parte retornada	Abreviação
<b>minute</b>	Minuto	mi, n
<b>second</b>	Segundo	ss, s
<b>millisecond</b>	Milissegundo	ms
<b>microsecond</b>	Microsegundo	mcs
<b>nanosecond</b>	Nanossegundo	ns
<b>TZoffset</b>	Diferença de fuso-horário	tz
<b>ISO_WEEK</b>	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 2014. Para isso, são especificadas as partes de ano (**YEAR**) e mês (**MONTH**):

```
SELECT * FROM TB_EMPREGADO
WHERE DATEPART(YEAR, DATA_ADMISSAO) = 2014 AND
      DATEPART(MONTH, DATA_ADMISSAO) = 12;
```

- **DATENAME()**

Esta função retorna o nome literal de uma data que pode ser mês ou dia da semana.

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

```
-- Empregados admitidos em uma sexta-feira
SELECT
    ID_EMPREGADO, 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;
```

ID_EMPREGADO	NOME	DATA_ADMISSAO	DIA_SEMANA	MES
1	OLAVO	2009-11-20 00:00:00.000	Friday	November
2	MARIA APARECIDA	2011-07-29 00:00:00.000	Friday	July
3	JOAO	2019-02-22 00:00:00.000	Friday	February
4	CARLOS FERNANDO	2015-08-14 00:00:00.000	Friday	August
5	SEBASTIÃO	2016-12-23 00:00:00.000	Friday	December
6	MANOEL	2017-02-24 00:00:00.000	Friday	February
7	LUIS	2018-07-13 00:00:00.000	Friday	July
8	JORGE	2016-03-11 00:00:00.000	Friday	March
9	JOSÉ	2015-08-21 00:00:00.000	Friday	August
10	RENAN	2018-06-22 00:00:00.000	Friday	June

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

- **ISDATE()**

Esta função permite validar uma data (DATE, TIME ou DATETIME). O retorno será:

- 1 – Para data válida;
- 0 – Para data inválida.

Verifique os exemplos:

- Teste para uma data válida:

```
SELECT ISDATE('2019.02.28') AS VALIDA_DATA
```

	Results	Messages
	VALIDA_DATA	
1	1	

- Teste para uma data inválida:

```
SELECT ISDATE('2019.02.29') AS VALIDA_DATA
```

	Results	Messages
	VALIDA_DATA	
1	1	

- **Configuração de idioma da sessão**

O padrão de idioma está relacionado ao login do usuário, porém, para termos um padrão, é necessário que todos os usuários estejam no mesmo padrão de linguagem. Uma opção é a configuração pontual da sessão.

Para visualizar a língua configurada na sessão:

```
SELECT @@LANGUAGE AS LINGUA_SESSAO
```

	Results	Messages
	Lingua_Sessao	
1	us_english	

# Consultando dados

```
SELECT DATENAME(MONTH, GETDATE()) AS MÊS
```

		Results	Messages
	MES		
1	May		

Para alterar a língua utilizada na sessão, utilize **SET LANGUAGE**:

```
SET LANGUAGE ITALIAN;
```

```
SELECT DATENAME(MONTH, GETDATE()) AS MES
```

		Results	Messages
	MES		
1	maggio		

Para saber quais as linguagens disponíveis no SQL, é possível realizar uma consulta na view **SYSLANGUAGES**:

```
SELECT * FROM SYSLANGUAGES
```

langid	dateformat	datefirst	upgrade	name	alias	months	shortmonths
1	mdy	7	0	us_english	English	January,February,March,April,May,June,July,August,...	Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,C
2	dmy	1	0	Deutsch	German	Januar,Februar,März,April,Mai,Juni,Juli,August,Septe...	Jan,Feb,Mär,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,D
3	dmy	1	0	Français	French	janvier,février,mars,avril,mai,juin,jUILlet,août,septembr...	janv,févr,mars,avr,mai,juin,juil,août,sept,oct,nov,d
4	ymd	7	0	日本語	Japanese	01,02,03,04,05,06,07,08,09,10,11,12	01,02,03,04,05,06,07,08,09,10,11,12
5	dmy	1	0	Dansk	Danish	januar,februar,marts,april,mai,juni,juli,august,septemb...	jan,feb,mar,apr,mai,jun,jul,aug,sep,okt,nov,dec



### Pontos principais

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

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

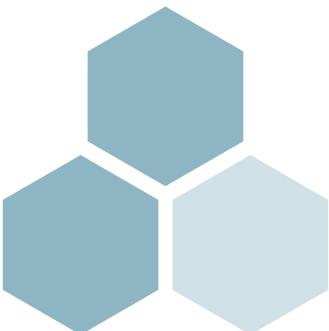




3

# Consultando dados

Teste seus conhecimentos



## 1. Verifique o comando a seguir e responda: Qual afirmação está errada?

```
SELECT ID_EMPREGADO 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 ID_EMPREGADO 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 ID\_CLIENTE do tipo INT)?**

- a) SELECT \* FROM TB\_CLIENTE WHERE ID\_CLIENTE 10
- b) SELECT \* FROM TB\_CLIENTE WHERE ID\_CLIENTE LIKE "%10%"
- c) SELECT \* FROM TB\_CLIENTE WHERE ID\_CLIENTE
- d) SELECT \* FROM TB\_CLIENTE WHERE ISNULL(ID\_CLIENTE, 0) >10
- e) SELECT \* FROM TB\_CLIENTE WHERE ID\_CLIENTE = 10

**5. Ao utilizarmos a cláusula TOP com ORDER BY, o que é possível?**

- 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.





3

# Consultando dados

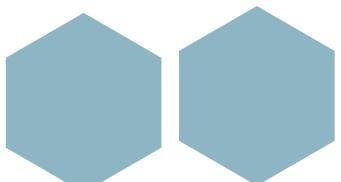


Mãos à obra!

Bruna Morimoto  
397.642-008-78



Editora  
**IMPACTA**



# Laboratório 1

A – Utilizando o banco de dados db\_Ecommerce e listando suas tabelas com base em diferentes critérios

1. Coloque em uso o banco de dados db\_Ecommerce;
2. Liste a tabela **TB\_PRODUTO**, mostrando as colunas **ID\_PRODUTO**, **DESCRICAO**, **PRECO\_CUSTO** e **PRECO\_VENDA** e calculando o lucro unitário (**PRECO\_VENDA** - **PRECO\_CUSTO**);
3. Liste a tabela **TB\_PRODUTO**, mostrando os campos **ID\_PRODUTO** e **DESCRICAO** e calculando o valor total investido no estoque daquele produto (**QTD\_REAL** \* **PRECO\_CUSTO**);
4. Liste a tabela **TB\_ITENSPEDIDO**, mostrando as colunas **ID\_PEDIDO**, **ITEM**, **ID\_PRODUTO**, **PR\_UNITARIO**, **QUANTIDADE** e **DESCONTO** e calculando o valor de cada item (**PR\_UNITARIO** \* **QUANTIDADE** \* (1-**DESCONTO**/100));
5. Liste a tabela **TB\_PRODUTO**, mostrando as colunas **ID\_PRODUTO**, **DESCRICAO**, **PRECO\_CUSTO** e **PRECO\_VENDA** e calculando o lucro estimado em reais (**QTD\_REAL** \* (**PRECO\_VENDA** - **PRECO\_CUSTO**));
6. Liste a tabela **TB\_PRODUTO**, mostrando os campos **ID\_PRODUTO**, **DESCRICAO**, **PRECO\_CUSTO** e **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 db\_Ecommerce e listando suas tabelas com base em novos critérios

1. Coloque em uso o banco de dados db\_Ecommerce;
2. Liste a tabela **TB\_PRODUTO**, criando campo calculado (**QTD\_REAL** - **QTD\_MINIMA**), e filtре 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 os 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 os produtos com **QTD\_REAL** entre 500 e 1000 unidades;
7. Liste os pedidos com valor total entre R\$ 15.000,00 e R\$ 25.000,00;
8. Liste os produtos com quantidade real acima de 5000 e código do tipo igual a 6;
9. Liste os produtos com quantidade real acima de 5000 ou código do tipo igual a 6;
10. Liste os pedidos com valor total inferior a R\$ 100,00 ou acima de R\$ 100.000,00;
11. Liste os produtos com **QTD\_REAL** menor que 500 ou maior que 1000.

## Laboratório 3

### A – Utilizando o banco de dados db\_Ecommerce

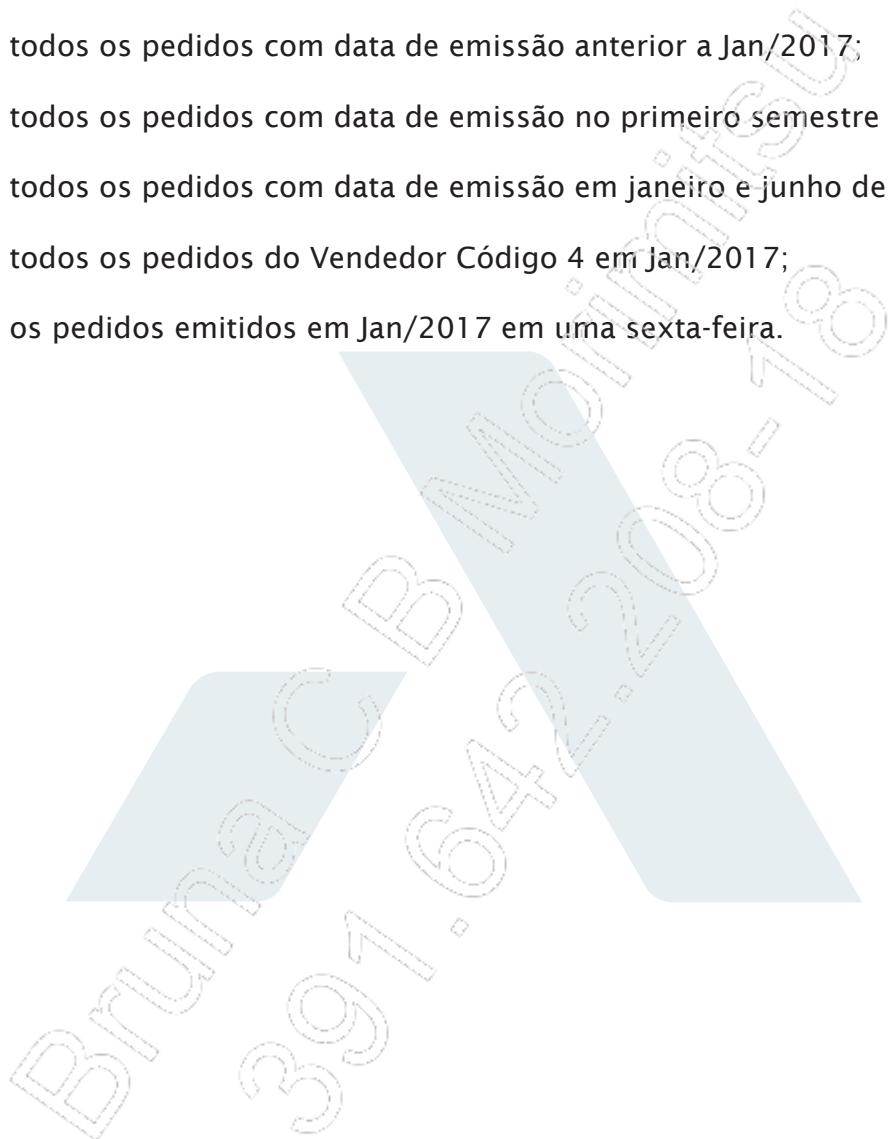
1. Coloque em uso o banco de dados **db\_Ecommerce**;
2. Liste os clientes (**ID\_CLIENTE**) do estado de São Paulo (**SP**). Para essa consulta, utilize a tabela **TB\_ENDERECO**;
3. Liste os clientes dos estados de Minas Gerais e Rio de Janeiro (**MG, RJ**);
4. Liste os clientes dos estados de São Paulo, Minas Gerais e Rio de Janeiro (**SP, MG, RJ**);
5. Realize uma pesquisa que retorne o nome do empregado **MARCO**;
6. Liste todos os clientes que tenham **NOME** começando com **BTB**;
7. Liste todos os clientes que tenham **NOME** terminando com **MMO**;
8. Liste todos os clientes que tenham **NOME** contendo **BCC**;
9. Liste todos os produtos com **DESCRICA0** começando por **CANETA**;
10. Liste todos os produtos com **DESCRICA0** contendo **SPECIAL**;
11. Liste todos os produtos com **DESCRICA0** 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 a letra **A** com terceiro caractere;
14. Liste todas as cidades que possuam o nome **brasil**.



### Laboratório 4

A – Utilizando novamente o banco de dados db\_ecommerce e listando suas tabelas com base em outros critérios

1. Coloque em uso o banco de dados db\_Ecommerce;
2. Liste todos os pedidos com data de emissão anterior a Jan/2017;
3. Liste todos os pedidos com data de emissão no primeiro semestre de 2018;
4. Liste todos os pedidos com data de emissão em janeiro e junho de 2016;
5. Liste todos os pedidos do Vendedor Código 4 em Jan/2017;
6. Liste os pedidos emitidos em Jan/2017 em uma sexta-feira.

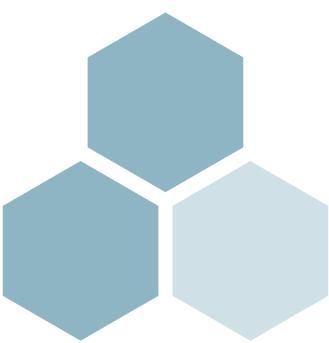




# 4

## Associando tabelas

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



## 4.1. 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.

Associação	Funcionalidade
<b>JOIN</b>	Realiza a associação de duas tabelas. Somente registros com mesmo valor serão retornados.
<b>LEFT JOIN</b>	Realiza a associação de duas tabelas, apresenta os registros com mesmo valor e os da tabela à esquerda, independente da relação.
<b>RIGHT JOIN</b>	Realiza a associação de duas tabelas, apresenta os registros com mesmo valor e os da tabela à direta, independente da relação.
<b>FULL JOIN</b>	Apresenta todos os registros das tabelas.
<b>CROSS JOIN</b>	Realiza um produto cartesiano entre as tabelas.

## 4.2. 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.

Verifique a consulta da tabela de pedidos (**TB\_PEDIDO**):

```
SELECT ID_PEDIDO, DATA_EMISSAO, VLR_TOTAL, ID_EMPREGADO
FROM TB_PEDIDO;
```

	ID_PEDIDO	DATA_EMISSAO	VLR_TOTAL	ID_EMPREGADO
1	1	2016-03-17 01:53:13.497	2.00	3
2	2	2016-12-17 01:53:13.497	30.70	2
3	3	2016-12-17 01:53:13.497	59.76	1
4	4	2016-12-17 01:53:13.497	603.25	2
5	5	2016-12-17 01:53:13.497	72.39	5
6	6	2016-12-17 01:53:13.497	970.57	1
7	7	2016-12-17 01:53:13.497	153.90	3
8	8	2016-12-17 01:53:13.497	874.00	3

Para descobrir qual o nome do vendedor, é necessário fazer uma consulta na tabela de vendedores (**TB\_EMPREGADO**).

```
SELECT ID_EMPREGADO , NOME
FROM TB_EMPREGADO
```

	ID_EMPREGADO	NOME
1	1	OLAVO
2	2	JOSE
3	3	MARCELO
4	4	PAULO
5	5	JOAO

Porém não é viável a consulta isolada das tabelas. Para isso podemos realizar a associação das tabelas usando as cláusulas **WHERE** ou **FROM**. Veja os exemplos a seguir:

```
SELECT TB_PEDIDO.ID_PEDIDO,
       TB_PEDIDO.DATA_EMISSAO,
       TB_PEDIDO.VLR_TOTAL,
       TB_EMPREGADO.NOME
  FROM TB_PEDIDO
 INNER JOIN TB_EMPREGADO
    ON TB_EMPREGADO.ID_EMPREGADO = TB_PEDIDO.ID_EMPREGADO;
```

ID_PEDIDO	DATA_EMISSAO	VLR_TOTAL	NOME
1	2016-03-17 01:53:13.497	2.00	MARCELO
2	2016-12-17 01:53:13.497	30.70	JOSE
3	2016-12-17 01:53:13.497	59.76	OLAVO
4	2016-12-17 01:53:13.497	603.25	JOSE
5	2016-12-17 01:53:13.497	72.39	JOAO
6	2016-12-17 01:53:13.497	970.57	OLAVO
7	2016-12-17 01:53:13.497	153.90	MARCELO
8	2016-12-17 01:53:13.497	874.00	MARCELO
9	2016-12-17 01:53:13.497	3163.50	JOSE
10	2016-12-18 00:00:00.000	229.90	JOSE

Não é recomendado realizar associações com a opção de **WHERE**.

```
SELECT TB_PEDIDO.ID_PEDIDO,
       TB_PEDIDO.DATA_EMISSAO,
       TB_PEDIDO.VLR_TOTAL,
       TB_EMPREGADO.NOME
  FROM TB_PEDIDO , TB_EMPREGADO
 WHERE TB_EMPREGADO.ID_EMPREGADO = TB_PEDIDO.ID_EMPREGADO;
```

A respeito do **INNER JOIN**, é importante considerar as seguintes informações:

- Somente registros que possuírem os mesmos valores nas duas tabelas serão apresentados;
- 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:

- Consulta dos empregados com o nome do departamento

```
SELECT ID_EMPREGADO, NOME, TB_DEPARTAMENTO.DEPARTAMENTO
FROM TB_EMPREGADO
JOIN TB_DEPARTAMENTO ON TB_EMPREGADO.ID_DEPARTAMENTO = TB_
DEPARTAMENTO.ID_DEPARTAMENTO
```

- Consulta de empregados e cargo

```
SELECT TB_EMPREGADO.ID_EMPREGADO, TB_EMPREGADO.NOME, TB_CARGO.
CARGO
FROM TB_EMPREGADO
JOIN TB_CARGO ON TB_CARGO.ID_CARGO = TB_EMPREGADO.ID_CARGO
```

Quando tivermos nomes de tabelas muito extensos, podemos simplificar a escrita, dando apelidos às tabelas. Veja os exemplos:

```
SELECT ID_EMPREGADO, NOME, C.CARGO
FROM TB_EMPREGADO AS E
JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
```

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 (**ID\_DEPARTAMENTO**), e se relaciona com a tabela **TB\_EMPREGADO**, que também possui um campo **ID\_DEPARTAMENTO**, que é a chave estrangeira.

Além da relação entre duas tabelas, é possível adicionar mais tabelas na consulta. Para isso é só continuar informando a tabela após o último **JOIN**.

```
SELECT ID_EMPREGADO, NOME, D.DEPARTAMENTO, C.CARGO
FROM TB_EMPREGADO AS E
JOIN TB_DEPARTAMENTO AS D ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
```

	ID_EMPREGADO	NOME	DEPARTAMENTO	CARGO
1	1	OLAVO	COMPRAS	VENDEDOR(A)
2	2	JOSE	TI	VENDEDOR(A)
3	3	MARCELO	PRODUCAO	VENDEDOR(A)
4	4	PAULO	FINANCIERO	VENDEDOR(A)
5	5	JOAO	COMPRAS	VENDEDOR(A)
6	7	CARLOS ALBERTO	PRESIDENCIA	VENDEDOR(A)
7	8	ELIANE	DIRETORIA	VENDEDOR(A)
8	9	RUDGE	TI	COMPRADOR(A)
9	10	MARIA APARECIDA	PRODUCAO	DIRETOR
	...			

Verifique a consulta a seguir:

```
SELECT ID_EMPREGADO, NOME, D.DEPARTAMENTO , C.CARGO  
FROM TB_DEPARTAMENTO AS D  
JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO  
JOIN TB_EMPREGADO AS E ON E.ID_DEPARTAMENTO = D.ID_  
DEPARTAMENTO
```

Quando executamos um **SELECT**, a primeira cláusula lida é **FROM**, antes de qualquer coisa. Depois é que as outras cláusulas são processadas. No código anterior ocorre o erro:

```
The multi-part identifier "E.ID_CARGO" could not be bound.
```

Isso ocorre porque a tabela **TB\_CARGO** não localizou a tabela **TB\_EMPREGADO**, que é informada na última linha.

A seguir, temos outros exemplos de **JOIN**:

- **Consulta de empregados, cargo, departamento e estado**

```
SELECT E.ID_EMPREGADO, E.NOME, C.CARGO  
FROM TB_EMPREGADO AS E  
JOIN TB_CARGO AS C ON E.ID_CARGO = C.ID_CARGO
```

No próximo exemplo, temos o uso de **JOIN** para consultar três tabelas:

```
SELECT P.DATA_EMISSAO ,  
P.VLR_TOTAL ,  
C.NOME AS CLIENTE ,  
E.NOME AS VENDEDOR  
FROM TB_PEDIDO AS P  
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE  
JOIN TB_EMPREGADO AS E ON E.ID_EMPREGADO = P.ID_EMPREGADO
```

Podemos executar um relacionamento com diversas tabelas. O importante é usar somente as tabelas necessárias para a consulta.

```
SELECT P.DATA_EMISSAO , P.VLR_TOTAL , C.NOME AS CLIENTE ,  
E.NOME AS VENDEDOR , CARGO.CARGO  
FROM TB_PEDIDO AS P  
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE  
JOIN TB_EMPREGADO AS E ON E.ID_EMPREGADO = P.ID_EMPREGADO  
JOIN TB_CARGO AS CARGO ON CARGO.ID_CARGO = E.ID_CARGO
```

Além da comparação com um campo entre as tabelas é possível adicionar a cláusula **AND** para adicionar mais condições na relação. A vantagem nessa opção é a restrição dos registros no momento do **JOIN**.

```
SELECT P.DATA_EMISSAO , P.VLR_TOTAL , C.NOME AS CLIENTE  
FROM TB_PEDIDO AS P  
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE  
AND YEAR(DATA_EMISSAO)= 2016
```

## 4.3. SELF JOIN

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.ID_EMPREGADO,
    E.NOME AS FUNCIONARIO,
    S.NOME AS SUPERVISOR,
    C.CARGO AS CARGO_EMPREGADO,
    CS.CARGO AS CARGO_SUPERVISOR
FROM TB_EMPREGADO AS E
JOIN TB_EMPREGADO AS S ON E.COD_SUPERVISOR = S.ID_EMPREGADO
JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
JOIN TB_CARGO AS CS ON CS.ID_CARGO = S.ID_CARGO
```

ID_EMPREGADO	FUNCIONARIO	SUPERVISOR	CARGO_EMPREGADO	CARGO_SUPERVISOR
1	OLAVO	LÚCIO	VENDEDOR(A)	REPRESENTANTE DA DIREÇÃO
2	JOSE	SEBASTIÃO	VENDEDOR(A)	SUPERVISORA COMERCIAL
3	MARCELO	ANA	VENDEDOR(A)	AUXILIAR LABORATORIO
4	PAULO	CARLOS ALBERTO	VENDEDOR(A)	VENDEDOR(A)
5	JOAO	LÚCIO	VENDEDOR(A)	REPRESENTANTE DA DIREÇÃO
6	ELIANE	ARLINDO	VENDEDOR(A)	VENDEDOR(A)

## 4.4. OUTER JOIN

A cláusula **INNER JOIN**, vista anteriormente, tem como característica retornar apenas as linhas em que o campo de relacionamento existe em ambas as tabelas. Se o conteúdo do campo chave de relacionamento existe em uma tabela, mas não na outra, essa linha não será retornada pelo **SELECT**. Vejamos um exemplo de **INNER JOIN**:

A consulta a seguir realiza uma consulta de tabela de empregado e retorna 61 registros:

```
SELECT * FROM TB_EMPREGADO
```

Ao realizar um **JOIN** com a tabela de cargo são retornados apenas 58 registros.

```
SELECT *
FROM TB_EMPREGADO AS E
JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
```

Isso ocorre porque existem registros na tabela de empregado que não têm relação com a de cargo. Nesse caso o problema está em registros que possuem valor nulo.

```
SELECT * FROM TB_EMPREGADO WHERE ID_CARGO IS NULL
```

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.

Vamos realizar a consulta anterior utilizando a cláusula **LEFT**, ou seja, além dos registros que são comuns, também serão mostrados todos os da tabela de empregados.

```
SELECT *
FROM TB_EMPREGADO AS E
LEFT JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
```

O **SELECT** a seguir verifica, na tabela **TB\_EMPREGADO**, os empregados que não possuem um código de cargo:

```
SELECT ID_EMPREGADO, NOME, E.ID_CARGO
FROM TB_EMPREGADO AS E
LEFT JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
WHERE E.ID_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 **ID\_CARGO** 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\_CARGO**:

```
SELECT ID_EMPREGADO, NOME, E.ID_CARGO , C.CARGO
FROM TB_EMPREGADO AS E
RIGHT JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
WHERE E.ID_CARGO 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**:

```
SELECT ID_EMPREGADO, NOME, E.ID_CARGO , C.CARGO
FROM TB_EMPREGADO AS E
FULL JOIN TB_CARGO AS C ON C.ID_CARGO = E.ID_CARGO
WHERE E.ID_CARGO IS NULL
```

## 4.5. 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**:

```
SELECT ID_EMPREGADO, NOME, E.ID_CARGO , C.CARGO
FROM TB_EMPREGADO AS E
CROSS JOIN TB_CARGO AS C
```

**A CROSS JOIN** deve ser utilizada apenas quando for realmente necessário um produto cartesiano, já que o resultado gerado pode ser muito grande.



### Pontos principais

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

- A associação de tabelas pode ser realizada, por exemplo, para converter em informação os dados encontrados em duas ou mais tabelas. As tabelas podem ser combinadas por meio de uma condição ou um grupo de condições de junção;
- É importante ressaltar que as tabelas devem ser associadas em pares, embora seja possível utilizar um único comando para combinar várias tabelas. Um procedimento muito comum é a associação da chave primária da primeira tabela com a chave estrangeira da segunda tabela;
- **JOIN** é uma cláusula que permite a associação entre várias tabelas, com base na relação existente entre elas. Por meio dessa cláusula, os dados de uma tabela são utilizados para selecionar dados pertencentes à outra tabela;
- **SELF JOIN** realiza um relacionamento com a própria tabela;
- Há diversos tipos de **JOIN**: **INNER JOIN**, **OUTER JOIN** (**LEFT JOIN**, **RIGHT JOIN** e **FULL JOIN**) e **CROSS JOIN**.

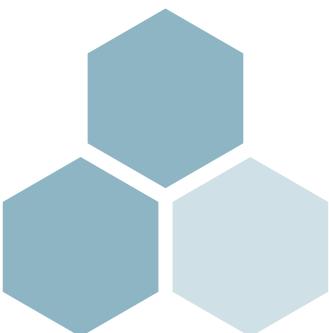




4

## Associando tabelas

Teste seus conhecimentos





## 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 a seguir e responda: Qual afirmação é a correta?

```
SELECT  
E.ID_EMPREGADO,  
E.NOME,  
C.CARGO  
FROM TB_EMPREGADO AS E  
JOIN TB_CARGO AS C ON E.ID_CARGO = E.ID_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 a seguir e responda: Qual afirmação é a correta?**

```

SELECT
    I.ID_PEDIDO, I.ITEM, I.ID_PRODUTO, PR.DESCRICAO,
    I.QUANTIDADE, I.PR_UNITARIO, T.TIPO, U.UNIDADE, CR.COR,
    PE.DATA_EMISSAO
FROM TB_ITENSPEDIDO I
    JOIN TB_PRODUTO      AS PR ON I.ID_PRODUTO = PR.ID_PRODUTO
    JOIN TB_COR           AS CR ON I.ID_COR     = CR.ID_
COR
    JOIN TB_TIPOPRODUTO AS T  ON PR.ID TIPO   = T.ID TIPO
    JOIN TB_UNIDADE       AS U  ON PR.ID UNIDADE = U.ID UNIDADE
    JOIN TB_PEDIDO        AS PE ON I.ID_PEDIDO = PE.ID_PEDIDO
WHERE PE.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31';

```

- 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. O que 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 os da outra.
- c) É um recurso disponível somente da versão SQL 2019 em diante.
- d) Devemos utilizar sempre e filtrarmos a consulta com WHERE.
- e) Não existe este tipo de JOIN.



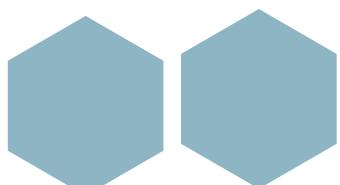
4

# Associando tabelas



Mãos à obra!

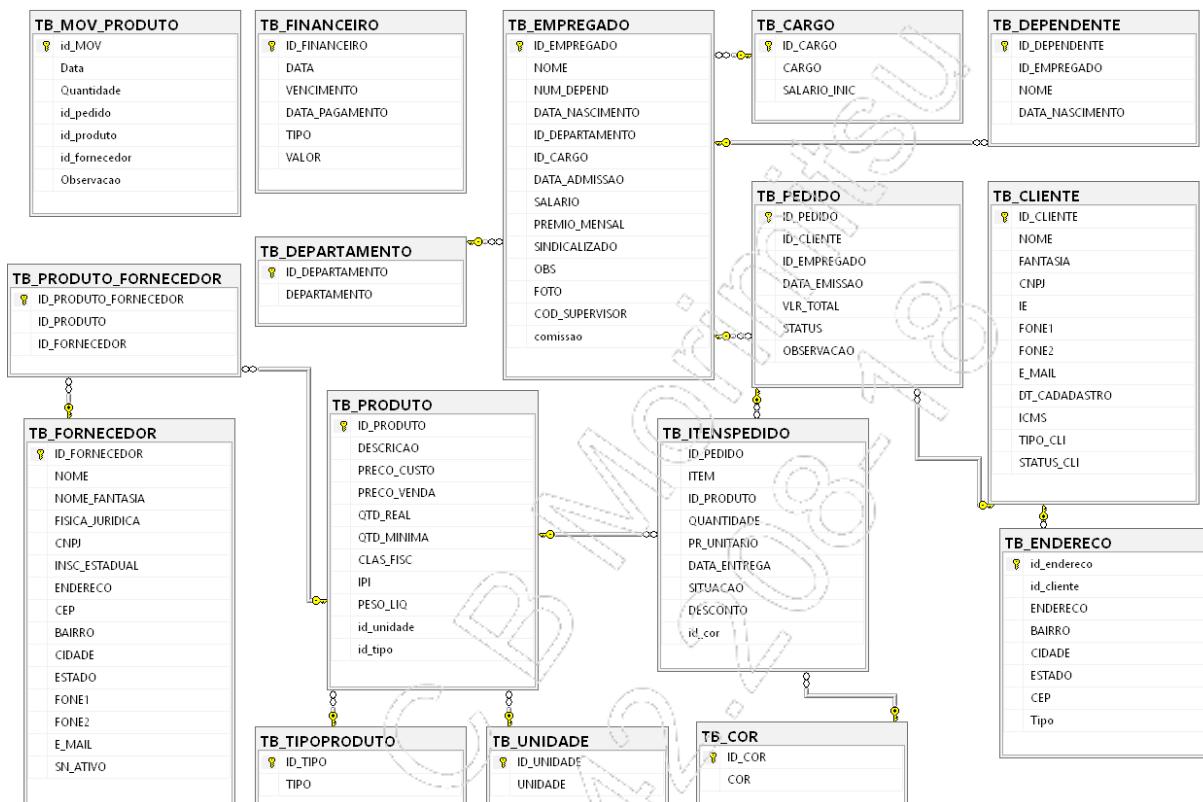
Bruna  
397  
C  
2009  
78  
Morimoto



## 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 **db\_Ecommerce**;
2. Liste os campos **ID\_PEDIDO**, **DATA\_EMISSAO** e **VLR\_TOTAL** da tabela **TB\_PEDIDO**, seguidos de **NOME** do vendedor da tabela **TB\_EMPREGADO**;
3. Liste os campos **ID\_PEDIDO**, **DATA\_EMISSAO** e **VLR\_TOTAL** da tabela **TB\_PEDIDO**, seguidos de **NOME** do cliente da tabela **TB\_CLIENTE**;
4. Liste os pedidos com o nome do vendedor e o nome do cliente;
5. Liste os pedidos com o nome do vendedor e o cargo do vendedor;
6. Liste os itens de pedido (**TB\_ITENS\_PEDIDO**) com o nome do produto (**TB\_PRODUTO**.**DESCRICAO**);
7. Liste os itens pedidos, o nome do produto e a cor (**TB\_COR**);
8. Liste os campos **ID\_PRODUTO** e **DESCRICAO** da tabela **TB\_PRODUTO**, seguidos da descrição do tipo de produto (**TB\_TIPO\_PRODUTO.TIPO**);

9. Liste os campos **ID\_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**);

10. Liste os campos **NUM\_PEDIDO**, **NUM\_ITEM**, **ID\_PRODUTO**, **QUANTIDADE** e **PR\_UNITARIO** da tabela **TB\_ITENSPEEDIDO**, 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**);

11. Liste os campos **NUM\_PEDIDO**, **NUM\_ITEM**, **ID\_PRODUTO**, **QUANTIDADE** e **PR\_UNITARIO** da tabela **TB\_ITENSPEEDIDO**, e os campos **ID\_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**);

12. Liste todos os pedidos (**TB\_PEDIDO**) do vendedor **MARCELO** em Jan/2017;



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

13. Liste os nomes dos clientes (**TB\_CLIENTE.NOME**) que efetuaram compras em janeiro de 2018;

14. Liste os nomes de produtos (**TB\_PRODUTO.DESCRIAO**) que foram vendidos em janeiro de 2019;

15. Liste **NUM\_PEDIDO**, **VLR\_TOTAL** (**PEDIDOS**) e **NOME** (**TB\_CLIENTE**). Mostre apenas pedidos de janeiro de 2017 do estado de SP;

16. Liste **NUM\_PEDIDO**, **QUANTIDADE** vendida e **PR\_UNITARIO** (**TB\_ITENSPEEDIDO**), **DESCRICAO** (**TB\_PRODUTO**), **NOME** do vendedor que vendeu cada item de pedido (**TB\_VENDEDOR**);

17. Liste o número do pedido e o valor dos pedidos de cor verde de 2018;

18. Liste os clientes de SP que realizaram compras em fevereiro de 2017;

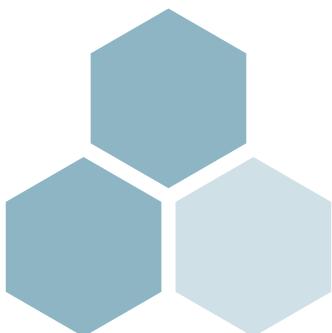
19. Liste os produtos fornecidos pelos fornecedores que iniciam com a letra A.



# 5

## Subconsulta

- ◆ Características;
- ◆ Subconsulta com IN e NOT IN;
- ◆ Operadores;
- ◆ Subconsulta correlacionada;
- ◆ Subconsultas e associações.



### 5.1. Introdução

Uma consulta aninhada em uma instrução **SELECT**, **INSERT**, **DELETE** ou **UPDATE** é chamada de subconsulta (SUBCONSULTAS). Isso ocorre quando usamos um **SELECT** dentro de um **SELECT**, **INSERT**, **UPDATE** ou **DELETE**. Também é comum chamar uma subconsulta de consulta interna. Já a instrução em que está inserida a subconsulta pode ser chamada de consulta externa.

O limite máximo de aninhamento de uma subconsulta 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.

### 5.2. Características

Vejamos, a seguir, características de uma subconsulta:

- As subconsultas podem ser escalares (retornam apenas uma linha) ou tabulares (retornam linhas e colunas);
- Em consultas escalares, somente é possível obter uma coluna por subconsulta;
- Uma subconsulta, que pode ser incluída dentro de outra subconsulta, deve estar entre parênteses, o que a diferenciará da consulta principal;
- Em instruções **SELECT**, **UPDATE**, **INSERT** e **DELETE**, uma subconsulta é utilizada nos mesmos locais em que poderiam ser utilizadas expressões;
- Pelo fato de podermos trabalhar com consultas estruturadas, as subconsultas 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 subconsulta;
- Alguns tipos de dados não podem ser utilizados na lista de seleção de uma subconsulta. São eles: **nvarchar(max)**, **varchar(max)** e **varbinary(max)**;
- Um único valor será retornado ao utilizarmos o sinal de igualdade (=) no início da subconsulta;
- As palavras-chave **ALL**, **ANY** e **SOME** podem ser utilizadas para modificar operadores de comparação que introduzem uma subconsulta. 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 ( $=$ ,  $<$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\neq$ ,  $\exists$  ou  $\leq$ ) para introduzir uma subconsulta, 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 subconsultas 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 T-SQL com subconsultas e associações;
- O nome de coluna que, ocasionalmente, estiver presente na cláusula **WHERE** de uma consulta externa deve ser associável com a coluna da lista de seleção da subconsulta;
- A qualificação dos nomes de colunas de uma instrução é feita pela tabela referenciada na cláusula **FROM**;
- Subconsultas que incluem **GROUP BY** não aceitam a utilização de **DISTINCT**;
- Uma view não pode ser atualizada caso ela tenha sido criada com uma subconsulta;
- Uma subconsulta 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**;
- Subconsultas 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 subconsultas eliminam a necessidade de utilização das cláusulas **JOIN** e **UNION** de maior complexidade;
- Uma instrução que possui uma subconsulta não apresenta muitas diferenças de performance em relação a uma versão semanticamente semelhante que não possui a subconsulta. 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 consulta externa, caso essa tabela apareça apenas em uma subconsulta e não na consulta externa.

A seguir, temos os formatos normalmente apresentados pelas instruções que possuem uma subconsulta:

- Em substituição de campos após o SELECT:

```
WHERE expressao [NOT] IN (subconsulta);  
WHERE expressao operador_comparacao [ANY | ALL] (subconsulta);  
WHERE [NOT] EXISTS (subconsulta).
```

Vejamos alguns exemplos:

- Consulta com subconsulta na expressão de campos do SELECT:

```
SELECT P.DATA_EMISSAO, P.ID_PEDIDO, P.VLR_TOTAL,  
(SELECT SUM(VLR_TOTAL) FROM TB_PEDIDO  
WHERE YEAR(DATA_EMISSAO) = 2016 ) AS TOTAL_VENDAS_2016  
FROM TB_PEDIDO AS P  
WHERE YEAR(DATA_EMISSAO) = 2016
```

	DATA_EMISSAO	ID_PEDIDO	VLR_TOTAL	TOTAL_VENDAS_2016
1	2016-03-17 01:53:13.497	1	2.00	4171608.35
2	2016-12-17 01:53:13.497	2	30.70	4171608.35
3	2016-12-17 01:53:13.497	3	59.76	4171608.35
4	2016-12-17 01:53:13.497	4	603.25	4171608.35
5	2016-12-17 01:53:13.497	5	72.39	4171608.35
6	2016-12-17 01:53:13.497	6	970.57	4171608.35
7	2016-12-17 01:53:13.497	7	153.90	4171608.35
8	2016-12-17 01:53:13.497	8	874.00	4171608.35
9	2016-12-17 01:53:13.497	9	3163.50	4171608.35

- Subconsulta na cláusula FROM:

```
SELECT *  
FROM  
(SELECT ID_PEDIDO, DATA_EMISSAO, VLR_TOTAL  
FROM TB_PEDIDO WHERE YEAR(DATA_EMISSAO) = 2016) AS QA
```

	ID_PEDIDO	DATA_EMISSAO	VLR_TOTAL
1	1	2016-03-17 01:53:13.497	2.00
2	2	2016-12-17 01:53:13.497	30.70
3	3	2016-12-17 01:53:13.497	59.76
4	4	2016-12-17 01:53:13.497	603.25
5	5	2016-12-17 01:53:13.497	72.39
6	6	2016-12-17 01:53:13.497	970.57
7	7	2016-12-17 01:53:13.497	153.90
8	8	2016-12-17 01:53:13.497	874.00
9	9	2016-12-17 01:53:13.497	3163.50

Vejamos um exemplo de subconsulta que verifica a existência de clientes que compraram no mês de janeiro de 2016:

```

SELECT * FROM TB_CLIENTE
WHERE EXISTS (SELECT * FROM TB_PEDIDO
    WHERE ID_CLIENTE = TB_CLIENTE.ID_CLIENTE AND
        DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31');
-- OU
SELECT * FROM TB_CLIENTE
WHERE ID_CLIENTE IN (SELECT ID_CLIENTE FROM TB_PEDIDO
    WHERE DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31');

```

	ID_CLIENTE	NOME	FANTASIA	CNPJ	IE	FONE1	FONE2	E_MAIL
1	19	XJKPOXBKHCUMOMUJHEJGHTCMMVN	QWIPMEFXKVIBTO	67227658833428	4671341525144465206	20201717328	43433428222	RBQFCIPDPX
2	24	NIKMNRCUHQUYUIGHFMVRTCNMYLVNEE	HCSOTDXXFUECYP	51178513163258	4147236858841264367	44384156341	44614713711	GNAFOEDIF
3	31	FNMVHMKMSFITGCFJEMLDUPBGDPCIFD	MFMUGXSIYKEKUVG	11418117756636	4370577147330424572	75774346215	50206245542	HRHOBVHPX
4	32	NOGIQEKLUSLPLBFMBSPECGRSAXTCOE	KHPIHRECERUMMAF	61464624443683	2012639335151507134	32466760226	74511785266	UUHSMVNRE
5	35	BPKTSOWDVBSICYNAKURIVEXJSXPIGU	UFUQNDSHXWTPDTN	20470887111527	3137716822475528703	80223524735	54713703635	PUGSINOEXJ
6	47	KVJLBXXWDXNEPNDXFINTIAOPVYNOR	SHCVEOIQYRDXLGW	43174365285177	717375574636075663	2550441188	63777353011	QUAQOWUSC
7	51	PDWJLCMQXOWEWLKVTKUHVNTDXGMO	PWDMCMBDLTEWBWJ	04176522892141	0571673151628632146	31732675505	47221663833	CHSSXIGGU
8	73	BKXIEMENBHHRANPEPOOJJDOCNOULCFI	TISHGXBJKGUDWA	51564431221211	2627656222250364262	80222953374	45343312456	VWMPFWNOX

No próximo exemplo, é verificada a existência de clientes que não compraram em janeiro de 2016:

```

SELECT * FROM TB_CLIENTE
WHERE NOT EXISTS (SELECT * FROM TB_PEDIDO
    WHERE ID_CLIENTE = TB_CLIENTE.ID_CLIENTE AND
        DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31');
-- OU
SELECT * FROM TB_CLIENTE
WHERE ID_CLIENTE NOT IN (SELECT ID_CLIENTE FROM TB_PEDIDO
    WHERE DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31');

```

	ID_CLIENTE	NOME	FANTASIA	CNPJ	IE	FONE1	FONE2	E_MAIL
1	3	SQSKRGYHTBLNWIAFRAEKTDQPVWFOLE	UQDDESKOGMNPVHJ	75441168107640	2366546204042665373	23266266455	56212106888	XIMIKNUJB
2	4	QCQQBNPUKNHDSFBMEKESJNMJMJDPP	CMEKPTSEFHGUHG	13415416137526	1232174750387123066	12377276844	31630275544	OXIBXDMV
3	5	TTOMDCPSXCWRXSQWWNOVURRENQDFKL	LKWDRITERXHQKMT	47554254311717	3736373517726751140	46116241685	53443550354	DORJNTQM
4	6	ANXTUDDIVPWQUVINKNFBVLEOSUODX	CXNEJDNDRNQYQHTN	15532554268767	0342873281571252357	18867421776	61751768457	EKTGENK
5	7	TCSGWJISYSISLFFELSMLLYSBMPQRNK	OSGDGNKHGJYFUPCJ	50765211004317	4144613835402321551	34271555417	49867565135	EOWSGJJ
6	8	QXVTRVGAOBVXCAAQIGMMHNFFIFVJH	RWFYBWCWINNNHM	63337228707746	1450345882181175375	63367513277	41554102380	KXDJFQVC
7	11	OVRIUUMSLRNTIJVGFCVLHFJQTKRSOKD	OCLKEQHOCUUYRLN	47521518136144	5334443718211822357	16243601251	25433201182	WFPVGRYI
8	12	RDVNRMHFEIRJCATCAQDFLBJHHMUJO	FNHUFRIWVCGIIIP	88112414378743	8173731723766842245	86343273643	74184811870	UKVRFLRL

## 5.3. Subconsulta com IN e NOT IN

Uma subconsulta 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 consulta externa.

Os exemplos adiante demonstram subconsultas introduzidas com **IN** e **NOT IN**:

- **Exemplo 1**

```
-- Lista de empregados cujo cargo tenha salário inicial
-- inferior a 5000
SELECT * FROM TB_EMPREGADO
WHERE ID_CARGO IN (SELECT ID_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 ID_DEPARTAMENTO NOT IN
      (SELECT DISTINCT ID_DEPARTAMENTO FROM TB_EMPREGADO
       WHERE ID_DEPARTAMENTO IS NOT NULL)
-- O mesmo que
SELECT
    E.ID_CARGO, E.NOME, E.ID_DEPARTAMENTO, E.ID_CARGO, D.ID_
DEPARTAMENTO, D.DEPARTAMENTO
  FROM TB_EMPREGADO E RIGHT JOIN TB_DEPARTAMENTO D ON E.ID_
DEPARTAMENTO = D.ID_DEPARTAMENTO
 WHERE E.ID_DEPARTAMENTO IS NULL
```

Results		Messages	
	ID_DEPARTAMENTO	DEPARTAMENTO	
1	10	TREINAMENTO	
2	13	CONTROLADORIA	

	ID_CARGO	NOME	ID_DEPARTAMENTO	ID_CARGO	ID_DEPARTAMENTO	DEPARTAMENTO
1	NULL	NULL	NULL	NULL	10	TREINAMENTO
2	NULL	NULL	NULL	NULL	13	CONTROLADORIA

- **Exemplo 3**

```
-- LISTA DE CARGOS EM QUE NÃO EXISTE NENHUM FUNCIONÁRIO CADASTRADO
SELECT * FROM TB_CARGO
WHERE ID_CARGO NOT IN
(SELECT DISTINCT ID_CARGO FROM TB_EMPREGADO
 WHERE ID_CARGO IS NOT NULL)
-- O MESMO QUE
SELECT
    C.ID_CARGO, C.CARGO
FROM TB_EMPREGADO E RIGHT JOIN TB_CARGO C ON E.ID_CARGO =
C.ID_CARGO
WHERE E.ID_CARGO IS NULL
```

## 5.4. Operadores

Ao utilizar uma subconsulta que possui um operador =, >, <, <>, entre outros, é necessário que o retorno seja um valor único.

Vejamos exemplos de como utilizar o sinal de igualdade (=) para inserir subconsultas:

- **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;
```

## 5.5. Subconsulta correlacionada

Quando uma subconsulta possui referência a uma ou mais colunas da consulta externa, ela é chamada de subconsulta correlacionada. É uma subconsulta repetitiva, pois é executada uma vez para cada linha da consulta externa. Assim, os valores da subconsulta correlacionada depende da consulta externa, o que significa que, para construir uma subconsulta desse tipo, será necessário criar tanto a consulta interna como a externa.

Também é possível que uma subconsulta correlacionada inclua, 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 consulta externa sejam referenciadas como argumento de uma função desse tipo. Então, será feita a avaliação dessa função de acordo com a subconsulta para cada linha da consulta externa.

Veja o exemplo a seguir, que mostra o salário inicial da tabela **TB\_CARGO**.

```
SELECT ID_EMPREGADO , NOME , SALARIO,
(SELECT SALARIO_INIC FROM TB_CARGO WHERE ID_CARGO = E.ID_
CARGO) AS SALARIO_INICIAL
FROM TB_EMPREGADO AS E
```

### 5.5.1. Correlação com EXISTS

Subconsultas 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 subconsulta, 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 subconsulta, 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 2016:

```
SELECT * FROM TB_CLIENTE
WHERE NOT EXISTS (SELECT * FROM TB_PEDIDO
WHERE ID_CLIENTE = TB_CLIENTE.ID_CLIENTE AND
DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31');
```

## 5.6. Subconsultas e associações

Ao comparar subconsultas 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 subconsultas, 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 subconsultas. Isso porque a visualização do aninhamento e da organização da consulta é mais simples em uma subconsulta, 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 subconsulta.

O código a seguir utiliza **JOIN** para calcular o total vendido por cada vendedor no período de janeiro de 2016 e a porcentagem de vendas em relação ao total de vendas realizadas no mesmo mês:

```
SELECT P.ID_EMPREGADO, 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 '2016.1.1' AND '2016.1.31')
AS PORCENTAGEM
FROM TB_PEDIDO AS P
JOIN TB_EMPREGADO AS V ON P.ID_EMPREGADO = V.ID_EMPREGADO
WHERE P.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31'
GROUP BY P.ID_EMPREGADO, V.NOME;
```

ID_EMPREGADO	NOME	TOT_VENDIDO	PORCENTAGEM
1	OLAVO	39623.76	12.911002
2	JOSE	55449.46	18.067647
3	MARCELO	34652.20	11.291069
4	PAULO	42232.22	13.760943
5	JOAO	37484.06	12.213802
7	CARLOS ALBERTO	23383.82	7.619381
8	ELIANE	32060.65	10.446639
69	RENAN	42013.00	13.689512

Já o código a seguir utiliza subconsultas para calcular, para cada departamento, o total de salários dos funcionários sindicalizados e o total de salários dos não sindicalizados:

```
SELECT ID_DEPARTAMENTO,
       (SELECT SUM(E.SALARIO) FROM TB_EMPREGADO E
        WHERE E.SINDICALIZADO = 'S' AND
              E.ID_DEPARTAMENTO = TB_EMPREGADO.ID_DEPARTAMENTO) AS
       TOT_SALARIO_SIND,
       (SELECT SUM(E.SALARIO) FROM TB_EMPREGADO E
        WHERE E.SINDICALIZADO = 'N' AND
              E.ID_DEPARTAMENTO = TB_EMPREGADO.ID_DEPARTAMENTO) AS
       TOT_SALARIO_NAO_SIND
  FROM TB_EMPREGADO
 GROUP BY ID_DEPARTAMENTO;
```

	ID_DEPARTAMENTO	TOT_SALARIO_SIND	TOT_SALARIO_NAO_SIND
1	NULL	NULL	NULL
2	1	36290.00	8300.00
3	2	5030.00	14400.00
4	3	5500.00	5700.00
5	4	18190.00	NULL
6	5	6500.00	12090.00
7	6	5800.00	NULL
8	7	1200.00	NULL
9	8	3900.00	1200.00

## Pontos principais

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

- Uma consulta aninhada em uma instrução **SELECT**, **INSERT**, **DELETE** ou **UPDATE** é denominada subconsulta (SUBCONSULTAS). As subconsultas são também referidas como consultas internas. Já a instrução em que está inserida a subconsulta pode ser chamada de consulta externa;
- Vejamos algumas das diversas características das subconsultas: podem ser escalares (retornam apenas uma linha) ou tabulares (retornam linhas e colunas). Elas, que podem ser incluídas dentro de outras subconsultas, devem estar entre parênteses, o que as diferenciará da consulta principal;
- Uma subconsulta 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 consulta externa;
- O sinal de igualdade (=) pode ser utilizado para inserir subconsultas;
- Quando uma subconsulta possui referência a uma ou mais colunas da consulta externa, ela é chamada de subconsulta correlacionada. Trata-se de uma subconsulta repetitiva, pois é executada uma vez para cada linha da consulta externa. Desta forma, os valores das subconsultas correlacionadas dependem da consulta externa, o que significa que, para construir uma subconsulta desse tipo, será necessário criar tanto a consulta interna como a externa;
- Ao comparar subconsultas 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 consulta é mais simples em uma subconsulta, enquanto que, em uma consulta com diversas associações, a visualização pode ser complicada.

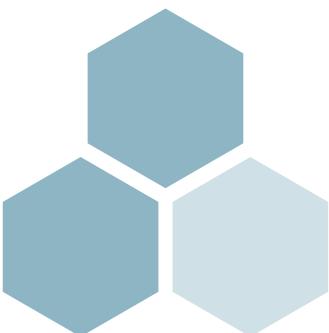




5

## Subconsulta

Teste seus conhecimentos



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

- a) A utilização da cláusula ORDER BY só é possível caso a cláusula TOP seja especificada.
- b) Subconsultas 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 subconsulta.
- d) Em instruções SELECT, UPDATE, INSERT e DELETE, uma subconsulta 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 subconsulta.

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

```
SELECT * FROM TB_CLIENTE  
WHERE ID_CLIENTE IN (SELECT ID_CLIENTE FROM TB_PEDIDO  
WHERE DATA_EMISSAO BETWEEN '2017.1.1' AND '2017.1.31');
```

- a) Apresenta os clientes que compraram em janeiro de 2017.
- b) Apresenta os clientes que não compraram em janeiro de 2017.
- 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 2017.

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

```
SELECT * FROM TB_EMPREGADO  
WHERE ID_CARGO = (SELECT ID_CARGO FROM TB_CARGO  
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 subconsultas correlacionadas?**

- a) Antes de EXISTS não deve haver nome de coluna, constante ou expressão.
- b) Quando EXISTS introduz uma subconsulta, sua lista de seleção será, normalmente, um asterisco.
- c) Subconsultas correlacionadas não podem incluir funções definidas pelo usuário.
- d) A cláusula EXISTS não retorna dados, mas apenas TRUE ou FALSE.
- e) A subconsulta será executada para cada linha da consulta externa.

**5. Verifique o comando a seguir:**

```
SELECT *
FROM
(SELECT ID_PEDIDO, DATA_EMISSAO, VLR_TOTAL
FROM TB_PEDIDO WHERE YEAR(DATA_EMISSAO) = 2016) AS QA
```

**Assinale a afirmação correta:**

- a) Não devemos utilizar subconsulta na cláusula FROM.
- b) O correto é usar uma tabela ou view.
- c) No momento da execução, não será retornado nenhum valor.
- d) A subconsulta retorna os pedidos de 2016.
- e) Gera um erro de sintaxe.





5

# Subconsultas



Mãos à obra!

Bruna Morimoto  
397-6422-0087-78



Editora  
**IMPACTA**





# Laboratório 1

## A – Utilizando subconsultas

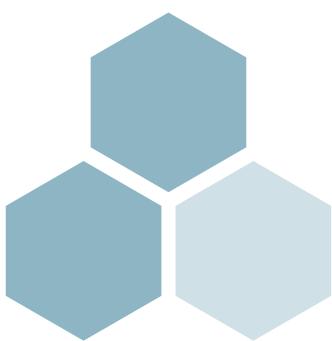
Realize as consultas adiante:

1. Coloque o banco **db\_Ecommerce** em uso;
2. Selecione os clientes que não compraram em Março de 2019;
3. Selecione os produtos que nunca foram vendidos;
4. Apresente os cargos que não possuem funcionários cadastrados;
5. Apresente os produtos vendidos em Abril de 2018 que não são da cor PRATA;
6. Apresente os produtos que foram vendidos em abril de 2017, menos CHAVE DESMONTADO;
7. Apresente os vendedores que não venderam em Dezembro de 2019;
8. Apresente os clientes que compraram, em Fevereiro de 2018, produtos da cor AZUL;
9. Apresente os empregados que não possuem dependentes e cargo que não seja MOTORISTA;
10. Apresente os empregados de cargo VENDEDOR(A) que não realizaram vendas em abril de 2018.

# 6

## Agrupando dados

- Funções de agregação;
- GROUP BY.



## 6.1. Introdução

Neste capítulo, você aprenderá como a cláusula **GROUP BY** pode ser utilizada para agrupar vários dados, tornando mais prática sua summarização. Também verá como utilizar funções de agregação para summarizar dados e como a cláusula **GROUP BY** pode ser usada com a cláusula **HAVING** e com os operadores **ALL**, **WITH ROLLUP** e **CUBE**.

## 6.2. 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.

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:

```
-- 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
-- ID_DEPARTAMENTO = 2
SELECT AVG(SALARIO) AS SALARIO_MEDIO FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 2;
```

Results	
	Messages
	SALARIO_MEDIO
1	2360.689655
	SALARIO_MEDIO
1	2428.750000

- **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
-- ID_DEPARTAMENTO = 2
SELECT COUNT(ID_DEPARTAMENTO) AS QTD_EMPREGADOS FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 2;
```

Results	
Messages	
QTD_EMPREGADOS	
1	61
QTD_EMPREGADOS	
1	8

! Se colocarmos o nome de um campo como argumento da função **COUNT**, não serão contados os registros em que o conteúdo desse campo seja **NUL**L.

- **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 = 7  
SELECT MIN(SALARIO) AS MENOR_SALARIO FROM TB_EMPREGADO  
WHERE ID_DEPARTAMENTO = 7
```

Results	
	Messages
1	MENOR_SALARIO
1	500.00
1	MENOR_SALARIO
1	1200.00

- **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 = 7  
SELECT MAX(SALARIO) AS MAIOR_SALARIO FROM TB_EMPREGADO  
WHERE ID_DEPARTAMENTO = 7
```

Results	
	Messages
1	MAIOR_SALARIO
1	8300.00
1	MAIOR_SALARIO
1	1200.00

- **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 ID_DEPARTAMENTO = 2
```

Results		Messages	
	SOMA_SALARIOS		
1	136920.00		
	SOMA_SALARIOS		
1	19430.00		

- **APPROX\_COUNT\_DISTINCT**

Conta os valores distintos. Podemos realizar a mesma operação usando **COUNT (nome do campo)**.

```
--COUNT conta as linhas
SELECT COUNT(*) AS QTD_EMPREGADOS FROM TB_EMPREGADO;
--COUNT conta os valores não nulos
SELECT COUNT(ID_DEPARTAMENTO) AS QTD_EMPREGADOS FROM TB_EMPREGADO;
--COUNT com o nome do campo, realiza a contagem sem valores duplicados
SELECT COUNT(DISTINCT ID_DEPARTAMENTO) AS QTD_EMPREGADOS
FROM TB_EMPREGADO;
--COUNT com o nome do campo, realiza a contagem sem valores duplicados
SELECT APPROX_COUNT_DISTINCT(ID_DEPARTAMENTO) AS QTD_EMPREGADOS
FROM TB_EMPREGADO;
```

- **COUNT\_BIG ( [ ALL | DISTINCT ] expressão)**

**COUNT\_BIG** é similar a **COUNT**, porém o retorno vai ser um tipo de dados **BIGINT**.

```
--COUNT conta as linhas
SELECT COUNT_BIG(*) AS QTD_EMPREGADOS FROM TB_EMPREGADO;
--COUNT conta os valores não nulos
SELECT COUNT_BIG(ID_DEPARTAMENTO) AS QTD_EMPREGADOS FROM TB_EMPREGADO;
--COUNT com o nome do campo, realiza a contagem sem valores duplicados
SELECT COUNT_BIG(DISTINCT ID_DEPARTAMENTO) AS QTD_EMPREGADOS
FROM TB_EMPREGADO;
```

Results	
Messages	
1	QTD_EMPREGADOS
1	61
Results	
1	QTD_EMPREGADOS
1	58
Results	
1	QTD_EMPREGADOS
1	12

- **STDEV**

**STDEV** retorna o desvio padrão estático.

```
--Apresenta o desvio padrão estático
SELECT STDEV (VLR_TOTAL) AS DESVIO_PADRAO FROM TB_PEDIDO;
```

Results	
Messages	
1	DESVIO_PADRAO
1	2458,98503468705

- **STDEVP**

**STDEVP** retorna o desvio padrão estático para todos os valores.

```
--Apresenta o desvio padrão estático
SELECT STDEVP (VLR_TOTAL) AS DESVIO_PADRAO FROM TB_PEDIDO;
```

Results	
Messages	
1	DESVIO_PADRAO
1	2458,87705647662

- VARP

VARP retorna a variância estática.

```
--Apresenta a variância estática
SELECT VARP (VLR_TOTAL) AS Variancia FROM TB_PEDIDO;
```

	Results	Messages
	Variancia	
1	6046076,37886711	

## 6.3. 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 ID_DEPARTAMENTO, SALARIO  
FROM TB_EMPREGADO  
ORDER BY ID_DEPARTAMENTO;
```

	ID_DEPARTAMENTO	SALARIO
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
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	6300.00
12	1	5000.00
13	1	3300.00

Veja que o campo **ID\_DEPARTAMENTO** 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 ID_DEPARTAMENTO, SUM( SALARIO ) AS TOT_SAL
FROM TB_EMPREGADO
GROUP BY ID_DEPARTAMENTO
ORDER BY TOT_SAL;
```

ID_DEPARTAMENTO	TOT_SAL
1	NULL
2	800.00
3	1200.00
4	3300.00
5	3330.00
6	4500.00
7	5100.00
8	5800.00
9	11200.00
10	18190.00

- Exemplo 2

```
-- GROUP BY + JOIN
SELECT E.ID_DEPARTAMENTO, D.DEPARTAMENTO, SUM( E.SALARIO ) AS
TOT_SAL
FROM TB_EMPREGADO E
JOIN TB_DEPARTAMENTO D ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
GROUP BY E.ID_DEPARTAMENTO, D.DEPARTAMENTO
ORDER BY TOT_SAL;
```

ID_DEPARTAMENTO	DEPARTAMENTO	TOT_SAL
12	PORTARIA	800.00
7	TELEMARKETING	1200.00
14	P.C.P.	3300.00
9	RECURSOS HUMANOS	3330.00
11	PRESIDENCIA	4500.00
8	FINANCEIRO	5100.00
6	DIRETORIA	5800.00
3	CONTROLE DE ESTOQUE	11200.00
4	COMPRAS	18190.00
5	PRODUCAO	18590.00
2	TI	19430.00
1	PESSOAL	45480.00

- Exemplo 3

```
-- Consulta do tipo RANKING utilizando TOP n + ORDER BY
-- Os 5 departamentos que mais gastam com salários
SELECT TOP 5 E.ID_DEPARTAMENTO, D.DEPARTAMENTO,
SUM( E.SALARIO ) AS TOT_SAL
FROM TB_EMPREGADO E
JOIN TB_DEPARTAMENTO D ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
GROUP BY E.ID_DEPARTAMENTO, D.DEPARTAMENTO
ORDER BY TOT_SAL DESC;
```

ID_DEPARTAMENTO	DEPARTAMENTO	TOT_SAL
1	PESSOAL	45480.00
2	TI	19430.00
5	PRODUCAO	18590.00
4	COMPRAS	18190.00
3	CONTROLE DE ESTOQUE	11200.00

- Exemplo 4

```
-- Os 10 clientes que mais compraram em Janeiro de 2016
SELECT TOP 10 C.ID_CLIENTE, C.NOME,
SUM(P.VLR_TOTAL) AS TOT_COMPRADO
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.ID_CLIENTE = C.ID_
CLIENTE
WHERE P.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31'
GROUP BY C.ID_CLIENTE, C.NOME
ORDER BY TOT_COMPRADO DESC;
```

ID_CLIENTE	NOME	TOT_COMPRADO
392	KLJIPFRUSHVKCBAXMLBJNTNKSYXXVH	11246.12
238	WFIUUHUEJNDPLUDREXPJSWDWCNICYV	10483.67
47	KVLJBXVWXDXNEPNDFXINTIABOFVNOR	9915.97
24	NIKMNRCUHQYUIGHFFMRVTNCNMYLVNEE	9312.59
334	IVEPNOCHNKXLUWDFOIKTDCBUECOJL	7799.00
179	MJUVDEHRVHHFPEJORQBJXCMGCBXOFV	7734.76
175	DKCWTMEOXSWEBCCJKHSSVDQFCKEWQQ	7684.40
167	RMBEAGQTDOJLRMFAVQUOFEPMNTQJJMQ	6997.42
370	BLVHKIALADHVDJRGLKXQPFTIBUBVIN	6968.31
234	MCVNJTMBXLEQIBTGMTYGOKENFAXBQN	6468.09

### 6.3.1. 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 2016. Veremos que
-- todas as linhas do resultado terão um total não nulo.
SELECT C.ID_CLIENTE, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRADO
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.ID_CLIENTE = C.ID_
CLIENTE
WHERE P.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31'
GROUP BY C.ID_CLIENTE, C.NOME;
```

ID_CLIENTE	NOME	TOT_COMPRADO
19	XJKPOXBKHCGCUMOMUJHEJGHTCJMMVN	5224.56
24	NIKMNRCUHQYUIGHHFMRVTNCMYLVNEE	9312.59
31	FNMVHMKMSFITGCFJEMLDUPBGDPICFD	1862.67
32	NOGIQEKLUSLPBFMBSPBECGRSAXTCDE	2878.63
35	BPKTSOWDVBSICYNNAUKRIVEXJSXPIGU	3536.30
47	KVLJBXWDXNEPNDFXINTIABOFVYNOR	9915.97
51	POVWJLCMQXOWEWLKVTKUHVNVJTDXGMO	5884.49
73	BKXIEMENBHRANPEPOOJJDOCNOULCFI	4137.87
85	HKYFJYJOSKMBAVEEYUUUKDCQKGKEKE	135.51
88	KROKWNIKEWRPORAORLSFBVFXDGXDVQJ	3533.71
114	NEUVCSKSFIAMDDMTMSUYOEQVCNFBVU	5042.72

```
-- Neste caso, aparecerão também os clientes que não
-- compraram. Totais estarão nulos.
```

```
SELECT C.ID_CLIENTE, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRADO
FROM TB_PEDIDO P JOIN TB_CLIENTE C ON P.ID_CLIENTE = C.ID_
CLIENTE
WHERE P.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31'
GROUP BY ALL C.ID_CLIENTE, C.NOME;
```

ID_CLIENTE	NOME	TOT_COMPRADO
431	IFSBSDDUQLIXNJQQCEGTNIJBSWFCKL	NULL
232	JRVBJMTMUGGVRXFRSLSIWRUEQLHHJ	NULL
313	IFOBYPGBKWKLEGPLNWVUEKENGUNRLM	NULL
260	WMUVSBwwLSUHTWAJHYS CUXUEUWRNQ	NULL
623	SSLPGFQQAVVQFURKXMMGJPNLKRDDBG	NULL
162	MDBNXCXBIUJURXRYSIIXKOCABPXXII	NULL
151	AIBFYQAKWHLTXHIMNRDQPJKUSLXICM	NULL
443	HYQFPWTQWKHEVLSLATXOASMXGSC	NULL
516	QLUDYCCXAMCFNHDYKUVUVYRKYEGQMW	NULL
495	TICMQXMWLODWLHBBDQPUQQXUGFRVV	NULL
190	JFWAWERCJO00VCPNTVYHFUFQERKGOIM	NULL
60	PEGBNYVFGPHOWDLIGOVCPTYKRECJGG	NULL

### 6.3.2. 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\$10.000,00 em salários:

```
SELECT E.ID_DEPARTAMENTO, D.DEPARTAMENTO,
SUM( E.SALARIO ) AS TOT_SAL
FROM TB_EMPREGADO E
JOIN TB_DEPARTAMENTO D ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
GROUP BY E.ID_DEPARTAMENTO, D.DEPARTAMENTO
HAVING SUM(E.SALARIO) > 10000
ORDER BY TOT_SAL;
```

ID_DEPARTAMENTO	DEPARTAMENTO	TOT_SAL
1	CONTROLE DE ESTOQUE	11200.00
2	COMPRAS	18190.00
3	PRODUÇÃO	18590.00
4	TI	19430.00
5	PESSOAL	45480.00

O próximo código consulta os clientes que compraram mais de R\$5.000,00 em janeiro de 2016:

```
SELECT C.ID_CLIENTE, C.NOME, SUM(P.VLR_TOTAL) AS TOT_COMPRADO
FROM TB_PEDIDO P
JOIN TB_CLIENTE C ON P.ID_CLIENTE = C.ID_CLIENTE
WHERE P.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31'
GROUP BY C.ID_CLIENTE, C.NOME
HAVING SUM(P.VLR_TOTAL) > 5.000
ORDER BY TOT_COMPRADO;
```

	ID_CLIENTE	NOME	TOT_COMPRADO
1	85	HKYFJYJOSKMBAVEEYUUKDCQXGTKEKE	135.51
2	337	EDTDGJGTLEMWVJHGMWPCQTRGUODTBD	155.51
3	609	SBMOLFHRUGWLXHRRILRWONJHRCCPQL	719.06
4	177	LVQNHHJUQCKUABE0BLMLPAWTSTEEVRW	817.30
5	193	PNYWVAEXLFDLVKSMKKGHJOSIDWJHJ	826.94
6	194	WFGGGFSEIEHCQURCNDGGDOJWESOVS	965.46
7	377	XOPDXSFUCBBBSEEWEPFDDKLWESUMH	1069.63
8	184	OYLCPTMQJIMKQFWOWSTOKRHOMUDK	1079.50
9	157	PCOFDXFVRTSUQHQBURKEUUJKDMBLTP	1104.61
10	331	MRGTIKNQLIAKUBXSYSQXWIVWCKWHFX	1176.97

Já o próximo código consulta os clientes que não realizaram compras em janeiro de 2016:

```
SELECT C.ID_CLIENTE, C.NOME,
SUM(P.VLR_TOTAL) AS TOT_COMPRADO
FROM TB_PEDIDO P
JOIN TB_CLIENTE C ON P.ID_CLIENTE = C.ID_CLIENTE
WHERE P.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.1.31'
GROUP BY ALL C.ID_CLIENTE, C.NOME
HAVING SUM(P.VLR_TOTAL) IS NULL;
```

	ID_CLIENTE	NOME	TOT_COMPRADO
1	431	IFBSDDUQLJXNJOOQEGTNIJBSWFCKL	NULL
2	232	JRVBJMTMUGGVFRSLSIVRUEQLHHJ	NULL
3	313	IFOBYPGBWKLEGPLNWVUEKENGUNRLM	NULL
4	260	WMUVSBWWLSUHTWAJHYSUXUEUWRNQ	NULL
5	623	SSLPGFQQAWQFURKFMMGJPNLKRDG	NULL
6	162	MDBNXCXBIJJURXRYSIIXKOCABPXXII	NULL
7	151	AIBFYQAKWHLTXHIMNRDQPJKUSLXICM	NULL
8	443	HYQFPVJTFQWKHEVLSELATXOASMXGSC	NULL
9	516	QLUOYCCXAMCFNHDYKUVUVMYRKYEGQMW	NULL
10	495	TICMQXMWLODWLHBBHDQPUQQXUGFRRV	NULL

## 6.3.3. 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 2016:

```
SELECT
    E.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      AS C ON PE.ID_CLIENTE = C.ID_CLIENTE
    JOIN TB_EMPREGADO    AS E ON PE.ID_EMPREGADO = E.ID_
EMPREGADO
    JOIN TB_ITENSPEDIDO   AS I ON PE.ID_PEDIDO = I.ID_
PEDIDO
    JOIN TB_PRODUTO        AS PR ON I.ID_PRODUTO = PR.ID_PRODUTO
    JOIN TB_TIPOPRODUTO    AS T ON PR.ID_TIPO = T.ID_TIPO
WHERE PE.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.6.30'
GROUP BY E.NOME , C.NOME, T.TIPO;
```

Suponha que o resultado retornado seja o seguinte:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
1	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	CANETA	314
2	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	CHAVEIRO	611
3	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	MATL DIVERSOS	537
4	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	PORTA MOEDAS	270
5	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	YO-YO	262
6	CARLOS ALBERTO	ANXTUDDIVPWQUVINKNFBVLEOSUODXI	MATL DIVERSOS	464
7	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	CANETA	741
8	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	CHAVEIRO	1573
9	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	REGUA	344
10	CARLOS ALBERTO	BPKTSOWDVBSICYNNAUKRIVEXJSXPIGU	ACESSORIOS P/CANETA	261
11	CARLOS ALBERTO	BPKTSOWDVBSICYNNAUKRIVEXJSXPIGU	MATL DIVERSOS	167
12	CARLOS ALBERTO	BXWDJFGCLUSFCAIOPMGTSRCETPKLJ	CANETA	933

Agora, acrescente a cláusula **WITH ROLLUP** após a linha de **GROUP BY**. O código anterior ficará assim:

```

SELECT
    E.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          AS C ON PE.ID_CLIENTE = C.ID_CLIENTE
    JOIN TB_EMPREGADO         AS E ON PE.ID_EMPREGADO = E.ID_
EMPREGADO
    JOIN TB_ITENSPEDIDO       AS I ON PE.ID_PEDIDO = I.ID_
PEDIDO
    JOIN TB_PRODUTO           AS PR ON I.ID_PRODUTO = PR.ID_PRODUTO
    JOIN TB_TIPOPRODUTO        AS T ON PR.ID_TIPO = T.ID_TIPO
WHERE PE.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.6.30'
GROUP BY E.NOME , C.NOME, T.TIPO
WITH ROLLUP;
  
```

O resultado será o seguinte:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
1	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	CANETA	314
2	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	CHAVEIRO	611
3	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	MATL DIVERSOS	537
4	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	PORTA MOEDAS	270
5	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	YO-YO	262
6	CARLOS ALBERTO	ADSMIAVQOCMSKMNEMMOXXWAKXUGMAWW	NULL	1994
7	CARLOS ALBERTO	ANXTUDDIVPWQUVINKNFBVLEOSUODXI	MATL DIVERSOS	464
8	CARLOS ALBERTO	ANXTUDDIVPWQUVINKNFBVLEOSUODXI	NULL	464
9	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	CANETA	741
10	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	CHAVEIRO	1573
11	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	REGUA	344
12	CARLOS ALBERTO	BLASTBVXSIFCJPEIIMMRGHEEGLMSDS	NULL	2658

Observe, na figura anterior, que, no campo **TIPO\_PRODUTO**, aparecem linhas com valor nulo. Nesse caso é o somatório das quantidades do vendedor e do cliente.

### 6.3.4. 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
    E.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      AS C  ON PE.ID_CLIENTE = C.ID_CLIENTE
    JOIN TB_EMPREGADO    AS E  ON PE.ID_EMPREGADO = E.ID_EMPREGADO
    JOIN TB_ITENSPEDEIDO AS I  ON PE.ID_PEDIDO = I.ID_PEDIDO
    JOIN TB_PRODUTO       AS PR ON I.ID_PRODUTO = PR.ID_PRODUTO
    JOIN TB_TIPOPRODUTO   AS T  ON PR.ID_TIPO = T.ID_TIPO
WHERE PE.DATA_EMISSAO BETWEEN '2016.1.1' AND '2016.6.30'
GROUP BY E.NOME, C.NOME, T.TIPO
WITH CUBE;
```

O resultado é o seguinte:

	VENDEDOR	CLIENTE	TIPO_PRODUTO	QTD_TOT
1	RENAN	ADSMAIVQOCMSKMENMOXXWAKXUGMAVV	ABRIDOR	230
2	NULL	ADSMAIVQOCMSKMENMOXXWAKXUGMAVV	ABRIDOR	230
3	MARCELO	AGMRBFMSMVXKFNEOUCHIHIWGXVRPPGM	ABRIDOR	244
4	NULL	AGMRBFMSMVXKFNEOUCHIHIWGXVRPPGM	ABRIDOR	244
5	PAULO	BFBBJWUTATNIIIFCKSQIPJOQRBLREI	ABRIDOR	513
6	NULL	BFBBJWUTATNIIIFCKSQIPJOQRBLREI	ABRIDOR	513
7	PAULO	BJUUMJBRSXVSPJPVFVBHKWLSRJMDT	ABRIDOR	103
8	NULL	BJUUMJBRSXVSPJPVFVBHKWLSRJMDT	ABRIDOR	103
9	JOAO	BKXIEMENBHRANPEPOOJJDOCNOULCFI	ABRIDOR	318
10	NULL	BKXIEMENBHRANPEPOOJJDOCNOULCFI	ABRIDOR	318
11	PAULO	BLVHKIALADHVDJRGKXQPFTIBUBVIN	ABRIDOR	718

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**.



### Pontos principais

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

- As funções de agregação fornecidas pelo SQL Server permitem summarizar dados. Por meio delas, é possível somar valores, calcular média e contar resultados. Os cálculos feitos pelas funções de agregação são feitos com base em um conjunto ou grupo de valores, porém retornam um único valor. Para obter os valores sobre os quais você poderá realizar os cálculos, geralmente são utilizadas as funções de agregação com a cláusula **GROUP BY**;
- Utilizando a cláusula **GROUP BY**, é possível agrupar diversos registros com base em uma ou mais colunas da tabela.

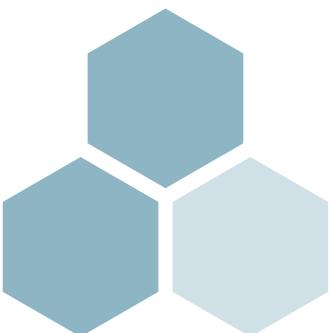




6

## Agrupando dados

Teste seus conhecimentos



## 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.ID_DEPARTAMENTO, D.DEPARTAMENTO, SUM( E.SALARIO  
) AS TOT_SAL  
FROM TB_EMPREGADO AS E  
JOIN TB_DEPARTAMENTO D ON E.ID_DEPARTAMENTO = D.ID_DEPARTAMENTO
```

- 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. Qual a alternativa 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.





6

# Agrupando dados

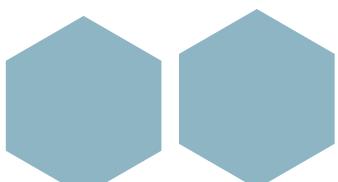


Mãos à obra!

Bruna Morimoto  
397.642.98-78



Editora  
**IMPACTA**



## Laboratório 1

### A – Realizando consultas e ordenando dados

1. Coloque em uso o banco de dados **db\_Ecommerce**;
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 2018 (o maior e o menor valor total, **VLR\_TOTAL**);
4. Calcule o valor total vendido (soma de **TB\_PEDIDO.VLR\_TOTAL**) em janeiro de 2016;
5. Calcule o valor total vendido pelo vendedor de código 4 em janeiro de 2017;
6. Calcule o valor total vendido pela vendedora **LEIA** em janeiro de 2018;
7. Calcule o valor total vendido pelo vendedor **MARCELO** em janeiro de 2016;
8. Calcule o valor da comissão (soma de **TB\_PEDIDO.VLR\_TOTAL \* TB\_VENDEDOR.PORC\_COMISSAO/100**) que a vendedora **ELIANE** recebeu em janeiro de 2017;
9. Calcule o valor da comissão que o vendedor **MARCELO** recebeu em janeiro de 2017;
10. Liste os totais vendidos por cada vendedor (mostrar **TB\_VENDEDOR.NOME** e a soma de **TB\_PEDIDO.VLR\_TOTAL**) em janeiro de 2017. Deve-se exibir o nome do vendedor;
11. Liste o total comprado por cada cliente em janeiro de 2017. Deve-se mostrar o nome do cliente;
12. Liste o valor e a quantidade total vendida de cada produto em janeiro de 2017;
13. Liste os totais vendidos por cada vendedor em janeiro de 2018. 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 2019. 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 2018. Devemos mostrar apenas os produtos que venderam mais de R\$16.000,00;
16. Liste o total comprado por cada cliente em janeiro de 2017. 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 2016. Devemos mostrar os 10 produtos que mais venderam;
18. Liste o total vendido em cada um dos meses de 2017.

# 7

## Modelando um banco de dados

- Design do banco de dados;
- Normalização de dados;
- Tipos de dados;
- Tabelas;
- CONSTRAINTS;
- Apagando tabelas;
- Alterando tabelas.

### 7.1. 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.

#### 7.1.1. 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 Impacta Natural 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 Impacta Natural 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.

## 7.1.2. Modelo conceitual

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

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

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

- CLIENTES = Código + CNPJ + razão social + ramo de atividade + data do cadastramento + {telefones} + {endereços} + pessoa de contato;
- EMPREGADOS = Matrícula + nome + {telefones} + cargo + salário + data de admissão + qualificações + endereço;
- EMPRESAS = CNPJ + razão social + {telefones} + pessoa de contato + endereço;
- FORNECEDORES = CNPJ + razão social + endereço + {telefones} + pessoa de contato;
- TIPO DE ENDEREÇO = Código + nome;
- ENDEREÇOS = Número + logradouro + complemento + CEP + bairro + cidade + Estado;
- ENCOMENDAS = Número + data da inclusão + valor total + valor do desconto + valor líquido + ID forma de pagamento + quantidade de parcelas;
- PRODUTOS = Código + nome + cor + dimensões + peso + preço + tempo de fabricação + desenho do produto + horas de mão de obra;
- TIPOS DE COMPONENTE = Código + nome;
- COMPONENTES = Código + nome + quantidade em estoque + preço unitário + unidade;
- MÁQUINAS = Tempo de vida + data da compra + data fim da garantia;
- RE = Quantidade necessária + unidade + tempo de uso + horas da mão de obra;
- RM = Data + descrição;
- RS = Quantidade + data de necessidade.

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

### 7.1.3. Modelo lógico

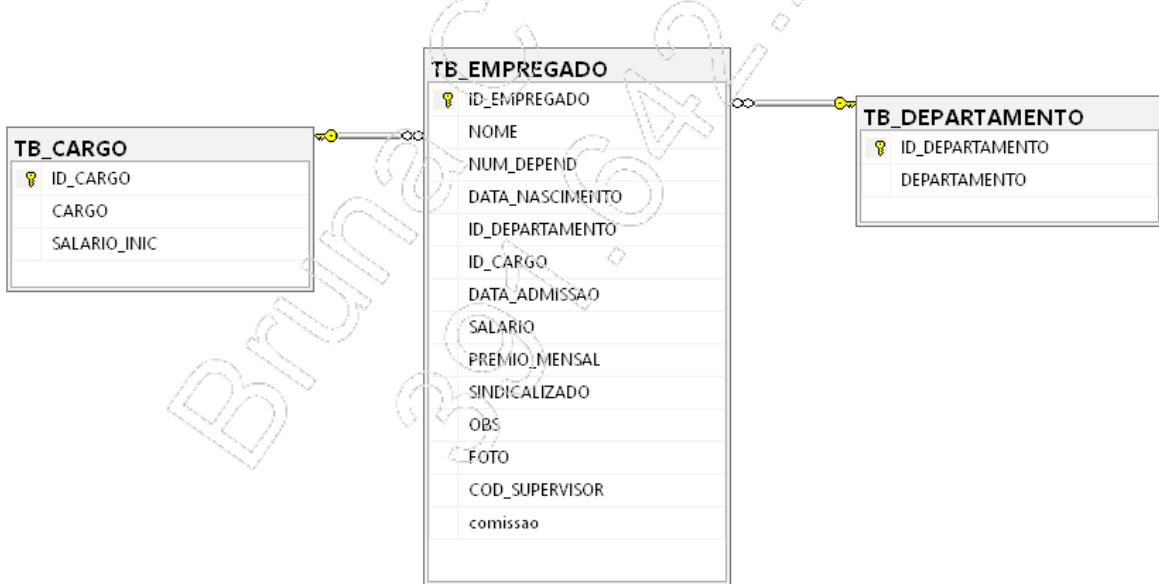
Esse modelo apresenta, em um formato de diagrama, as entidades e os atributos encontrados nos modelos anteriores. Nesse modelo, também conhecido como diagrama lógico de dados, ainda não é possível determinar qual banco de dados será utilizado.

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.



### 7.1.4. 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 2019.

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;
- **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:

TB_EMPREGADO		
Column Name	Data Type	Allow Nulls
ID_EMPREGADO	int	<input type="checkbox"/>
NOME	varchar(50)	<input checked="" type="checkbox"/>
NUM_DEPEND	smallint	<input checked="" type="checkbox"/>
DATA_NASCIMENTO	datetime	<input checked="" type="checkbox"/>
ID_DEPARTAMENTO	int	<input checked="" type="checkbox"/>
ID_CARGO	int	<input checked="" type="checkbox"/>
DATA_ADMISSAO	datetime	<input checked="" type="checkbox"/>
SALARIO	decimal(10, 2)	<input checked="" type="checkbox"/>
PREMIO_MENSAL	decimal(10, 2)	<input checked="" type="checkbox"/>
SINDICALIZADO	char()	<input checked="" type="checkbox"/>
OBS	varchar(MAX)	<input checked="" type="checkbox"/>
FOTO	varbinary(MAX)	<input checked="" type="checkbox"/>
COD_SUPERVISOR	int	<input checked="" type="checkbox"/>
comissao	decimal(5, 2)	<input checked="" type="checkbox"/>

TB_CARGO		
Column Name	Data Type	Allow Nulls
ID_CARGO	int	<input type="checkbox"/>
CARGO	varchar(30)	<input checked="" type="checkbox"/>
SALARIO_INIC	decimal(10, 2)	<input checked="" type="checkbox"/>

TB_DEPARTAMENTO		
Column Name	Data Type	Allow Nulls
ID_DEPARTAMENTO	int	<input type="checkbox"/>
DEPARTAMENTO	varchar(30)	<input checked="" type="checkbox"/>

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

## 7.1.5. 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				

## 7.2. 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.

### 7.2.1. 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.

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_COML
	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.

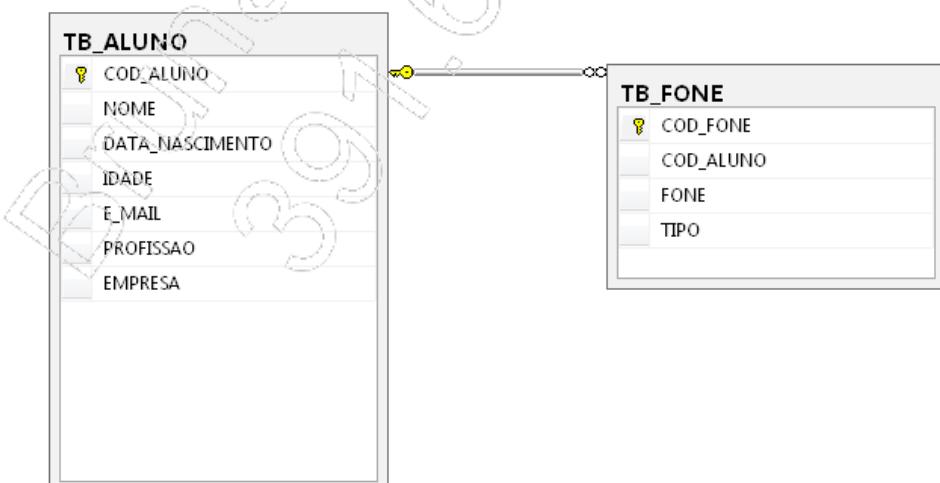
Verifique que existe um conjunto de dados repetidos: **FONE\_RES**, **FONE\_COML**, **FONE\_CELULAR** e **FONE\_RECADO**.

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.

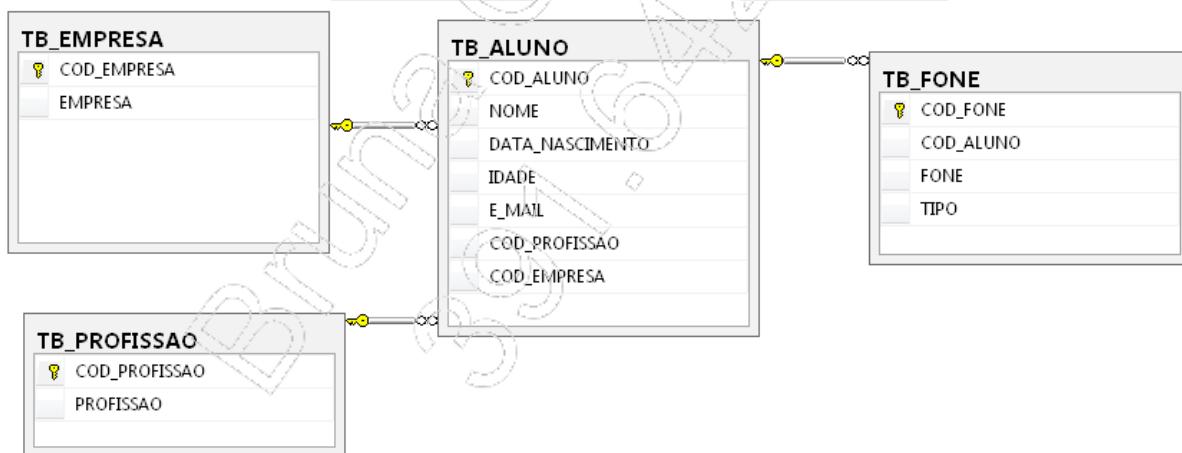
Com a finalidade de criar relacionamentos, devemos relacionar essas novas tabelas com uma chave estrangeira e identificar cada grupo de dados relacionados com uma chave primária.

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)

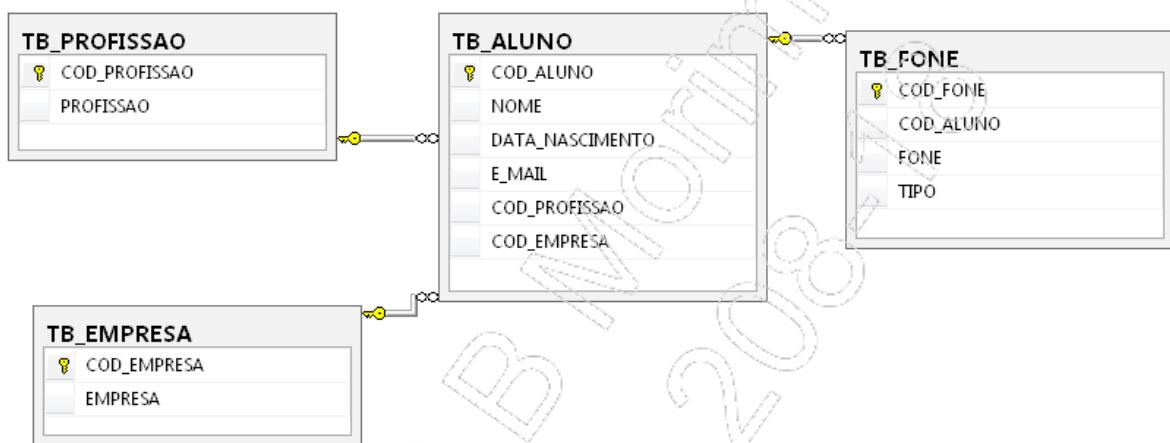
No terceiro nível de normalização, após ter concluído todas as tarefas do **1NF** e **2NF**, devemos eliminar os campos que não dependem de chaves primárias.

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:



## 7.3. 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.

No SQL Server 2019, foi implementado o suporte a caracteres UTF-8. A vantagem da utilização desse conjunto de caracteres é a internacionalização dos dados.

Para conhecer quais COLLECTIONS possuem suporte a UTF-8, realize a consulta:

```
SELECT NAME, DESCRIPTION
FROM FN_HELPCOLLATIONS()
WHERE NAME LIKE '%UTF8';
```

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

Além disso, também serão apresentados os tipos de dados suportados pelo T-SQL.

### 7.3.1. Numéricos exatos

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

- **Inteiros**

Nome	Descrição
<b>BIGINT 8 BYTES</b>	Valor de número inteiro compreendido entre $-2^{63}$ (-9,223,372,036,854,775,808) e $2^{63}-1$ (9,223,372,036,854,775,807).
<b>INT 4 BYTES</b>	Valor de número inteiro compreendido entre $-2^{31}$ (-2,147,483,648) e $2^{31}-1$ (2,147,483,647).
<b>SMALLINT 2 BYTES</b>	Valor de número inteiro compreendido entre $-2^{15}$ (-32,768) e $2^{15}-1$ (32,767).
<b>TINYINT 1 BYTE</b>	Valor de número inteiro de 0 a 255.

- **Bit**

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

- **Numéricos exatos**

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

Nos numéricos exatos, é importante considerar as seguintes informações:

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

- Valores monetários

Nome	Descrição
<b>MONEY 8 BYTES</b>	Compreende valores monetários ou de moeda corrente entre -922.337.203.685.477,5808 e 922.337.203.685.477,5807.
<b>SMALLMONEY 4 BYTES</b>	Compreende valores monetários ou de moeda corrente entre -214,748.3648 e +214,748.3647.

## 7.3.2. Numéricos aproximados

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

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

Em que:

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

! Esses tipos são chamados de “Numéricos aproximados” porque podem gerar imprecisão na parte decimal.

## 7.3.3. Data e hora

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

Nome	Descrição
<b>DATETIME 8 BYTES</b>	Data e hora compreendidas entre 1º de janeiro de 1753 e 31 de dezembro de 9999, com a exatidão de 3.33 milissegundos.
<b>SMALLDATETIME 4 BYTES</b>	Data e hora compreendidas entre 1º de janeiro de 1900 e 6 de junho de 2079, com a exatidão de 1 minuto.

Nome	Descrição
<b>DATETIME2(P) 8 BYTES</b>	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.
<b>DATE 3 BYTES</b>	Data compreendida entre 01/01/0001 e 31/12/9999, com precisão de 1 dia.
<b>TIME(P) 5 BYTES</b>	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.
<b>DATETIMEOFFSET(P)</b>	Data e hora compreendidas entre 01/01/0001 e 31/12/9999 com precisão de até 100 nanossegundos e com indicação do fuso horário, cujo intervalo pode variar de -14:00 a +14:00. O parâmetro p indica a quantidade de dígitos na fração de segundo.

### 7.3.4. Caracteres texto

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

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

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

Em que:

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

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

## 7.3.5. Caracteres UNICODE

**UNICODE** é um padrão computacional que permite a utilização de caracteres mundiais, como: japonês, árabe, chinês etc.

Em caracteres 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.

## 7.3.6. Valores binários

No caso de valores binários, 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.

### 7.3.7. Outros tipos de dados

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

Nome	Descrição
<b>TABLE</b>	Serve para definir um dado tabular, composto de linhas e colunas, assim como uma tabela.
<b>CURSOR</b>	Serve para percorrer as linhas de um dado tabular.
<b>SQL_VARIANT</b>	Um tipo de dado que armazena valores de vários tipos suportados pelo SQL Server, exceto os seguintes: <b>TEXT</b> , <b>NTEXT</b> , <b>TIMESTAMP</b> e <b>SQL_VARIANT</b> .
<b>TIMESTAMP ou ROWVERSION</b>	Número hexadecimal sequencial gerado automaticamente.
<b>UNIQUEIDENTIFIER</b>	GLOBALLY UNIQUE IDENTIFIER (GUID), também conhecido como Identificador Único Global ou Identificador Único Universal. É um número hexadecimal de 16 bytes semelhante a 64261228-50A9-467C-85C5-D73C51A914F1.
<b>XML</b>	Armazena dados no formato XML.
<b>HIERARCHYID</b>	Posição de uma hierarquia.
<b>GEOGRAPHY</b>	Representa dados de coordenadas terrestres.
<b>GEOMETRY</b>	Representação de coordenadas euclidianas.

## 7.4. Tabelas

O banco de dados SQL Server permite a criação de diversos tipos de tabelas. A gestão dessas tabelas cabe ao administrador de banco de dados, por isso, é fundamental compreender cada tipo de tabela, seus usos, benefícios e consequências de sua utilização.

Devemos entender que as tabelas podem ter finalidades diferentes. Em relação ao tempo, as tabelas podem ser permanentes ou temporárias. Elas ainda podem ser inteiras ou particionadas. Essa prática de particionamento pode ser útil no caso de tabelas com grandes quantidades de dados.

Existem vários tipos de tabelas em um banco de dados SQL Server:

- Tabelas regulares;
- Tabelas temporárias locais;
- Tabelas temporárias globais;
- Tabelas baseadas em consulta;
- Tabelas particionadas (Módulo III);
- Tabelas baseadas em arquivos;
- Tabelas IN-MEMORY (Módulo III).

## 7.4.1. Tabelas regulares

As tabelas regulares são as tabelas criadas normalmente por meio do comando **CREATE TABLE**.

Tabelas regulares podem ser criadas com ou sem as **CONSTRAINTS** (será apresentado logo em seguida), como veremos adiante:

```
CREATE TABLE TB_ALUNO
(
    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)
)
```

## 7.4.2. Tabelas temporárias locais

Tabelas temporárias têm como principal característica a visibilidade apenas para o usuário que a criou e durante a conexão vigente. Elas são eliminadas após a desconexão do usuário e utilizam o banco de dados **TEMPDB**. As tabelas temporárias locais podem utilizar o mecanismo de **CONSTRAINTS**, assim como as tabelas permanentes. Para criarmos uma tabela temporária local, basta adicionar o caractere # no início do nome da tabela.

Exemplo:

```
CREATE TABLE #MATRICULA
(
    NUM_MATRICULA INTEGER NOT NULL ,
    DAT_MATRICULA DATETIME NOT NULL ,
    VAL_MATRICULA DECIMAL (12,2) NOT NULL ,
    COD_ALUNO INTEGER NOT NULL
)
```

### 7.4.3. Tabelas temporárias globais

Tabelas temporárias globais têm como principal característica a visibilidade para todos os usuários. Estas são eliminadas após a desconexão do usuário. As tabelas temporárias globais podem utilizar o mecanismo de CONSTRAINTS, assim como as tabelas permanentes. Para criarmos uma tabela temporária global, basta adicionar o caractere ## no início do nome da tabela.

Exemplo:

```
CREATE TABLE ##MATRICULA
(
    NUM_MATRICULA INTEGER NOT NULL ,
    DAT_MATRICULA DATETIME NOT NULL ,
    VAL_MATRICULA DECIMAL (12,2) NOT NULL ,
    COD_ALUNO INTEGER NOT NULL
)
```

### 7.4.4. Tabelas baseadas em consultas

Tabelas baseadas em consultas são essencialmente tabelas geradas e carregadas durante a execução de uma consulta. Para isso, a tabela gerada não pode ter sido previamente criada, logo, não pode existir. Ela é carregada através do comando **SELECT ... INTO**. Sua origem poderá ser uma tabela única, ou a junção de várias tabelas, ou ainda uma ou mais visões. Essas tabelas são criadas sem o mecanismo de CONSTRAINT, mesmo que nas tabelas originais existam CONSTRAINTS. Esse tipo de tabela pode ser temporária ou permanente.

Exemplo de criação de tabela baseada em consulta:

```
SELECT * INTO ALUNO_BACKUP FROM TB_ALUNO
```

### 7.4.5. Criando tabelas (CREATE TABLE)

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:

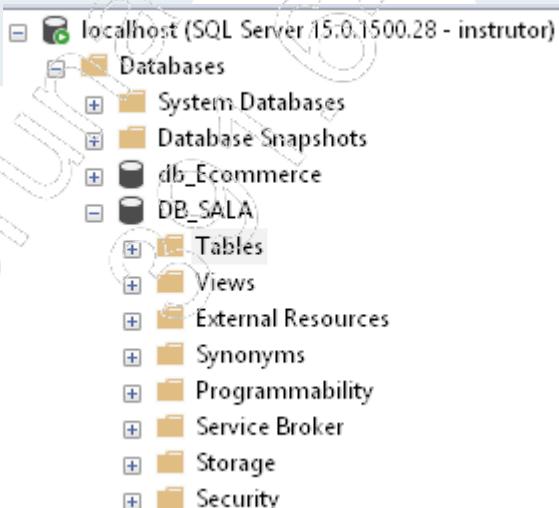
```
CREATE TABLE TB_ALUNO
(
    COD_ALUNO           INT,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO     DATETIME,
    IDADE               TINYINT,
    E_MAIL               VARCHAR(50),
    FONE_RES             CHAR(9),
    FONE_COM             CHAR(9),
    FAX                  CHAR(9),
    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.



## 7.4.6. Auto numeração (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.

Podemos definir o valor inicial e o incremento.

Sem informar o valor inicial e incremento, o valor será inicializado em 1 e o incremento em 1:

**IDENTITY**

Para informar os valores, é necessário abrir parênteses. Adiante o exemplo para iniciar com o valor 1 e incremento 2:

**IDENTITY (1,2)**

É importante saber que uma tabela pode ter apenas uma coluna auto numerável e que não é possível inserir ou alterar seu valor, que é gerado automaticamente pelo T-SQL. 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) );

```

## 7.5. 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**.

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.

### 7.5.1. Nulos

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 );

```

### 7.5.2. Chave Primária (PRIMARY KEY)

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.

Garante que esse campo não possua valores duplicados. Em função da modelagem de dados e do relacionamento existente entre as tabelas, é necessário que a tabela possua chave primária. Uma (chave primária simples) ou mais colunas (chave primária composta) podem fazer parte da chave primária. Neste caso, a soma dos valores de todas as colunas representa a chave primária por si mesma. Também é conhecida como **PRIMARY KEY** ou ainda **PK**. Geralmente as tabelas possuem uma coluna que as representa, mas, caso essa coluna não exista, é possível criá-la. As colunas devem ser sempre listadas como as primeiras de uma tabela, por questões de organização física nas páginas de dados. Toda chave primária é representada por um índice único, que faz o suporte que garante que os dados não serão duplicados. Ela deve ser criada ao mesmo tempo ou depois que a tabela, porém, antes das chaves estrangeiras das tabelas.

Por padrão a **PRIMARY KEY**:

- É única – não permite valores duplicados;
- Não permite valores nulos;
- Gera um índice CLUSTERIZADO, ou seja, ordena toda a tabela pelo campo.
- Exemplos

O banco **DB\_ECOMMERCE** possui as tabelas:

**Tabela:** TB\_CLIENTE

**Campo:** ID\_CLIENTE

**PRIMARY KEY:** PK\_TB\_CLIENTE\_23A34130A6CA98DE

**Tabela:** TB\_PRODUTO

**Campo:** ID\_PRODUTO

**PRIMARY KEY:** PK\_PRODUTOS

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

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.

- Criando uma PRIMARY KEY

É possível criar uma CONSTRAINT PRIMARY KEY, colocando a cláusula PRIMARY KEY após o tipo de dados da coluna desejada.

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

CREATE TABLE TB_ALUNO
(
    NUM_ALUNO          INT PRIMARY KEY 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));

```

No exemplo anterior, o SQL automaticamente vai definir o nome da PRIMARY KEY. O SQL criou a tabela e a PRIMARY KEY de nome: **PK\_\_TB\_ALUNO\_\_002C1E0EE0795079**.

Outra opção é a definição das **CONSTRAINTS** após a definição dos campos. A vantagem dessa opção é de definir o nome do objeto conforme padronização previamente definida.

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

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)
    CONSTRAINT PK_TB_ALUNO PRIMARY KEY (NUM_ALUNO)
);

```

Para as tabelas que já foram criadas, é necessário utilizar a opção **ALTER TABLE**:

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

```
ALTER TABLE TB_ALUNO ADD CONSTRAINT PK_TB_ALUNO PRIMARY KEY  
(NUM_ALUNO)
```

## 7.5.3. Chave única (UNIQUE)

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

Ela não é obrigatória e serve como uma garantia de não duplicidade de uma informação. Além disso, a chave única é uma forma de permitir pesquisas mais precisas, como no caso das chaves primárias. Há muita confusão entre essas duas chaves, porém, para simplificar, devemos entender que a chave única é um mecanismo eficiente e rápido de localização de dados apenas. As chaves únicas também são conhecidas como chaves alternadas ou ainda **UNIQUE KEY** ou **UK**. Para exemplificar, imaginemos uma tabela que contenha dados de alunos. É natural relacionar essa informação com suas matrículas em determinados cursos, logo há uma relação entre matrículas e alunos. Dessa forma, o código do aluno pode ser considerado como chave primária da tabela de alunos, porém, os alunos têm registro geral (RG), que é uma informação única. Se a pessoa não souber seu código de aluno, com certeza saberá seu RG. Então, podemos usar essa informação como chave única. Em uma tabela, é possível ter uma ou mais colunas e várias chaves únicas. Por exemplo, se o aluno tiver como informação o CPF, passamos a ter duas chaves únicas individuais, RG e/ou CPF, além da chave primária que é o código do aluno.

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

- **Exemplos**

Na tabela TB\_CLIENTE

Na tabela TB\_TIPOPRODUTO

Na tabela TB\_UNIDADE

Campos CPF, RG e E-mail

Descrição do tipo de produto

Descrição da unidade de medida

- Criação de CONSTRAINT UNIQUE

```
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.

Adiante a criação de uma tabela com uma CONSTRAINT UNIQUE:

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

A seguir, o comando de criação da tabela e da CONSTRAINT UNIQUE, com o comando depois da definição dos campos:

```
CREATE TABLE TB_ALUNO
(
    NUM_ALUNO           INT PRIMARY KEY 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)
    CONSTRAINT UQ_TB_ALUNO_E_MAIL UNIQUE (E_MAIL)
);
```

Ou com o comando será o seguinte, depois de criada a tabela:

```
ALTER TABLE TB_ALUNO ADD CONSTRAINT UQ_TB_ALUNO_E_MAIL UNIQUE  
(E_MAIL);
```

O nome da CONSTRAINT deve ser sugestivo, informando o tipo da CONSTRAINT (UQ), o nome da tabela e o nome do campo. Exemplo: **UQ\_TB\_ALUNO\_E\_MAIL**.

## 7.5.4. Checagem (CHECK)

CONSTRAINT de **CHECK** é a regra que garante que uma determinada coluna terá apenas valores específicos, impedindo que valores indesejados sejam atribuídos a ela. Por exemplo, temos a coluna sexo, na qual são esperados dois valores: **M** para representar o sexo masculino, **F** para o feminino e mais nenhum outro valor além desses dois. Cada tabela pode conter várias colunas com CONSTRAINTS de 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 TB\_CLIENTE

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

Tabela TB\_PRODUTO

- **Comando**

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

Adiante o comando de criação da tabela e da CONSTRAINT CHECK, com o comando depois da definição dos campos:

```
CREATE TABLE TB_ALUNO
(
    NUM_ALUNO             INT PRIMARY KEY IDENTITY,
    NOME                   VARCHAR(30),
    DATA_NASCIMENTO       DATETIME,
    IDADE                  TINYINT,
    E_MAIL                 VARCHAR(50),
    SEXO                   CHAR(1),
    FONE_RES               CHAR(8),
    FONE_COM               CHAR(8),
    FAX                     CHAR(8),
    CELULAR                CHAR(9),
    PROFISSAO              VARCHAR(40),
    EMPRESA                VARCHAR(50),
    CONSTRAINT CK_TB_ALUNO_SEXO CHECK (SEXO IN ('F', 'M'))
);
```

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

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

ALTER TABLE TB_ALUNO ADD CONSTRAINT CK_TB_ALUNO_SEXO CHECK
(SEXO IN ('F', 'M'))
```

O nome da CONSTRAINT deve ser sugestivo. Para isso, utilize: CK + Nome da tabela + regra. Exemplo: CK\_TB\_ALUNO\_SEXO.

## 7.5.5. Valor padrão (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,
    ...
)
```

A seguir, o comando de criação da tabela e da CONSTRAINT DEFAULT, com o comando depois da definição dos campos:

```
CREATE TABLE TB_ALUNO
(
    NUM_ALUNO           INT PRIMARY KEY IDENTITY,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO    DATETIME,
    IDADE               TINYINT,
    E_MAIL              VARCHAR(50),
    SEXO                CHAR(1) DEFAULT ('M'),
    FONE_RES            CHAR(8),
    FONE_COM            CHAR(8),
    FAX                 CHAR(8),
    CELULAR             CHAR(9),
    PROFISSAO           VARCHAR(40),
    EMPRESA             VARCHAR(50))
```

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

```
ALTER TABLE tabela ADD
CONSTRAINT NomeDefault DEFAULT (valorDefault) FOR CAMPO_DE-
FAULT

ALTER TABLE TB_ALUNO ADD CONSTRAINT DF_TB_ALUNO_SEXO DEFAULT
('M') FOR SEXO;
```

Podemos utilizar o acrônimo **DF** para iniciar o nome da CONSTRAINT. Use: DF + Nome da tabela + Nome do campo.

## 7.5.6. FOREIGN KEY (chave estrangeira)

Esta CONSTRAINT indica que uma tabela está ligada a outra. Para isso, é necessário entender que, antes, a CONSTRAINT de chave primária precisa estar ativa na tabela de origem da informação.

Para que a tabela TB\_PEDIDO seja relacionada fisicamente com a tabela TB\_CLIENTE, é necessário que exista um campo que relate as duas: o campo ID\_CLIENTE, presente nas duas tabelas e com informações. ID\_CLIENTE é chave primária da tabela TB\_CLIENTE e é necessário que seja uma FOREIGN KEY na tabela TB\_PEDIDO.

# Modelando um banco de dados

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)
)
```

Primeiro, é necessário que a tabela referenciada possua uma chave primária.

```
CREATE TABLE TB_PROFISSAO
(
    ID_PROFISSAO INT IDENTITY PRIMARY KEY,
    PROFISSAO      VARCHAR(50)
)
```

Na tabela que será criada a CONSTRAINT FOREIGN KEY, é necessário que possua um campo com o mesmo tipo de dados da tabela em que vai existir a relação.

```
CREATE TABLE TB_ALUNO
(
    NUM_ALUNO          INT PRIMARY KEY 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),
    ID_PROFISSAO        INT,
    EMPRESA             VARCHAR(50) );

```

Após a criação das tabelas, execute o comando de alteração da tabela **ALTER TABLE**:

```
ALTER TABLE TB_ALUNO ADD CONSTRAINT FK_TB_ALUNO_TB_PROFISSAO
FOREIGN KEY (ID_PROFISSAO)
REFERENCES TB_PROFISSAO (ID_PROFISSAO);
```

Utilize o acrônimo **FK** para iniciar o nome da CONSTRAINT. Use: **FK + Nome da tabela que vai receber o relacionamento + Tabela referenciada**.

### 7.6. Apagando tabelas

Para apagar uma tabela, é necessário utilizar o comando **DROP TABLE**. Lembre-se que essa ação é permanente, então tenha cuidado ao executar esse comando.

Somente é possível apagar tabelas que não possuam FOREIGN KEY associadas.

Ao tentar apagar a tabela TB\_PROFISSAO criada anteriormente, ocorrerá um erro:

```
DROP TABLE TB_PROFISSAO;
```

```
Msg 3726, Level 16, State 1, Line 116  
Could not drop object 'TB_PROFISSAO' because it is referenced  
by a FOREIGN KEY constraint.
```

Para apagar a tabela TB\_PROFISSAO, você deve apagar a CONSTRAINT ou a tabela TB\_ALUNO:

```
DROP TABLE TB_ALUNO;
```

```
DROP TABLE TB_PROFISSAO;
```

### 7.7. Alterando tabelas

Como a área de desenvolvimento é dinâmica, as alterações são constantes e vão ocorrer no ciclo de vida do sistema. Para isso, é possível a modificação das tabelas e CONSTRAINTS. Para alterar uma tabela, existe o comando **ALTER TABLE**. Com ele, podemos realizar as seguintes ações:

- Adicionar CONSTRAINTS;
- Adicionar nova coluna;
- Alterar coluna existente;
- Eliminar uma coluna;
- Habilitar CONSTRAINT;
- Desabilitar CONSTRAINT;
- Eliminar CONSTRAINT;
- Renomear uma tabela.

Vejamos os exemplos a seguir:

# Modelando um banco de dados

- **Exemplo 1** – Adicionando uma nova coluna em uma tabela:

```
CREATE TABLE TB_ALUNO
(
    COD_ALUNO INTEGER NOT NULL ,
    NOM_ALUNO VARCHAR (60) ,
    NUM_CPF NUMERIC (11) ,
    NUM_RG NUMERIC (12) ,
    DES_EMAIL VARCHAR (80) ,
    DAT_NASCIMENTO DATETIME
)
GO

ALTER TABLE TB_ALUNO ADD DAT_CADASTRO DATETIME;
```

- **Exemplo 2** – Adicionando mais de uma nova coluna em uma tabela:

```
ALTER TABLE TB_ALUNO ADD DATA_CADASTRO DATETIME,SIT_ATIVO
CHAR(1) NOT NULL
```

- **Exemplo 3** – Adicionando uma CONSTRAINT (chave primária, única e estrangeira) em uma tabela:

```
ALTER TABLE TB_ALUNO ADD CONSTRAINT ALUNO_PK PRIMARY KEY (COD_ALUNO)
ALTER TABLE TB_ALUNO ADD CONSTRAINT UK_ALUNO_EMAIL UNIQUE
(DES_EMAIL)
ALTER TABLE TB_MATRICULA ADD CONSTRAINT FK_ALUNO_MATRICULA
FOREIGN KEY(COD_ALUNO) REFERENCES TB_ALUNO (COD_ALUNO)
```

Ou:

```
ALTER TABLE TB_ALUNO ADD PRIMARY KEY (COD_ALUNO)

ALTER TABLE TB_ALUNO ADD UNIQUE (DES_EMAIL)

ALTER TABLE TB_MATRICULA ADD
FOREIGN KEY (COD_ALUNO) REFERENCES TB_ALUNO (COD_ALUNO)
```

- **Exemplo 4** – Eliminando uma chave primária, única ou estrangeira:

```
ALTER TABLE TB_ALUNO DROP CONSTRAINT ALUNO_PK
ALTER TABLE TB_ALUNO DROP CONSTRAINT UK_ALUNO_EMAIL
ALTER TABLE TB_MATRICULA DROP CONSTRAINT FK_ALUNO_MATRICULA
```

- **Exemplo 5** – Para desabilitar qualquer CONSTRAINT específica:

```
--Criação da CONSTRAINT  
ALTER TABLE TB_MATRICULA ADD CONSTRAINT CK_MATRICULA_DT_NAO_  
PODE_SER_FUTURA  
    CHECK (DAT_MATRICULA>=GETDATE() )  
  
--Desabilita a CONSTRAINT  
ALTER TABLE TB_MATRICULA NOCHECK CONSTRAINT CK_MATRICULA_DT_  
NAO_PODE_SER_FUTURA
```

- **Exemplo 6** – Para desabilitar todas as CONSTRAINTS de uma tabela:

```
ALTER TABLE TB_MATRICULA NOCHECK CONSTRAINT ALL
```

- **Exemplo 7** – Para habilitar uma CONSTRAINT específica:

```
ALTER TABLE TB_MATRICULA CHECK CONSTRAINT FK_ALUNO_MATRICULA
```

- **Exemplo 8** – Para habilitar todas as CONSTRAINTS em uma tabela:

```
ALTER TABLE TB_MATRICULA CHECK CONSTRAINT ALL
```

- **Exemplo 9** – Para eliminar uma coluna da tabela:

```
ALTER TABLE TB_ALUNO DROP COLUMN DAT_CADASTRO
```

- **Exemplo 10** – Para renomear uma tabela (o comando não é ALTER e sim SP\_RENAME):

```
EXEC SP_RENAME 'TB_MATRICULA', 'MATRICULA_ALUNO'
```

- **Exemplo 11** – Para visualizar a estrutura da tabela:

```
EXEC SP_HELP 'TB_MATRICULA'
```

- **Exemplo 12** – Para listar as tabelas de um banco de dados:

```
SELECT * FROM SYS.TABLES
```

```
SELECT * FROM SYSOBJECTS WHERE XTYPE = 'U'
```

```
EXEC SP_TABLES
```

- **Exemplo 13** – Para eliminar uma tabela:

```
DROP TABLE TB_MATRICULA
```

## Pontos principais

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

- 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;
- 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;
- 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** e **ALTER TABLE**.

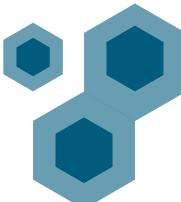
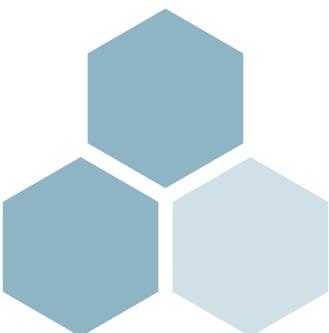




7

# Modelando um banco de dados

Teste seus conhecimentos



## 1. 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.

## 2. O que é Normalização?

- a) Normalização é um processo de limpeza das tabelas e dos dados.
- b) É um processo que deve ser evitado, pois cria muitas tabelas.
- c) A normalização reduz a quantidade de tabelas, permitindo maior performance.
- d) É um processo de organização das tabelas para deixar a estrutura inteligente e rápida.
- e) Um processo que duplica os dados das tabelas.

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

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

## 4. Selecione o comando que cria uma tabela temporária local:

- a) CREATE TABLE ###TABELA ...
- b) CREATE TABLE TMP\_TABELA ...
- c) CREATE TABLE TABELA ...
- d) CREATE TABLE ##TABELA ...
- e) CREATE TABLE #TABELA ...

## 5. Qual a função da CONSTRAINT FOREIGN KEY?

- a) Criar um índice.
- b) Relacionar fisicamente duas tabelas.
- c) Gerar uma regra de validação.
- d) Validar um campo.
- e) Criar uma estrutura parecida com um índice para comparar duas tabelas.





7

# Modelando um banco de dados



Mãos à obra!

Bruna Morimoto  
397-6428-78



Editora  
**IMPACTA**



## 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	<input type="text"/>	Cidade	<input type="text"/>
Nome	<input type="text"/>	Estado	<input type="text"/>
Data do Cadastro	<input type="text"/>	CEP	<input type="text"/>
Dt de Nascimento	<input type="text"/>	Limite de Compra	<input type="text"/>
Idade	<input type="text"/>		<input type="text"/>
Profissao	<input type="text"/>		<input type="text"/>
Telefone Residencial	<input type="text"/>		<input type="text"/>
Telefone Comercial	<input type="text"/>		<input type="text"/>
Celular	<input type="text"/>		<input type="text"/>
Endereço	<input type="text"/>		<input type="text"/>
Bairro	<input type="text"/>		<input type="text"/>

Realize os passos a seguir:

1. Crie as tabelas e campos necessários para atender a 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 regras em tabelas

1. Caso o banco **DB\_PEDIDO** não esteja criado, execute o script **Cria banco DB\_PEDIDO.sql** localizado na pasta **Cap\_01**;
2. Coloque em uso o banco de dados **DB\_PEDIDO**;
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**.

# 8

## Opções de definição de tabelas

- Tipos de dados definidos pelo usuário;
- Tabelas de sistema;
- Sequências;
- Sinônimos;
- Trabalhando com objetos binários;
- FILETABLE;
- Colunas computadas.

### 8.1. Introdução

No desenvolvimento de um projeto, é necessária a definição de como armazenar as informações. Para isso, utilizamos o recurso de tabelas. Embora já tenham sido exploradas no primeiro módulo, existem mais recursos que ampliam as possibilidades desse tipo de armazenamento. Neste capítulo, iremos explorar esses recursos, que ajudarão a incrementar o desenvolvimento de tabelas.

### 8.2. Tipos de dados definidos pelo usuário

Os usuários podem definir seus próprios tipos de dados com base nos tipos de dados fornecidos pelo SQL Server. Conhecido como **User Defined Datatype (UDDT)**, o **tipo de dados definido pelo usuário** também pode ser considerado um sinônimo de um tipo já disponível. Para trabalharmos com tipos de dados definidos pelo usuário, podemos empregar duas operações:

- **CREATE TYPE**: Responsável por criar um UDDT;
- **DROP TYPE**: Por meio desta operação, podemos eliminar um tipo de dados definido pelo usuário.

Também é possível implementar um User Defined Datatype utilizando o CLR (Common Language Runtime), que pode ser acessado através das ferramentas da plataforma Microsoft .NET.

Derivados dos System Datatypes, os User Defined Datatypes são voltados especificamente para o banco de dados no qual eles estão sendo criados. Portanto, não é possível utilizá-los em outros bancos de dados. A exceção fica por conta dos UDDTs criados no banco de dados modelo, pois os bancos de dados subsequentes também possuirão esses tipos de dados.

Os tipos de dados definidos pelo usuário são atribuídos a variáveis de memória ou associados a colunas de uma tabela, assim como os System Datatypes. As características de um tipo de dados definido pelo usuário provêm dos operadores e métodos de uma classe também criada pelo usuário.

#### 8.2.1. CREATE TYPE

Essa instrução tem a função de criar um tipo de dados definido pelo usuário.

A sintaxe de **CREATE TYPE** é a seguinte:

```
CREATE TYPE <tipo_uddt>
    FROM <tipo_builtin> <tipo_null>;
```

# Opções de definição de tabelas

Em que:

- **tipo\_udt**: É uma string que define o nome do novo tipo de dado criado pelo usuário. Tal nome deve ser exclusivo em um banco de dados;
- **tipo\_builtin**: É uma string que define o tipo de dado nativo em que o novo tipo de dado se baseia;
- **tipo\_null**: É uma string que define se o novo tipo de dado pode aceitar ou não valores nulos.

Ao criar um tipo de dados, ele é adicionado à tabela **SYS.TYPES** do banco de dados em que foi criado. Ele também pode ser disponibilizado em todos os novos bancos de dados, bastando, para isso, que seja adicionado ao banco de dados modelo.

A principal vantagem de criar um tipo de dados de usuário é a possibilidade de associar a ele regras de validação e valores default, como veremos nos tópicos subsequentes.

Vejamos os exemplos adiante:

- Criação do tipo TIPO\_CODIGO baseado no tipo INT e que não permite nulo:

```
CREATE TYPE TIPO_CODIGO FROM INT NULL
```

- Criação do tipo TIPO\_NOME baseado no tipo VARCHAR e que permite valores nulos:

```
CREATE TYPE TIPO_NOME FROM VARCHAR(50);
```

## 8.2.2. DROP TYPE

Essa instrução tem a função de apagar um tipo de dados definido pelo usuário anteriormente.

A sintaxe do **DROP TYPE** é a seguinte:

```
DROP TYPE <tipo_udt>
```

Em que:

- **tipo\_udt**: É o nome do tipo de dados que pretendemos deletar.

Vejamos o exemplo:

```
DROP TYPE TIPO_CODIGO;
```

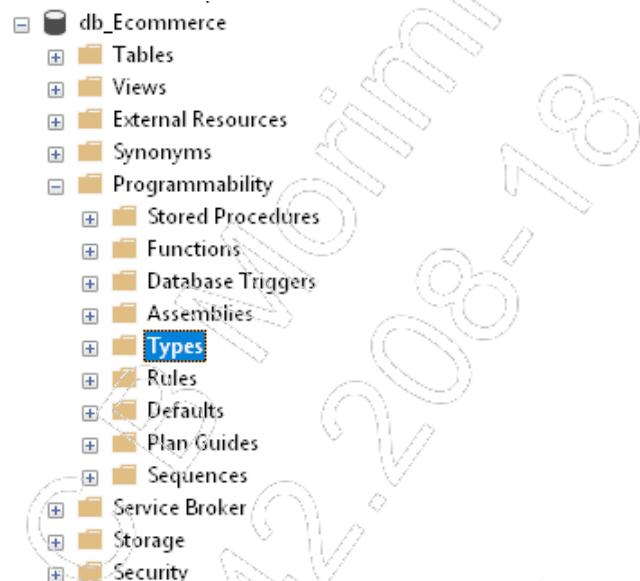
```
DROP TYPE TIPO_NOME;
```

## 8.2.3. Regras e valores padrão

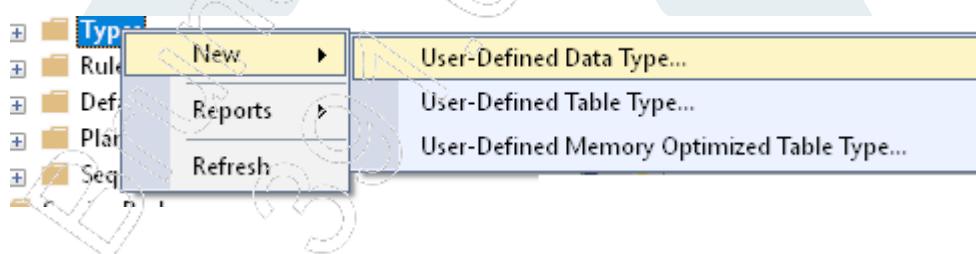
Para criar regras e valores DEFAULT, é recomendada a utilização de CONSTRAINTS. Os comandos CREATE RULE, CREATE DEFAULT, SP\_BINDEFAULT e SP\_BINDRULE estão sendo descontinuados e não devem ser utilizados.

## 8.2.4. Criação de tipo de dados graficamente

Para a criação de um tipo de dados graficamente, expanda o banco de dados no Object Explorer. Expanda **Programmability** e **Types**:

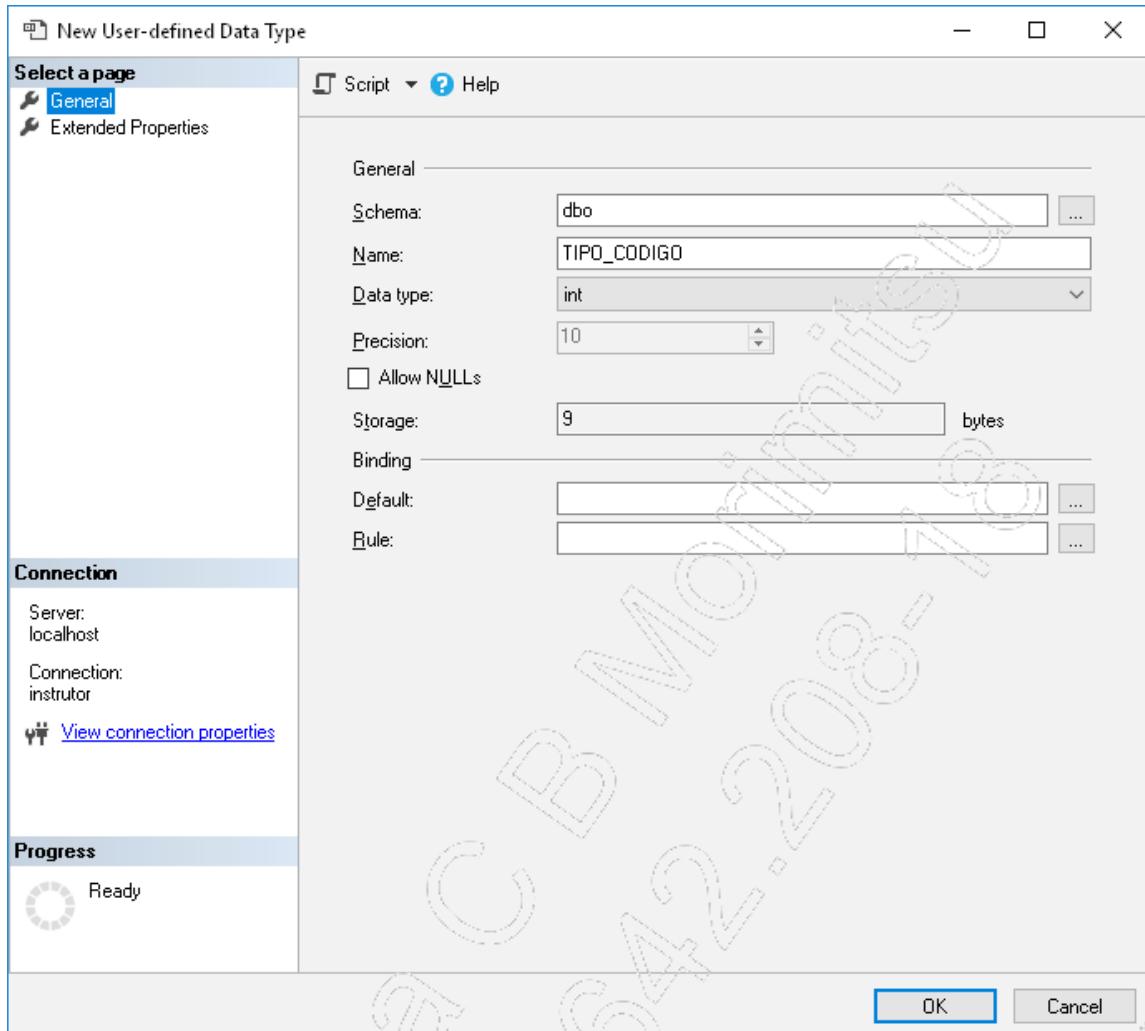


Clicando com o botão direito sobre **Types**, selecione **User-Defined Data Type**:



# Opções de definição de tabelas

A tela para a criação do tipo de dados é apresentada:



Para criar o tipo, informe as opções e clique em **OK**.

Uma opção também é o uso do botão **SCRIPT**, que gerará um código T-SQL conforme as opções definidas.

## 8.2.5. Trabalhando com UDDT

Nos passos a seguir, com base nos conceitos apresentados previamente, demonstraremos como utilizar tipos de dados definidos pelo usuário (UDDT):

1. Crie um banco de dados de teste;

```
CREATE DATABASE DB_UDDT
```

2. Coloque-o em uso;

```
USE DB_UDDT
```

3. Crie os UDDT;

```
CREATE TYPE TYPE_NOME_PESSOA FROM VARCHAR(40) NOT NULL;
CREATE TYPE TYPE_NOME_EMPRESA FROM VARCHAR(60) NOT NULL;
CREATE TYPE TYPE_PRECO   FROM NUMERIC(12,2) NOT NULL;
CREATE TYPE TYPE_SN     FROM CHAR(1) NULL;
CREATE TYPE TYPE_DATA_MOVTO  FROM DATETIME NULL;
```

4. Exiba os UDDT que acabamos de criar;

```
SELECT * FROM SYSTYPES WHERE UID = 1
```

name	xtype	status	xusertype	length	xprec	xscale	tdefault	domain	uid	reserved
TYPE_NOME_PESSOA	167	1	257	40	0	0	0	0	1	0
TYPE_NOME_EMPRESA	167	1	258	60	0	0	0	0	1	0
TYPE_PRECO	108	1	259	9	12	2	0	0	1	0
TYPE_SN	175	0	260	1	0	0	0	0	1	0
TYPE_DATA_MOVTO	61	0	261	8	23	3	0	0	1	0

5. Utilização do tipo de dados na criação de tabelas;

```
CREATE TABLE PRODUTOS
(
    COD_PROD           INT IDENTITY,
    DESCRICAO          VARCHAR(80),
    PRECO_CUSTO        TYPE_PRECO,
    PRECO_VENDA        TYPE_PRECO,
    DATA_CADASTRO     TYPE_DATA_MOVTO,
    SN_ATIVO           TYPE_SN,
    CONSTRAINT PK_PRODUTOS PRIMARY KEY (COD_PROD)
)
```

6. Também é possível utilizar em variáveis, que serão tratadas nos capítulos posteriores.

```
DECLARE @PRECO TYPE_PRECO

SET @PRECO = 10

PRINT @PRECO
```

## 8.2.6. Tipo tabular

É possível criar um tipo tabular semelhante a uma tabela regular, porém sem a persistência em disco e totalmente em memória.

Esse recurso aumenta consideravelmente a performance, porém deve ser utilizado com cuidado para não sobrecarregar o ambiente.

Verifique o exemplo adiante de criação de um tipo tabular:

- Criação do tipo tabular:

```
CREATE TYPE TYPE_CLIENTE AS TABLE
(
    ID          INT,
    NOME        VARCHAR(50),
    ESTADO      CHAR(2)
)
```

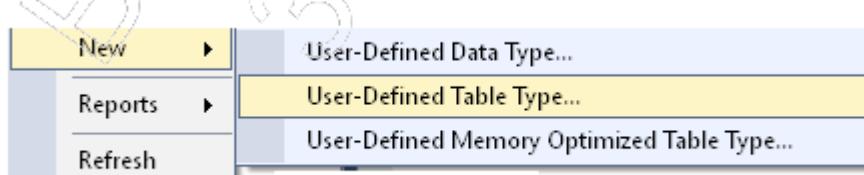
- Utilização de um tipo tabular:

```
DECLARE @TC TYPE_CLIENTE

INSERT INTO @TC VALUES
(1 , 'A' , 'SP'),
(2 , 'B' , 'RJ'),
(3 , 'C' , 'SP')

SELECT * FROM @TC
```

Para criação gráfica do tipo tabular, expanda o banco de dados e **Programmability**. Com o botão direito do mouse, selecione **User-Defined Table Type**:



O SQL disponibilizará uma estrutura padrão com o comando para a criação do tipo tabular.

```
-- =====
-- Create User-defined Table Type
-- =====
USE <database_name,sysname>,AdventureWorks
GO

-- Create the data type
CREATE TYPE <schema_name,sysname>,dbo>.<type_name,sysname>,TVP>
AS TABLE
(
    <columns_in_primary_key, , c1> <column1_datatype, ,
int> <column1_nullability,, NOT NULL>,
    <column2_name, sysname, c2> <column2_datatype, ,
char(10)> <column2_nullability,, NULL>,
    <column3_name, sysname, c3> <column3_datatype, , date-
time> <column3_nullability,, NULL>,
    PRIMARY KEY (<columns_in_primary_key, , c1>)
)
GO
```

## 8.2.6.1. Tipo tabular otimizado em memória

Esse tipo de dados implementa uma melhora na utilização e desempenho em memória.

As vantagens do tipo de dados otimizado em memória são:

- O armazenamento da variável é realizado somente em memória;
- Algoritmo otimizado;
- Não utiliza o TEMPDB.

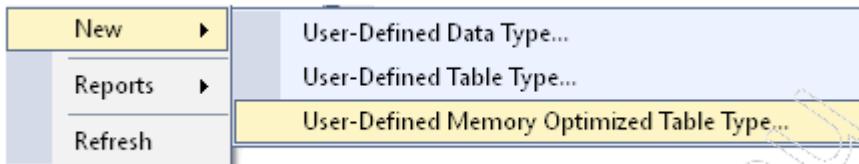
Na criação, é necessária a elaboração de um índice HASH ou PRIMARY KEY NONCLUSTERED.

Também é necessário que o banco possua um FILEGROUP otimizado em memória (FILEGROUP será apresentado no módulo III).

```
CREATE TYPE TYPE_CLIENTE AS TABLE
(
    ID           INT,
    NOME         VARCHAR(50),
    ESTADO       CHAR(2),
    PRIMARY KEY NONCLUSTERED (ID)
)
WITH (MEMORY_OPTIMIZED = ON)
```

# Opções de definição de tabelas

Também é possível criar um tipo com o auxílio do SSMS. Expanda o banco de dados, **Programmability**, **Type** e, com o botão direito do mouse, selecione **User Defined Memory Optimized Type**.



```
=====
-- Create User-defined Memory Optimized Table Type Template
-- Use the Specify Values for Template Parameters command
-- (Ctrl-Shift-M) to fill in the parameter values below.
-- This template creates a memory optimized table type and
-- indexes on the memory optimized table type.
-- The database must have a MEMORY_OPTIMIZED_DATA filegroup
-- before the memory optimized table type can be created.
=====

USE <database, sysname, AdventureWorks>
GO

--Drop table if it already exists.
IF TYPE_ID('<schema_name, sysname, dbo>.<type_
name,sysname,sample_memoryoptimizedtabletype>') IS NOT NULL
    DROP TYPE <schema_name, sysname, dbo>.<type_
name,sysname,sample_memoryoptimizedtabletype>
GO

CREATE TYPE <schema_name, sysname, dbo>.<type_
name,sysname,sample_memoryoptimizedtabletype> AS TABLE
(
    <column_in_primary_key, sysname, c1> <column1_datatype,
, int> <column1_nullability, , NOT NULL>,
    <column2_name, sysname, c2> <column2_datatype, , float>
<column2_nullability, , NOT NULL>,
    <column3_name, sysname, c3> <column3_datatype, ,
decimal(10,2)> <column3_nullability, , NOT NULL> INDEX
<index3_name, sysname, index_sample_memoryoptimizedtabletype_
c3> NONCLUSTERED (<column3_name, sysname, c3>),
    PRIMARY KEY NONCLUSTERED (<column1_name, sysname, c1>),
    -- See SQL Server Books Online for guidelines on
    determining appropriate bucket count for the index
    INDEX <index2_name, sysname, hash_index_sample_
memoryoptimizedtabletype_c2> HASH (<column2_name, sysname,
c2>) WITH (BUCKET_COUNT = <sample_bucket_count, int, 131072>)
) WITH (MEMORY_OPTIMIZED = ON)
GO
```

## 8.3. Tabelas de sistema

Os dados que determinam a configuração de um servidor e todas as tabelas que ele possui são armazenados no SQL Server. Eles ficam armazenados em um conjunto específico de tabelas, que são as tabelas de sistema. Uma tabela de sistema não pode ter seus dados alterados diretamente.

Algumas dessas tabelas armazenam informações de sistema utilizadas em cada banco de dados criado pelo usuário. Entre elas, estão as tabelas **SYSTYPES** e **SYSOBJECTS**.

### 8.3.1. Tabela SYSTYPES

A tabela **SYSTYPES** é responsável por armazenar tanto os tipos de dados fornecidos pelo sistema quanto os definidos pelo usuário. Como já vimos, cada tipo de dados criado por um usuário em um banco de dados específico é adicionado à sua tabela **SYSTYPES**. Tipos de dados do próprio SQL possuem a coluna **UID** preenchida com 4; já os tipos de dados de usuário possuem **UID** igual a 1.

The screenshot shows a SQL Server Management Studio window with the following details:

- Query Editor:** The query `SELECT * FROM SYSTYPES` is entered.
- Results Grid:** The results show the following data:

name	xtype	status	xusertype	length	xprec	xscale	tdefault	domain	uid	reserved	ci
33 xml	241	0	241	-1	0	0	0	0	4	0	N
34 sysname	231	1	256	256	0	0	0	0	4	0	S
35 TYPE_NOME_PESSOA	167	1	257	40	0	0	0	0	1	0	S
36 TYPE_NOME_EMPRESA	167	1	258	60	0	0	0	0	1	0	S
37 TYPE_PRECO	108	1	259	9	12	2	0	0	1	0	N
38 TYPE_SN	175	0	260	1	0	0	0	0	1	0	S
39 TYPE_DATA_MOVTO	61	0	261	8	23	3	0	0	1	0	N

### 8.3.2. Tabela SYSOBJECTS

A tabela **SYSOBJECTS** é responsável por armazenar os objetos existentes no banco de dados (de usuário e de sistema) no escopo do esquema em um banco de dados. Sendo assim, cada objeto rule, default, table, constraints, entre outros criados em um banco de dados, é adicionado à tabela **SYSOBJECTS**.

```
SELECT * FROM SYSOBJECTS
```

	name	id	xtype	uid	info	status	base_schema_ver	replinfo	parent_obj
42	sysasymkeys	95	S	4	0	0	0	0	0
43	syssqlguides	96	S	4	0	0	0	0	0
44	sysbinsubobjs	97	S	4	0	0	0	0	0
45	syssoftobjrefs	98	S	4	0	0	0	0	0
46	TabelaCar	5575058	U	1	0	0	0	0	0
47	PK_TabelaCar	21575115	PK	1	0	0	0	0	5575058
48	PROD_FORN	37575172	U	1	0	0	0	0	0
49	PK_PROD_FORN	53575229	PK	1	0	0	0	0	37575172
50	CLIENTES	69575286	U	1	0	0	0	0	0

Consulta ex... SOMA5\SQLEXPRESS2008 (10.0 ... SOMA5\CARLOS MAGNO SOU... PEDIDOS 00:00:00 | 113 linhas

Para o exemplo anterior, consideremos as seguintes informações:

- **A** - Número identificador do objeto;
- **B** - Tabelas de sistema;
- **C** - Tabelas de usuário.

### 8.3.3. Tabela SYSCOMMENTS

Esta tabela é composta por entradas para cada um dos seguintes itens de um banco de dados: CONSTRAINT, DEFAULT, CONSTRAINT CHECK, STORED PROCEDURES, VIEWS, regras, padrões e TRIGGERS.

Vejamos, a seguir, quais são as colunas da tabela **SYSCOMMENTS**, seus tipos de dados e suas finalidades:

- **COLID**: Essa coluna é formada por dados do tipo **SMALLINT** e possui o número sequencial da linha para as definições de objetos que ultrapassam a quantidade de quatro mil caracteres;
- **COMPRESSED**: Essa coluna é formada por dados do tipo bit e tem a função de determinar que uma procedure está comprimida, retornando sempre o valor zero;

- **CTEXT:** Essa coluna é formada por dados **VARBINARY(8000)** e possui os bytes referentes ao comando de definição SQL;
- **ID:** Essa coluna é formada por dados do tipo **INT** e possui o ID do objeto ao qual se aplica o texto;
- **ENCRYPTED:** Essa coluna é formada por dados do tipo bit e sua função é definir se a procedure está criptografada ou não, sendo que o valor **0** indica que não está criptografada e o valor **1** indica que está. Vale destacar que é possível não apenas criptografar, mas também ocultar as definições de STORED PROCEDURES. Para tanto, basta utilizar o comando **CREATE PROCEDURE** em conjunto com a palavra-chave **ENCRYPTION**;
- **LANGUAGE:** Essa coluna é formada por dados do tipo **SMALLINT** e é destinada apenas ao uso interno;
- **NUMBER:** Essa coluna é formada por dados do tipo **SMALLINT** e possui o número que se encontra dentro de um agrupamento da procedure caso ela esteja agrupada. Quando o valor é igual a zero significa que as entradas não são procedures;
- **STATUS:** Essa coluna é formada por dados do tipo **SMALLINT** e é destinada apenas para uso interno;
- **TEXT:** Essa coluna é formada por dados do tipo **NVARCHAR(4000)** e tem a função de determinar o texto de um comando de definição SQL;
- **TEXTPTYPE:** Essa coluna é formada por dados **SMALLINT** e apresenta um dos seguintes valores: **0** para os comentários feitos pelo usuário; **1** para os comentários do sistema; e **4** para os comentários criptografados.

Vale destacar que, a fim de realizar a leitura de uma tabela **SYSCOMMENTS**, basta utilizar o comando **SELECT**, conforme demonstrado a seguir:

```
SELECT TEXT FROM SYSCOMMENTS
```

	id	number	colid	status	ctext	textrtype	language	text
1	2105058535	1	1	0	0x43... CREATE PROCEDURE SP_INTEIROS @N INT = 100 AS BEGIN DECLAR...	2	0	
2	2121058592	1	1	0	0x43... CREATE PROCEDURE DBO.SP_PARES @N INT = 100 AS BEGIN DECL...	2	0	

## 8.4. Sequências

Desde a versão do SQL Server 2012, é possível criarmos um objeto chamado sequência (SEQUENCE). Esse objeto visa retornar um número sequencial, que pode ser utilizado em qualquer aplicação, inclusive como base para a criação de chaves primárias em registros.

A forma como a sequência é controlada depende de aplicação. Nada impede que duas ou mais aplicações utilizem a mesma sequência para obter valores para inserção de dados em uma única tabela, ou mesmo para utilização em diferentes tabelas. Apenas a aplicação pode impedir ou vincular o uso de uma sequência.

## Opções de definição de tabelas

Não é recomendada a utilização de sequências para valores que não possam conter “furos”, pois, entre o momento em que se captura o valor da sequência e o momento em que esta é efetivamente utilizada, pode ocorrer algum tipo de falha ou mesmo a aplicação sofrer um comando rollback.

Vejamos a sintaxe para criação de sequências:

```
CREATE SEQUENCE [schema_name .] sequence_name  
    [ AS [ built_in_integer_type | user-defined_integer_type ]  
    ]  
    [ START WITH <constant> ]  
    [ INCREMENT BY <constant> ]  
    [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]  
    [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]  
    [ CYCLE | { NO CYCLE } ]  
    [ { CACHE [ <constant> ] } | { NO CACHE } ]  
    [ ; ]
```

Pode-se definir o valor inicial da sequência, como ela será incrementada, se terá um limite ou não e se, ao atingir um limite, a sequência deverá ser reciclada ou não reciclada. Ainda para melhorar a performance da sequência, pode-se definir que ela seja gerenciada em memória (CACHE).

Exemplo de criação de sequências:

```
CREATE SEQUENCE SEQ_ALUNO;
```

No exemplo anterior, a sequência iniciará em -9223372036854775808, será incrementada de 1 em 1, o limite será 9223372036854775807 e não sofrerá cache em memória:

```
CREATE SEQUENCE SEQ_ALUNO  
START WITH 1000  
INCREMENT BY 10  
MINVALUE 10  
MAXVALUE 10000  
CYCLE CACHE 10;
```

No exemplo anterior, a sequência iniciará no número 1000, será incrementada de 10 em 10 e terminará em 10000. Ao final, será retornada ao valor original 10 (MINVALUE).

Para utilizarmos o valor de uma sequência em uma inserção, devemos usar a cláusula **NEXT VALUE FOR** e o nome da sequência.

```
CREATE TABLE T_ALUNO
(COD_ALUNO          INT,
NOM_ALUNO           VARCHAR(50) )
GO

INSERT INTO T_ALUNO (COD_ALUNO, NOM_ALUNO)
VALUES (NEXT VALUE FOR DBO.SEQ_ALUNO, 'TESTE');
```

Para encontrarmos as sequências criadas, podemos utilizar o seguinte comando:

```
SELECT * FROM SYS.SEQUENCES;
```

## 8.5. Sinônimos

Sinônimos são recursos que permitem substituir o nome de um objeto, por exemplo, uma tabela. Todos os objetos pertencem a um schema nomeado ou, caso não existam schemas criados, podemos utilizar o schema DBO. À medida que aumentam a complexidade do modelo e a quantidade de objetos, podemos usar o recurso de criação de sinônimos.

Vejamos a seguir a sintaxe para criação de sinônimos:

```
CREATE SYNONYM [ schema_name_1. ] synonym_name FOR <object>

<object> ::=
{
    [ server_name.[ database_name ] . [ schema_name_2 ].| da-
    tabase_name . [ schema_name_2 ].| schema_name_2. ] object_name
}
```

Criando um sinônimo para uma tabela:

```
CREATE TABLE TB_ALUNO
(
ID      INT,
ALUNO   VARCHAR(10),
SEXO    BIT,
DT_NASC DATETIME
)
GO
CREATE SYNONYM TAB_ALUNO FOR DBO.ALUNO;
GO

SELECT * FROM TAB_ALUNO;
```

# Opções de definição de tabelas

Acessando um sinônimo em uma função:

```

CREATE FUNCTION dbo.Fun_teste (@valor int)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
IF @valor % 12 <> 0
BEGIN
    SET @valor += 12 - (@valor % 12)
END
RETURN(@valor);
END;
GO

SELECT dbo.Fun_teste(10)

CREATE SYNONYM FUN_TESTE_EXEMPLO FOR DBO.FUN_TESTE;
GO
SELECT dbo.FUN_TESTE_EXEMPLO(10)

```

Podemos criar sinônimos para tabelas, visões, procedimentos, funções, sequências, entre outros objetos.

## 8.6. Trabalhando com objetos binários

O SQL permite a persistência de arquivos binários. Este tipo de recurso simplifica o gerenciamento de arquivos binários como: imagens, documentos, planilhas etc., porém consome recursos do repositório do SQL Server.

### 8.6.1. Campos binários

Campos binários permitem a inclusão de arquivos binários diretamente na tabela. Os tipos de dados binários são: **BINARY**, **VARBINARY** e **IMAGE**.

Sua vantagem é permitir o trabalho do objeto binário diretamente na tabela. Porém, por conter dados não estruturados, a tabela tende a consumir mais recursos de disco, aumentando o tempo na resposta das transações. Vejamos, a seguir, a utilização desse recurso:

- Criação da tabela

```
--Criação da tabela
CREATE TABLE TB_CLIENTE_DOCUMENTO(
ID_DOC INT IDENTITY PRIMARY KEY,
DESCRICAO VARCHAR(50),
DOCUMENTO VARBINARY(MAX) )

GO
```

- Inserindo arquivos com o recurso OPENROWSET

```
--Inserção de um arquivo binário
Insert Into TB_CLIENTE_DOCUMENTO(DESCRICAO, DOCUMENTO)
    Select 'Planilha Excel', BulkColumn
        from Openrowset (Bulk 'C:\DADOS\PESSOA.XLS', Single_Blob)
    as Image
```

- Consulta da tabela

```
SELECT * FROM TB_CLIENTE_DOCUMENTO
```

- Atualizando um campo binário

```
-- Atualização
UPDATE TB_CLIENTE_DOCUMENTO SET DOCUMENTO =
    (SELECT * from Openrowset(Bulk 'C:\DADOS\PESSOA.XLS',
Single_Blob) as Arq)
where ID_DOC = 1
```

- Devolvendo o arquivo para disco utilizando BCP

```
-- Opções avançadas e utilização de comandos shell
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'xp_cmdshell',1
GO
RECONFIGURE;
GO
```

```
-- Retrieve utilizando BCP
Declare @sql varchar(500)
SET @sql = 'BCP "SELECT DOCUMENTO FROM PEDIDOS.DBO.TB_CLIENTE_
DOCUMENTO where ID_DOC = 1" QUERYOUT C:\dados\PESSOA1.XLS
-SINSTRUTOR -T -N'
EXEC master.dbo.xp_CmdShell @sql
--Onde:
--S--> Nome do Servidor
--T--> Autenticação do Windows
```

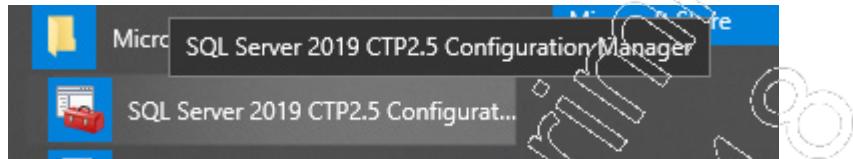
 Sempre dê preferência à programação com STORED PROCEDURES ou através do SQL Server Integration Services – SSIS, que permite uma programação mais estruturada.

## 8.7. FILETABLE

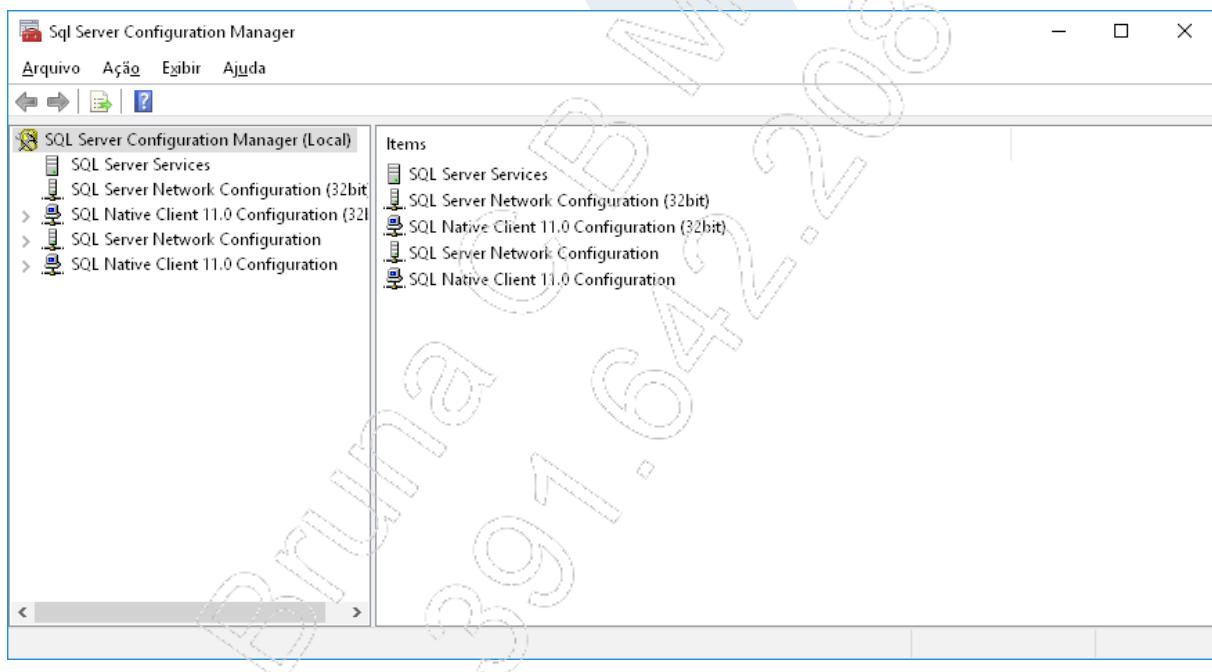
**FILETABLE** é um recurso que permite carregar objetos binários numa área do sistema do Windows e ser gerenciado pelo SQL Server. A vantagem deste recurso é a separação dos dados estruturados e dos arquivos binários, outra vantagem é a cópia diretamente através do sistema de arquivos.

Para utilizar este recurso é necessário que o FILESTREAM esteja habilitado no nível do servidor. Vejamos os passos a seguir:

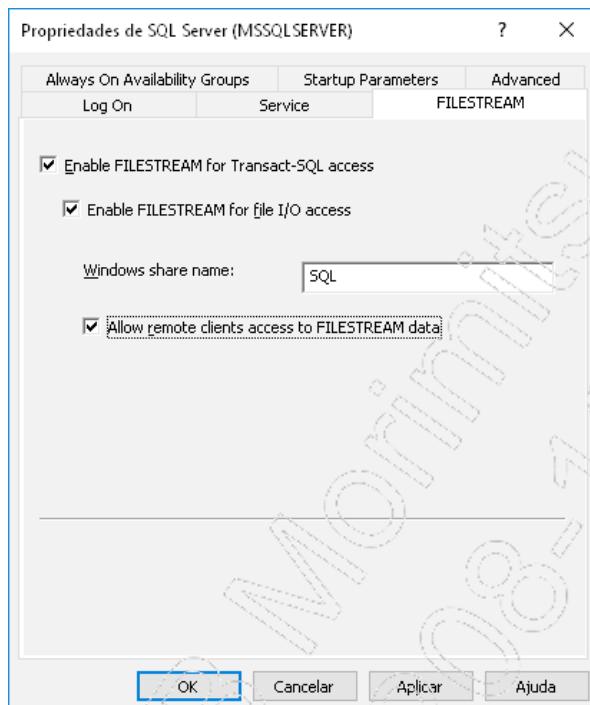
1. Acesse Microsoft SQL Server 2019 CTP2.5 / Configuration Manager;



2. Selecione SQL Server Configuration Manager;



3. Selecione o serviço do SQL Server e abra a guia FILESTREAM;



4. Habilite as opções:

- **Enable FILESTREAM for Transact-SQL access;**
- **Enable FILESTREAM for file I/O access;**
- Informe o nome do compartilhamento do FILESTREAM;
- **Allow remote clients access to FILESTREAM data.**

5. Execute o comando para habilitar o FILESTREAM;

```
-- Enable FileStream  
EXEC sp_configure filestream_access_level, 2  
RECONFIGURE  
GO
```

# Opções de definição de tabelas

6. A criação do banco deve possuir um FILEGROUP com acesso para o FILESTREAM;

```
-- Create Database
CREATE DATABASE Banco_Filestream
ON PRIMARY
(NAME = FG_Filestream_PRIMARY,
FILENAME = 'C:\DADOS\Filestream_DATA.mdf'),
FILEGROUP FG_Filestream_FS CONTAINS FILESTREAM
(NAME = Filestream_ARQ,
FILENAME='C:\DADOS\Filestream_ARQ')
LOG ON
(NAME = Filestream_log,
FILENAME = 'C:\DADOS\Filestream_log.ldf')
WITH FILESTREAM (NON_TRANSACTED_ACCESS = FULL,
DIRECTORY_NAME = N'Filestream_ARQ');
GO
```

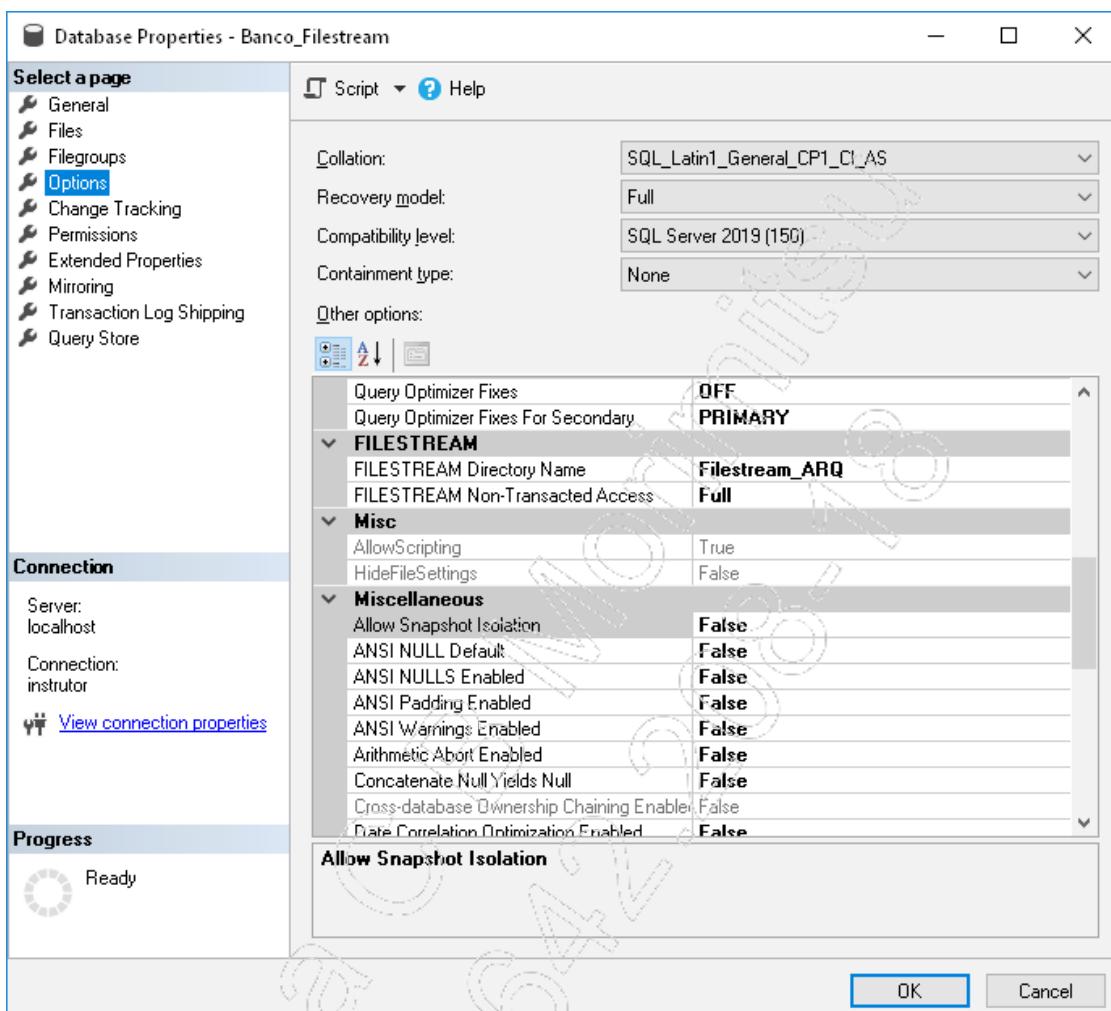
7. No exemplo adiante, o banco de dados já foi criado e será adicionado ao FILEGROUP com o comando ALTER DATABASE:

```
USE MASTER

DROP DATABASE BANCO_FILESTREAM
GO
--Criando o banco de dados
CREATE DATABASE Banco_Filestream
ON PRIMARY
(NAME = FG_Filestream_PRIMARY,
FILENAME = 'C:\DADOS\Filestream_DATA.mdf')
LOG ON
(NAME = Filestream_log,
FILENAME = 'C:\DADOS\Filestream_log.ldf')
GO
--Adicionando FILEGROUP
ALTER DATABASE Banco_Filestream ADD FILEGROUP FG_Filestream_FS
CONTAINS FILESTREAM

--Adicionando arquivos
ALTER DATABASE Banco_Filestream ADD FILE
(NAME = Filestream_ARQ,FILENAME='C:\DADOS\Filestream_ARQ')
TO FILEGROUP FG_Filestream_FS
```

8. Antes da criação das tabelas, verifique, em **OPTIONS**, se o item **FILESTREAM Directory Name** está com valor (neste caso, foi atribuído o valor **Filestream\_ARQ**):



- **Criação de tabelas FILETABLE**

A seguir, um exemplo de criação de tabela FILETABLE:

```
USE BANCO_FILESTREAM
GO
CREATE TABLE FT_Documento AS FileTable
--OU
CREATE TABLE FT_Documento AS FileTable
WITH (
    FileTable_Directory = 'Filestream_ARQ',
    FileTable_Collate_Filename = database_default
);
GO
```

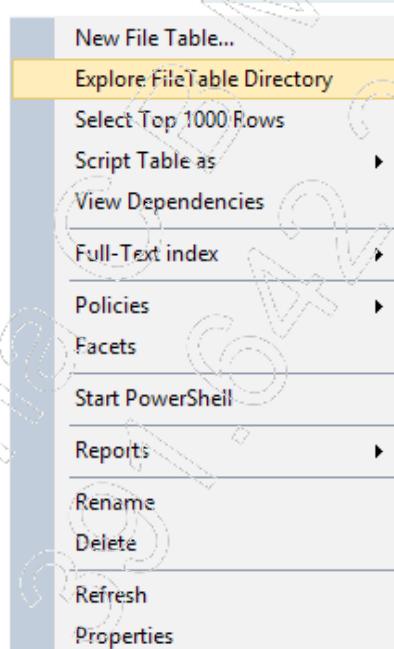
# Opções de definição de tabelas

- **Visualização e manutenção do modo gráfico**

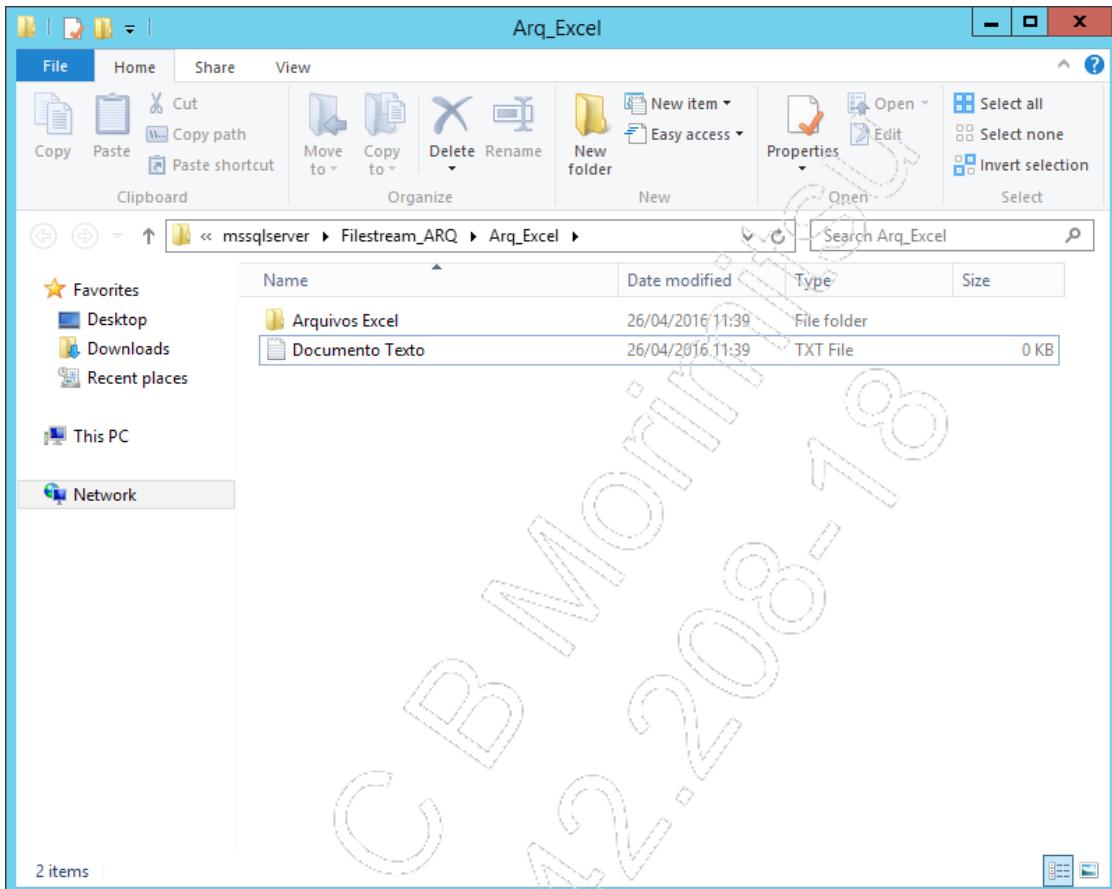
1. Abra o SSMS e, no Object Explorer, expanda o banco de dados, **Tables** e **FileTables**:



2. Com o botão direito, clique sobre a tabela e selecione a opção **Explore FileTable Directory**:



Será apresentada uma tela como a do Windows Explorer, em que é possível visualizar, copiar, mover e excluir arquivos:



- **Inserindo arquivos através de comando TSQL**

```
--Inserção de um arquivo binário
```

```
Insert Into FT_Documento (name, file_stream)
Select 'Planiha Excel', BulkColumn
from Openrowset (Bulk 'C:\DADOS\PESSOA.XLS', Single_Blob)
as Image
```

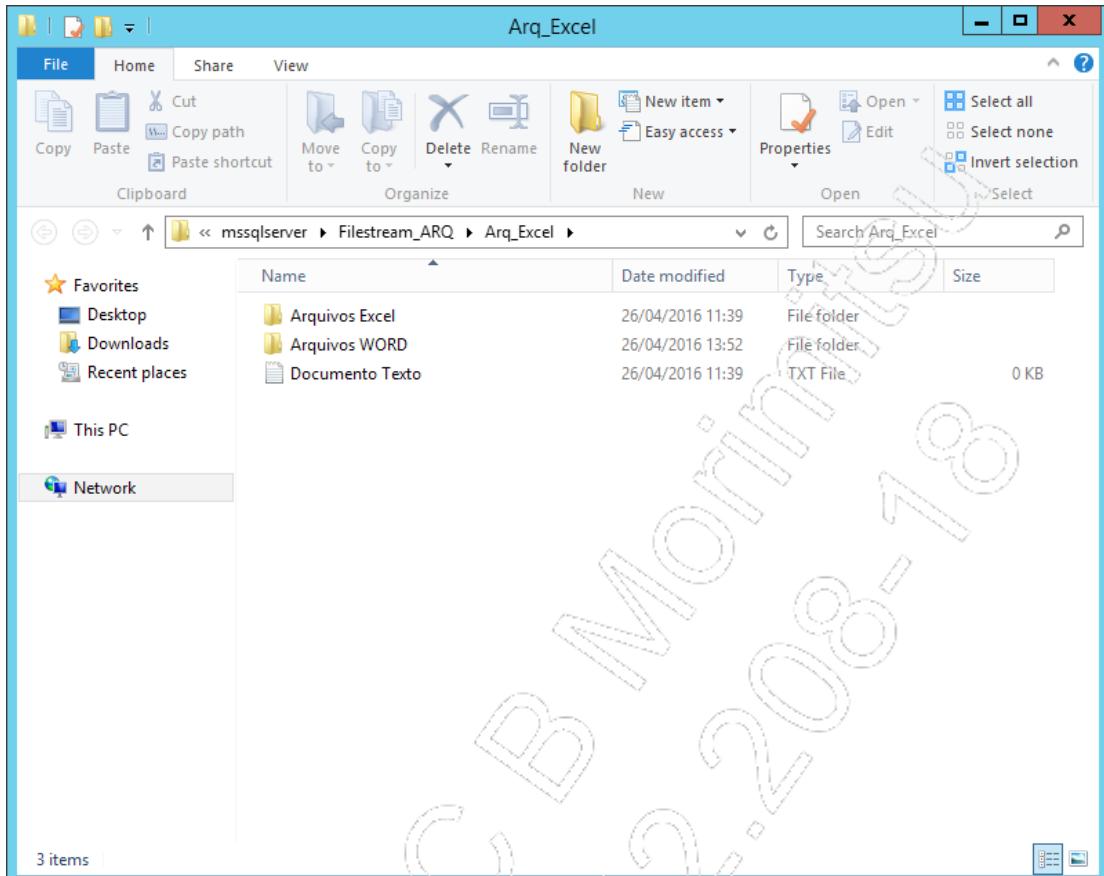
- **Inserindo uma subpasta**

É possível inserir subpastas para melhorar a visualização e organização dos arquivos:

```
INSERT INTO FT_Documento (name,is_directory,is_archive)
values
('Arquivos WORD' , 1 ,0)
```

# Opções de definição de tabelas

Vejamos o resultado após execução do comando:



- Consultando uma tabela FILETABLE**

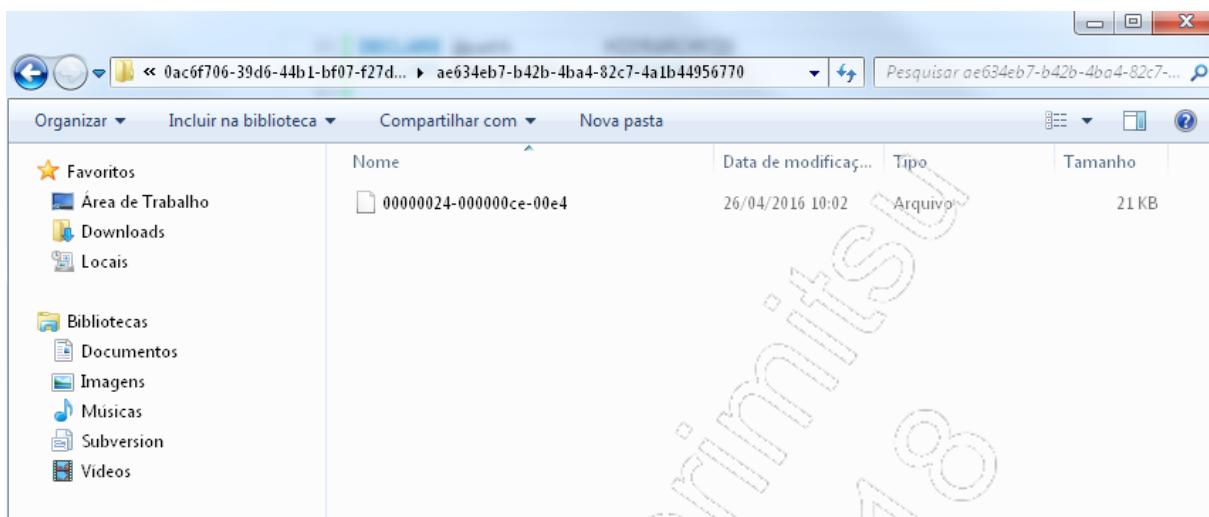
```
SELECT * FROM FT_Documento;
```

O resultado apresenta as informações dos arquivos inseridos na tabela:

stream_id	file_stream	name	path_locator	parent_pa
4E2D54A3-BC0B-E611-80C0-00155D12C902	NULL	Arquivos Excel	0xFCAE1116E9C1316FDD101052ADD232F9A89E8D42A0	NULL
502D54A3-BC0B-E611-80C0-00155D12C902		Documento Texto.txt	0xFEB72568494B1BEFD61170E450D81AFA94602C2260	NULL
8E13E93B-CF0B-E611-80C0-00155D12C902	NULL	Arquivos WORD	0xFE72A7422398B38FD391513251CAB6FAD0DC174720	NULL

# SQL 2019 - Módulo II

Fisicamente, o arquivo ficará na pasta do sistema operacional com a estrutura e organização do SQL Server:



O comando **FileTableRootPath** permite a visualização do caminho dos arquivos da tabela:

```
SELECT FileTableRootPath('FT_Documento') [Caminho]
```

O resultado da consulta apresenta a localização dos arquivos:

```
Caminho  
\LOCALHOST\SQL\Filestream_ARQ\Filestream_ARQ
```

A próxima consulta apresenta as informações básicas dos arquivos e pastas:

```
SELECT Tab.Name as Nome,  
IIF(Tab.is_directory=1,'Diretório','Arquivo') as Tipo,  
Tab.file_type as Extensao,  
Tab.cached_file_size/1024.0 as Tamanho_KB,  
Tab.creation_time as Data_Criacao,  
Tab.file_stream.GetFileNamespacePath(1,0) as Caminho,  
ISNULL(Doc.file_stream.GetFileNamespacePath(1,0),'Root  
Directory') [Parent Path]  
FROM FT_Documento as Tab  
LEFT JOIN FT_Documento as Doc  
ON Tab.path_locator.GetAncestor(1) = Doc.path_locator
```

Nome	Tipo	Extensao	Tamanho_KB	Data_Criacao	Caminho
Arquivos Excel	Diretório	NULL	NULL	2016-04-26 11:39:11.3963849 -03:00	\LOCALHOST\MSSQLSERVER\Filestream_ARQ\Arq_Excel\...
Documento Texto.txt	Arquivo	txt	0.000000	2016-04-26 11:39:16.5839513 -03:00	\LOCALHOST\MSSQLSERVER\Filestream_ARQ\Arq_Excel\...
Arquivos WORD	Diretório	NULL	NULL	2016-04-26 13:52:18.2491387 -03:00	\LOCALHOST\MSSQLSERVER\Filestream_ARQ\Arq_Excel\...

- Atualizando arquivos

```
UPDATE FT_Documento SET file_stream =
    (SELECT * from Openrowset(Bulk 'C:\DADOS\ArqTXT.txt',
Single_Blob) as Arq)
where name = 'Arquivos Excel'
```

- Excluindo arquivos

```
DELETE FT_Documento
WHERE NAME = 'Arquivos Excel'
```

## 8.8. Colunas computadas

Na criação de uma tabela, é possível a utilização de colunas computadas. O recurso de criar uma coluna calculada simplifica a construção de consultas, com cálculos, que são frequentemente utilizadas.

Estas colunas são virtuais e não são armazenadas no banco, porém, é possível realizar esta persistência com a cláusula **PERSISTED**. Exemplo:

A área de RH solicita frequentemente solicita a consulta dos empregados com data de nascimento e idade. Vejamos:

```
USE DB_ECOMMERCE
GO

SELECT
NOME, DATA_NASCIMENTO,
FLOOR(CAST(GETDATE()-DATA_NASCIMENTO AS FLOAT)/365.25) AS
IDADE
FROM TB_EMPREGADO
```

Criação da coluna calculada:

```
ALTER TABLE TB_EMPREGADO
ADD IDADE AS
FLOOR(CAST(GETDATE()-DATA_NASCIMENTO AS FLOAT)/365.25)
```

A mesma consulta utilizando a coluna calculada **IDADE**:

```
SELECT
NOME, DATA_NASCIMENTO, IDADE
FROM TB_EMPREGADO
```



### Pontos principais

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

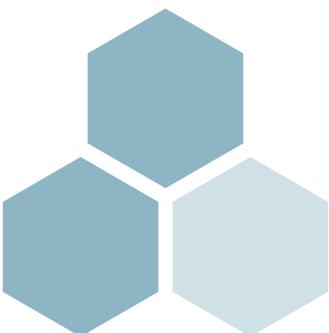
- Os **tipos de dados** determinam os tipos de valores que podem ser inseridos em uma coluna. Os **tipos de dados nativos** são aqueles fornecidos pelo próprio SQL Server. Conhecido como **UDDT**, o **tipo de dados definido pelo usuário** também pode ser considerado um sinônimo de um tipo já disponível;
- Os dados que determinam a configuração de um servidor e todas as tabelas que ele possui são armazenados no SQL Server. Eles ficam armazenados nas tabelas de sistema;
- **Sequências** são objetos sequenciadores que disponibilizam um número sequencial único. Este recurso aumenta as possibilidades da programação e criação de chaves primárias;
- **Sinônimos** são recursos que permitem substituir nomes de objetos;
- É possível o armazenamento de objetos binários dentro do SQL. Para isso, podemos utilizar um campo binário ou o recurso de **FILETABLE**;
- **Colunas computadas** permitem a criação de um campo calculado. Utilize esse recurso quando for realizar cálculos frequentes nas consultas.



8

# Opções de definição de tabelas

Teste seus conhecimentos



**1. Ao criarmos uma tabela de nome TB\_Funcionario, foi definida a necessidade de se criar um campo para salário. Qual tipo de dados não é recomendado?**

- a) MONEY
- b) DECIMAL(10,2)
- c) VARCHAR(10)
- d) NUMERIC(10,2)
- e) FLOAT

**2. Verifique o comando de criação de tabela a seguir:**

```
CREATE TABLE tb_Aluno (
    Id_ALUNO           INT   PRIMARY KEY,
    Nome               CHAR(100),
    DT_NASC DATETIME,
    Fone               CHAR(14) )
```

**O que pode ser melhorado na definição dos tipos de dados?**

- a) Não é recomendada a utilização de PRIMARY KEY.
- b) A tabela está correta e não deve ser alterada.
- c) Falta a opção IDENTITY na coluna ID\_Aluno.
- d) Alterar o campo Fone para NUMERIC(14).
- e) Como o campo nome possui um tamanho variável, o recomendado é a utilização de VARCHAR(100)..

**3. O que é um FILETABLE?**

- a) É uma tabela de arquivos.
- b) É uma tabela para objetos binários que utiliza um FILESTREAM.
- c) É uma tabela que armazena objetos binários e dados comuns.
- d) É uma tabela que armazena objetos binários dos bancos de sistema.
- e) É uma tabela como uma tabela regular, porém usa um FILESTREAM.

## Opções de definição de tabelas

### 4. Com relação a SEQUENCE, qual afirmação está errada?

- a) É um objeto que retorna um valor sequencial.
- b) Este objeto não está vinculado a apenas uma tabela.
- c) A sequência pode apresentar “furos”.
- d) Pode ser utilizado em campos que são chaves primárias.
- e) Este recurso não é funcional, pois a tabela já possui um IDENTITY.

### 5. O que é um Sinônimo?

- a) Recurso que substitui somente o nome de uma tabela.
- b) Recurso que substitui somente o nome de uma procedure.
- c) Objeto que sequencia uma tabela.
- d) Objeto que cria uma tabela com nome alternativo.
- e) Recurso que substitui o nome de um objeto.





8

# Opções de definição de tabelas



Mãos à obra!



## Laboratório 1

### A – Criando e associando UDDTs, regras de validação e objetos DEFAULT

1. Coloque o banco **DB\_ECOMMERCE** em uso;
2. Crie os tipos de dados definidos pelo usuário (UDDT), conforme especificação a seguir:

TIPO_CODIGO	INT	NOT NULL
TIPO_ENDERECO	VARCHAR(60)	NULL
TIPO_FONE	CHAR(14)	NULL
TIPO_SEXO	CHAR(1)	NOT NULL
TIPO_ALIQUOTA	NUMERIC(4,2)	NULL
TIPO_PRAZO	INT	NOT NULL

3. Exiba os **UDDT** recém-criados;
4. Crie a tabela **PESSOAS**, seguindo o modelo adiante:

COD_PESSOA	TIPO_CODIGO	autonumeração	chave primária
NOME	VARCHAR(30)		
ENDERECO	TIPO_ENDERECO		
SEXO	TIPO_SEXO		

5. Crie uma **CONSTRAINT CHECK** para o campo **SEXO**, permitindo apenas valores **F** e **M**;
6. Crie uma **CONSTRAINT DEFAULT** para o campo **SEXO** com valor **M**;
7. Crie a tabela **CONTAS**, seguindo o modelo adiante:

COD_CONTA	TIPO_CODIGO	autonumeração	chave primária
VALOR	NUMERIC(10,2)		
QTD_PARCELAS	TIPO_PRAZO		
PORC_MULTA	TIPO_ALIQUOTA		

8. Crie as **CONSTRAINTS CHECK** para a tabela de **CONTAS**:
  - Valor maior que zero;
  - Quantidade de parcelas maior ou igual a 1;
  - Percentual de multa maior ou igual a zero.
9. Crie a **CONSTRAINT DEFAULT** com valor **1** para o campo **Quantidade de parcelas**;

# Opções de definição de tabelas

10. Crie a tabela **FUNCIONARIO**, seguindo o modelo adiante:

FUNCIONARIO		
COD_FUNC	TIPO_CODIGO	chave primária
NOME	VARCHAR(30)	
ENDERECO	VARCHAR(80)	
SEXO	TIPO_SEXO	

11. Crie um sinônimo de nome **tb\_Funcionario** para a tabela **FUNCIONARIO**;

12. Crie uma SEQUENCE de nome **SQ FUNCIONARIO**, que inicie em 100, com incremento 2;

13. Crie um tipo de dados tabular com as informações:

- Nome: TIPO\_FORNECEDOR
- Campos: ID\_FORNECEDOR  
Nome do fornecedor  
Cidade  
Estado

## Laboratório 2

### A – Trabalhando com objetos binários

1. Coloque o banco **BANCO\_FILESTREAM** em uso;

2. Crie a tabela **TB\_DOCUMENTO** com as seguintes características:

- **ID\_DOCUMENTO**: Inteiro, autonumerável e chave primária;
- **Descrição do Documento**: Texto livre com até 100 caracteres;
- **Data do Cadastro**: Deve possuir valor padrão (Data e Hora do servidor);
- **Documento**: Campo binário.

3. Insira dois documentos na tabela **TB\_DOCUMENTO**;

4. Consulte a tabela **TB\_DOCUMENTO**.

### B – Inserindo e visualizando arquivos

1. Crie uma tabela **FILETABLE** de nome **FT\_Documento**;

2. Insira dois documentos nessa tabela;

3. Visualize os documentos de forma gráfica;

4. Visualize os documentos com comando TSQL.



### Laboratório 3

#### A – Trabalhando com colunas computadas

1. Coloque o banco **DB\_ECOMMERCE** em uso;
2. Crie a tabela **TB\_FUNC\_IDADE** com os seguintes campos:

<b>Id_funcionario</b>	inteiro, autonumerável e chave primária
<b>Nome do funcionário</b>	alfanumérico com 50 caracteres
<b>Data de Nascimento</b>	Campo data
<b>Idade</b>	Campo calculado

3. Insira os dados da tabela de empregados para a tabela **TB\_FUNC\_IDADE**;
4. Consulte as informações e verifique o campo calculado;
5. Adicione o campo **VLR\_ITEM** na tabela **TB\_ITENSPEDIDO**, com o cálculo adiante:

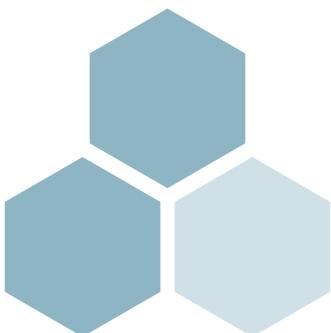
**PR\_UNITARIO \* QUANTIDADE \* (1 - DESCONTO /100)**

6. Faça uma consulta na tabela e verifique o resultado.

# 9

## Inserção de dados

- Constantes;
- Inserindo dados;
- Cláusula TOP;
- OUTPUT;
- Funções para campos autonomeáveis.



### 9.1. 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	$2.53 \times 10^4$	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.

## 9.2. 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:

- Criação da tabela:

```
CREATE TABLE TB_ALUNO  
( COD_ALUNO INT IDENTITY PRIMARY KEY,  
  NOME VARCHAR(30),  
  DATA_NASCIMENTO DATETIME,  
  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
```

- Comando de inserção de dados:

```
INSERT INTO TB_ALUNO
(NOME, DATA_NASCIMENTO, E_MAIL, FONE_RES, FONE_COM, FAX,
CELULAR, PROFISSAO, EMPRESA )
VALUES
('ANTONIO DA SILVA', '1974.10.04', 'antonioss@impacta.com',
'123456789', '123456789', '', '987654321',
'ANALISTA DE SISTEMAS', 'IMPACTA TECNOLOGIA');
```

- Consulta na tabela:

```
SELECT * FROM TB_ALUNO;
```

	COD_ALUNO	NOME	DATA_NASCIMENTO	E_MAIL	FONE_RES	FONE_COM	FAX	CELULAR	PROFISSAO
1	1	ANTONIO DA SILVA	1974-10-04 00:00:00.000	antonioss@impacta.com	123456789	123456789		987654321	ANALISTA DE SISTEMAS

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

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

- Apresentando os dados:

```
SELECT * FROM TB_ALUNO;
```

	COD_ALUNO	NOME	DATA_NASCIMENTO	E_MAIL	FONE_RES	FONE_COM	FAX	CELULAR	PROFISSAO
1	1	ANTONIO DA SILVA	1974-10-04 00:00:00.000	antonioss@impacta.com	123456789	123456789		987654321	ANALISTA DE SISTEMAS
2	2	André da Silva	1980-01-02 00:00:00.000	andre@silva.com	23456789	23459876		998765432	ANALISTA DE SISTEMAS
3	3	Marcelo Soares	1983-04-21 00:00:00.000	marcelo@soares.com	23456789	23459876		998765432	INSTRUTOR

Podemos também fazer **INSERT de SELECT**:

- Criando uma tabela para receber os dados:

```
CREATE TABLE TB_ALUNO2
(
    COD_ALUNO           INT,
    NOME                VARCHAR(30),
    DATA_NASCIMENTO    DATETIME,
    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
```

- Inserindo através do **SELECT**:

```
INSERT INTO TB_ALUNO2
(COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_COM,FAX,C
ELULAR,PROFISSAO,EMPRESA)
SELECT
COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_COM,FAX,CE
LULAR,PROFISSAO,EMPRESA
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.

## 9.2.1. **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',
'luiza@luiza.com',
'23456789',
'23459876',
 '',
'998765432',
'ESTUDANTE',
'COLÉGIO MONTE VIDEL');
```

## 9.2.2. 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,
E_MAIL,
FONE_RES,
FONE_COM,
FAX,
CELULAR,
PROFISSAO,
EMPRESA )
VALUES
(
'PEDRO PAULO',
'1994.2.5',
'pedro@pedro.com',
'23456789',
'23459876',
 '',
'998765432',
'ESTUDANTE',
'COLÉGIO MONTE VIDEL'
);
```

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.

## 9.2.3. INSERT com CTE

É possível realizar um **INSERT** com CTE. Para isso, é necessário armazenar os dados na CTE e depois realizar a inserção na tabela. Vejamos o exemplo adiante:

```
WITH CTE (COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_
COM,FAX,
CELULAR,PROFISSAO,EMPRESA) AS
(
SELECT COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_
COM,FAX,
CELULAR,PROFISSAO,EMPRESA
FROM TB_ALUNO
)
INSERT INTO TB_ALUNO2
SELECT * FROM CTE
```

## 9.2.4. INSERT com tipo tabular

Podemos utilizar um tipo tabular para a inserção de dados. Vejamos o exemplo:

- Criação de um tipo tabular TYPE\_ALUNO:

```
CREATE TYPE TYPE_ALUNO AS TABLE
(
    ID           INT,
    ALUNO        VARCHAR(30),
    DATA_NASCIMENTO DATETIME
)
```

- Inserção no tipo tabular e depois na tabela:

```
DECLARE @TA TYPE_ALUNO

INSERT INTO @TA VALUES
(1 , 'ANTONIO' , '1998.10.1'),
(2 , 'MARIA' , '2000.8.2'),
(3 , 'ANA' , '1974.7.4')

INSERT INTO TB_ALUNO2 ( COD_ALUNO,NOME,DATA_NASCIMENTO)
SELECT * FROM @TA
```

## 9.3. Cláusula TOP

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.

No exemplo a seguir, será inserido apenas um registro, mesmo que a consulta retorne mais informações:

```
INSERT TOP (1) TB_ALUNO2
(COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_COM,FAX,
CELULAR,PROFISSAO,EMPRESA)
SELECT
COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_COM,FAX,
CELULAR,PROFISSAO,EMPRESA
FROM TB_ALUNO;
```

Messages

(1 row affected)

## 9.4. 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.

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**:

```
INSERT TB_ALUNO2
(COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_COM,FAX,
CELULAR,PROFISSAO,EMPRESA)
OUTPUT inserted.*
SELECT
COD_ALUNO,NOME,DATA_NASCIMENTO,E_MAIL,FONE_RES,FONE_COM,FAX,
CELULAR,PROFISSAO,EMPRESA
FROM TB_ALUNO;
```

Além da inserção dos dados, também é mostrado quais registros foram inseridos:

Results Messages										
	COD_ALUNO	NOME	DATA_NASCIMENTO	E_MAIL	FONE_RES	FONE_COM	FAX	CELULAR	PROFISSAO	E
1	1	ANTONIO DA SILVA	1974-10-04 00:00:00.000	antonios@impacta.com	123456789	123456789		987654321	ANALISTA DE SISTEMAS	I
2	2	ANTONIO DA SILVA	1974-10-04 00:00:00.000	antonios@impacta.com	123456789	123456789		987654321	ANALISTA DE SISTEMAS	I
3	3	MARIA LUIZA	1997-10-29 00:00:00.000	luiza@luiza.com	23456789	23459876		998765432	ESTUDANTE	I

Esse tipo de comando é muito útil quando necessitamos comprovar a execução de um comando.

## 9.5. Funções para campos autonomeáveis

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

Opções	Forma de utilização
<b>DBCC CHECKIDENT</b>	DBCC CHECKIDENT('Tabela')
<b>IDENTITYCOL</b>	SELECT IDENTITYCOL FROM Tabela
<b>IDENT_INCR</b>	SELECT IDENT_INCR('Tabela')
<b>IDENT_SEED</b>	SELECT IDENT_SEED('Tabela')
<b>@@IDENTITY</b>	SELECT @@IDENTITY
<b>SCOPE_IDENTITY</b>	SELECT SCOPE_IDENTITY
<b>IDENT_CURRENT</b>	SELECT IDENT_CURRENT('Tabela')
<b>SET IDENTITY_INSERT</b>	SET IDENTITY_INSERT Tabela ON/OFF

Para explicar a utilização de cada uma dessas opções, criaremos um banco de dados chamado **TESTE\_IDENTITY** e o colocaremos em uso. Depois, criaremos nesse banco as seguintes tabelas:

- **PROFESSOR**: Campo identidade iniciando em **1** e incremento **1**;
- **ALUNO**: Campo identidade iniciando em **10** e incremento **2**.

O código a seguir executa essas tarefas:

```

CREATE DATABASE TESTE_IDENTITY
GO
USE TESTE_IDENTITY;
-- Criar tabela PROFESSOR com campo identidade iniciando em 1
e ---- incrementando 1
CREATE TABLE PROFESSOR
( COD_PROF           INT IDENTITY,
  NOME                VARCHAR(30),
  CONSTRAINT PK_PROFESSOR PRIMARY KEY (COD_PROF) )
-- Criar tabela ALUNO com campo identidade iniciando em 10 e
-- incrementando 2
CREATE TABLE ALUNO
( COD_ALUNO          INT IDENTITY(10,2),
  NOME                VARCHAR(30),
  CONSTRAINT PK_ALUNO PRIMARY KEY (COD_ALUNO) );

```

O próximo procedimento é inserir dados nas duas tabelas criadas e consultá-las:

```

-- Inserir dados na tabela PROFESSOR
INSERT PROFESSOR VALUES ('MAGNO'),('AGNALDO'),('ROBERTO'),
                         ('RENATA'),('EDUARDO'),('MARCIO');
-- Inserir dados na tabela ALUNO
INSERT ALUNO VALUES ('ZÉ DA SILVA'),('CARLOS P. SILVA'),
                   ('ITAMAR COSTA');

-- Consultar as tabelas
SELECT * FROM PROFESSOR
SELECT * FROM ALUNO

```

O resultado da consulta é mostrado na imagem adiante:

	Réultados	Mensagens
	COD_PROF	NOME
1	1	MAGNO
2	2	AGNALDO
3	3	ROBERTO
4	4	RENATA
5	5	EDUARDO
6	6	MARCIO

	COD_ALU...	NOME
1	10	ZÉ DA SILVA
2	12	CARLOS SILVA
3	14	ITAMAR COSTA

- **SET IDENTITY\_INSERT**

Ativa ou desativa a permissão para inserir valor na coluna identidade de uma tabela.

Vamos excluir o professor cujo código é 2 e consultar a tabela **PROFESSOR**:

```
DELETE FROM PROFESSOR WHERE COD_PROF = 2  
SELECT * FROM PROFESSOR
```

No resultado, podemos perceber que a exclusão foi feita:

	COD_PROF	NOME
1	1	MAGNO
2	3	ROBERTO
3	4	RENATA
4	5	EDUARDO
5	6	MARCIO

Agora, vamos ativar a inserção de dados no campo **IDENTITY** para acrescentar **GODOFREDO** como professor de código 2.

```
SET IDENTITY_INSERT PROFESSOR ON  
-- OBS.: Não aceita INSERT posicional  
INSERT PROFESSOR (COD_PROF, NOME)  
VALUES (2, 'GODOFREDO')
```

Em seguida, desativamos a inserção de dados em **IDENTITY** e consultamos novamente a tabela **PROFESSOR**:

```
SET IDENTITY_INSERT PROFESSOR OFF  
SELECT * FROM PROFESSOR
```

No resultado, podemos perceber que a inserção foi feita corretamente:

	COD_PROF	NOME
1	1	MAGNO
2	2	GODOFREDO
3	3	ROBERTO
4	4	RENATA
5	5	EDUARDO
6	6	MARCIO

- **IDENTITYCOL**

Por meio desta opção, a coluna que possui a propriedade **IDENTITY**, bem como seus valores, é exibida.

```
SELECT IDENTITYCOL FROM PROFESSOR
```

```
SELECT IDENTITYCOL FROM ALUNO
```

- **IDENTITY\_SEED**

Por meio desta opção, o valor original definido para o **IDENTITY** da tabela é exibido.

```
SELECT IDENT_SEED('PROFESSOR')
```

```
SELECT IDENT_SEED('ALUNO')
```

- **IDENT\_INCR**

Por meio desta opção, o valor do incremento definido ao **IDENTITY** da tabela é exibido.

```
SELECT IDENT_INCR('PROFESSOR')
```

```
SELECT IDENT_INCR('ALUNO')
```

- **SCOPE\_IDENTITY**

Exibe o valor mais recente que foi definido para uma coluna **IDENTITY** pelo SQL Server. Essa coluna pode pertencer a qualquer tabela que faça parte de um escopo de uma procedure ou de qualquer trigger.

```
SELECT SCOPE_IDENTITY()
```

- **@@IDENTITY**

Trata-se de uma função de sistema que sempre exibirá o valor mais recente do **IDENTITY** que foi incluído em uma coluna proveniente de qualquer tabela.

```
SELECT @@IDENTITY
```

- **IDENT\_CURRENT**

Esta função retorna o valor mais recente do **IDENTITY** que foi incluído em qualquer sessão ou escopo.

```
SELECT IDENT_CURRENT ('tabela')
```

- **DBCC CHECKIDENT**

Erros podem surgir assim que o SQL Server gera o próximo valor do **IDENTITY** em uma tabela. Por exemplo, é possível que ocorra um erro de valor duplicado caso a coluna **IDENTITY** possua uma constraint **UNIQUE** ou **PRIMARY KEY**. Caso isso aconteça, basta executar o comando **DBCC CHECKIDENT**, cuja sintaxe é descrita a seguir:

```
DBCC CHECKIDENT( 'NomeTabela'[, {NORESEED|{RESEED[, novo_valor]} }])
```

Em que:

- **NomeTabela**: É o nome da tabela cujo valor de **IDENTITY** será verificado;
- **NORESEED**: Determina a não alteração do valor atual de **IDENTITY**;
- **RESEED**: Determina a alteração do valor de **IDENTITY**;
- **novo\_valor**: É o novo valor de **IDENTITY** a ser utilizado.

Vejamos o seguinte exemplo do uso de **DBCC CHECKIDENT**. Primeiramente excluiremos todos os registros da tabela **PROFESSOR** e, em seguida, acrescentaremos um novo registro (**MAGNO**):

```
DELETE FROM PROFESSOR  
INSERT INTO PROFESSOR VALUES ('MAGNO')
```

Agora, consultamos a tabela:

```
SELECT * FROM PROFESSOR
```

O resultado é mostrado na imagem a seguir:

COD_PROF	NOME
1	MAGNO

Podemos perceber que o código gerado (7) para **MAGNO** seguiu a mesma sequência anteriormente adotada. Agora, vamos excluir novamente os registros da tabela **PROFESSOR** e zerar, por meio de **DBCC CHECKIDENT**, o contador do **IDENTITY**:

```
DELETE FROM PROFESSOR  
-- "Zerar" o contador do IDENTITY  
DBCC CHECKIDENT( 'PROFESSOR', RESEED, 0 )
```

Por fim, acrescentamos novos dados à **PROFESSOR** e a consultamos:

```
INSERT PROFESSOR VALUES ('MAGNO'), ('AGNALDO'), ('ROBERTO'),  
('RENATA'), ('EDUARDO'), ('MARCIO');  
  
SELECT * FROM PROFESSOR
```

O resultado é exibido na imagem a seguir. Podemos perceber que **MAGNO** agora possui código 1:

	COD_PROF	NOME
1	1	MAGNO
2	2	AGNALDO
3	3	ROBERTO
4	4	RENATA
5	5	EDUARDO
6	6	MARCIO



### Pontos principais

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

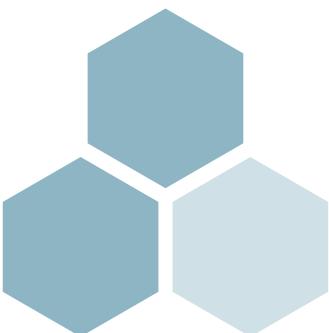
- Para acrescentar novas linhas de dados em uma tabela, utilizamos o comando **INSERT**;
- No **INSERT** posicional, não é necessário mencionar o nome dos campos, porém será obrigatório incluir todos os campos, exceto o campo autonumerável. Já no **INSERT** declarativo, os campos deverão ser informados no comando;
- A utilização do CTE permite que seja realizada uma consulta com dados temporários e inserção em bloco na tabela de destino;
- 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**;
- O SQL Server disponibiliza diversas opções para a verificação de informações relacionadas ao **IDENTITY** de uma tabela. **DBCC CHECKIDENT** e **IDENTITYCOL** são duas dessas opções.



9

# Inserção de dados

Teste seus conhecimentos



## 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. 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.

**5. 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')



9

# Inserção de dados

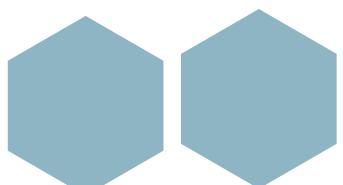


Mãos à obra!

Bruna Morimoto  
397.642.008-78



Editora  
**IMPACTA**



## 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:

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

3. Crie a tabela **TB\_UNIDADE** para armazenar unidades de medida:

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

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

8. Realize uma consulta no banco DB\_ECOMMERCE na tabela TB\_UNIDADE e insira os dados na tabela TB\_UNIDADE do banco PEDIDOS\_VENDA;

9. Faça uma consulta e mostre qual foi o último valor do campo autonumerável;

10. Realize uma consulta no banco DB\_ECOMMERCE na tabela TB\_CATEGORIA e insira os dados na tabela TB\_CATEGORIA do banco PEDIDOS\_VENDA, porém apenas 3 registros.

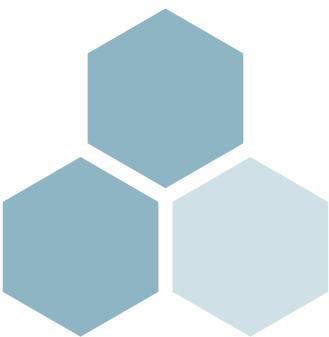




10

# Atualizando e excluindo dados

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



### 10.1. Introdução

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

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

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

### 10.2. UPDATE

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

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

A sintaxe de **UPDATE** é a seguinte:

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

Em que:

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

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

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

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

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

## 10.2.1. Alterando dados de uma coluna

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

```
-- Coloca o banco db_Ecommerce em uso
USE db_Ecommerce

-- Cria uma tabela a partir da tabela TB_EMPREGADO
SELECT *
INTO EMPREGADO_UPDATE
FROM TB_EMPREGADO

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

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

## 10.2.2. Alterando dados de diversas colunas

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

```
UPDATE EMPREGADO_UPDATE
SET
Nome = 'xxxxxxxxxxxxxxxxxx',
NUM_DEPEND =3,
DATA_NASCIMENTO ='2000.1.6'
WHERE ID_EMPREGADO = 5;
```

## 10.2.3. Utilizando TOP

Utilizada em uma instrução UPDATE, a cláusula TOP define uma quantidade ou porcentagem de linhas que serão atualizadas, conforme o exemplo a seguir, em que será aumentado em 40% o salário do empregado, porém somente os 5 primeiros registros serão afetados!

```
UPDATE TOP ( 5 ) EMPREGADO_UPDATE
SET SALARIO = SALARIO * 1.4;
```

## 10.2.4. UPDATE com subconsulta

Ao utilizarmos a instrução UPDATE em uma subconsulta, 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 subconsulta.

No exemplo adiante, será realizado um aumento de 10% nos salários dos empregados do departamento de TI:

```
UPDATE EMPREGADO_UPDATE
SET SALARIO = SALARIO * 1.10
WHERE ID_DEPARTAMENTO = (SELECT ID_DEPARTAMENTO FROM TB_
DEPARTAMENTO WHERE DEPARTAMENTO = 'TI');
```

## 10.2.5. 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 EMPREGADO_UPDATE
SET SALARIO *= 1.10
FROM TB_EMPREGADO E JOIN TB_DEPARTAMENTO D ON E.ID_
DEPARTAMENTO = D.ID_DEPARTAMENTO
WHERE D.DEPARTAMENTO = 'TI';
```

## 10.3. DELETE

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

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

Em que:

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

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

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

A sintaxe dessa instrução é a seguinte:

```
TRUNCATE TABLE <nome_tabela>
```

### 10.3.1. Excluindo todas as linhas de uma tabela

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

1. Primeiramente, gere uma cópia da tabela **TB\_EMPREGADO**:

```
SELECT * INTO EMPREGADO_DELETE  
FROM TB_EMPREGADO
```

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

```
DELETE FROM EMPREGADO_DELETE WHERE SALARIO > 5000;
```

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

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

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

```
SELECT * FROM EMPREGADO_DELETE WHERE ID_EMPREGADO IN (3,5,7);  
--  
DELETE FROM EMPREGADO_DELETE WHERE ID_EMPREGADO IN (3,5,7);
```

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

```
SELECT * FROM EMPREGADO_DELETE WHERE ID_EMPREGADO IN (3,5,7);
```

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

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

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

```
SELECT * FROM EMPREGADO_DELETE;
```

### 10.3.2. Utilizando TOP em uma instrução DELETE

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

```
-- Criar a tabela a partir do comando SELECT INTO
SELECT * INTO CLIENTE_MG FROM TB_CLIENTE;

-- Consultar CLIENTE_MG
SELECT * FROM CLIENTE_MG;

-- Excluir 10 registros da tabela CLIENTE_MG
DELETE TOP(10) FROM CLIENTE_MG;
-- Consultar
SELECT * FROM CLIENTE_MG;
```

### 10.3.3. DELETE com subconsulta

Podemos utilizar subconsulta para remover dados de uma tabela. Basta definir a cláusula **WHERE** da instrução **DELETE** como uma subconsulta. 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 2016. No exemplo a seguir é criada a tabela **TB\_PEDIDO\_DELETE** para realizar o teste:

```
SELECT *
INTO TB_PEDIDO_DELETE
FROM TB_PEDIDO

DELETE FROM TB_PEDIDO_DELETE
WHERE DATA_EMISSAO BETWEEN '2016.12.1' AND '2016.12.15' AND
ID_CLIENTE = (SELECT ID_CLIENTE FROM TB_VENDEDOR WHERE NOME =
'MARCELO');
```

### 10.3.4. 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 segunda quinzena de dezembro de 2016:

```
DELETE FROM TB_PEDIDO_DELETE
FROM TB_PEDIDO_DELETE AS P
JOIN TB_EMPREGADO AS E ON E.ID_EMPREGADO = P.ID_EMPREGADO
WHERE DATA_EMISSAO BETWEEN '2016.12.15' AND '2016.12.31'
AND NOME = 'MARCELO';
```

## 10.4. OUTPUT para DELETE e UPDATE

A sintaxe é a seguinte:

```
DELETE [FROM] <tabela> [OUTPUT deleted.<nomeCampo>|*[,...]]
WHERE <condição>
```

```
UPDATE <tabela> SET <campo1> = <expr1> [,...]
[OUTPUT deleted|inserted.<nomeCampo> [,...]]
[WHERE <condição>]
```

Veja alguns exemplos de **OUTPUT**:

```
-- Gerar uma cópia da tabela TB_EMPREGADO chamada EMP_TEMP
CREATE TABLE EMP_TEMP
```

```
( CODFUN          INT PRIMARY KEY,
  NOME            VARCHAR(30),
  ID_DEPARTAMENTO INT,
  ID_CARGO        INT,
  SALARIO         NUMERIC(10,2)
GO
```

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

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

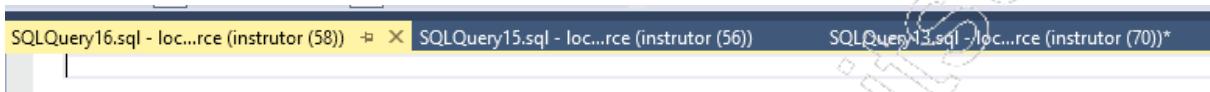
```
-- Alterar registros e mostrar os dados antes e depois da
-- alteração
UPDATE EMP_TEMP SET SALARIO *= 1.5
OUTPUT
  INSERTED.CODFUN, INSERTED.NOME, INSERTED.ID_DEPARTAMENTO,
  DELETED.SALARIO AS SALARIO_ANTIGO,
  INSERTED.SALARIO AS SALARIO_NOVO
WHERE ID_DEPARTAMENTO = 3;
GO
```

Observe que podemos conferir se a alteração foi feita corretamente porque é possível verificar se a condição está correta (**ID\_DEPARTAMENTO = 3**), e verificar o campo **SALARIO** antes e depois da alteração.

## 10.5. Transações

Quando uma conexão ocorre no MS-SQL, ela recebe um número de sessão. Mesmo que a conexão ocorra a partir da mesma máquina e mesmo login, cada conexão é identificada por um número único de sessão.

Observe a figura a seguir, em que temos três conexões identificadas pelas sessões 58, 56 e 70:



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.

### 10.5.1. Transações explícitas

As transações explícitas são aquelas em que seu início e seu término são determinados de forma explícita. Para definir este tipo de transação, os scripts Transact-SQL utilizam os seguintes comandos:

- **BEGIN TRANSACTION** ou **BEGIN TRAN**

Inicia um processo de transação para a sessão atual.

- **COMMIT TRANSACTION**, **COMMIT WORK** ou simplesmente **COMMIT**

Finaliza o processo de transação atual, confirmando todas as alterações efetuadas desde o início do processo.

- **ROLLBACK TRANSACTION**, **ROLLBACK WORK** ou **ROLLBACK**

Finaliza o processo de transação atual, descartando todas as alterações efetuadas desde o início do processo.

Sobre as transações explícitas, é importante considerar as seguintes informações:

- Se uma conexão for fechada com uma transação aberta, um **ROLLBACK** será executado;

- As operações feitas por um processo de transação ficam armazenadas em um arquivo de log (.ldf), que todo banco de dados possui;
- Durante um processo de transação, as linhas das tabelas que foram alteradas ficam bloqueadas para outras sessões.

Veja exemplo de transação:

- Para alterar os salários dos funcionários de ID\_CARGO = 5 para R\$ 950,00

```
-- Alterar os salários dos funcionários com ID_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.ID_EMPREGADO, inserted.NOME ,
deleted.salario as Salario_Anterior,
inserted.salario as Salario_Atualizado
WHERE ID_CARGO = 5

-- Conferir os resultados na listagem gerada pela cláusula
-- OUTPUT
-- Se estiver tudo OK...

COMMIT TRANSACTION

-- caso contrário

ROLLBACK TRANSACTION
```

## 10.6. MERGE

A instrução **MERGE** efetua operações de inserção, exclusão ou atualização em uma tabela de destino. Isso é feito com base nos resultados obtidos na junção com a tabela de origem. Assim, é possível sincronizar duas tabelas inserindo, excluindo ou atualizando as linhas de uma tabela a partir das diferenças encontradas na outra tabela. A seguir, temos a sintaxe da instrução **MERGE**:

```

MERGE
[ TOP ( expressao ) [ PERCENT ] ]
[ INTO ] tabela_alvo [[AS] alias_alvo ]
USING tabela_fonte [[AS] alias_fonte]
ON <condicao_busca_merge>
[ WHEN MATCHED [ AND <condicao_busca_clausula> ]
    THEN {UPDATE SET nome_coluna = expressao [,...]|
          DELETE}]
[ WHEN NOT MATCHED [ BY TARGET ] [ AND <condicao_busca_
clausula> ]
    THEN INSERT [(lista_coluna) VALUES (lista_valores) ]
[ WHEN NOT MATCHED BY SOURCE [ AND <condicao_busca_clausula>]
    THEN {UPDATE SET nome_coluna = expressao [,...]|
          DELETE} ]
[ OUTPUT <select_list_dml> ]

```

Em que:

- **TOP (expressao)**: Avalia somente as N primeiras linhas da tabela. Se a opção **PERCENT** estiver presente, a expressão será considerada como percentual;
- **tabela\_alvo**: A tabela alvo, ou seja, a tabela que sofrerá alterações;
- **alias\_alvo**: Alias da tabela alvo;
- **tabela\_fonte**: A tabela fonte, ou seja, a tabela que será comparada com a tabela alvo;
- **alias\_fonte**: Alias da tabela fonte;
- **ON <condicao\_busca\_merge>**: Define a expressão condicional que irá comparar a tabela fonte com a tabela alvo;
- **WHEN MATCHED**: Define o que será feito quando a condição **ON** for verdadeira. Opcionalmente, podemos inserir uma condição adicional utilizando **AND <condicao\_busca\_clausula>**. Podemos ter várias **WHEN MATCHED** no mesmo **MERGE**. Neste caso, as cláusulas iniciais terão que ter uma condição adicional, obrigatoriamente. Apenas o primeiro **WHEN MATCHED** verdadeiro será executado;

- **THEN:** Define o que será executado caso **WHEN MATCHED** seja verdadeiro. Pode ser um **UPDATE** ou **DELETE**;
- **WHEN NOT MATCHED [ BY TARGET ]:** Define uma instrução **INSERT** que será executada quando existir um registro na tabela fonte que não existe na tabela alvo de acordo com a condição **ON**. Uma condição adicional pode ser inserida com **AND <condicao\_busca\_clausula>**. Podemos ter várias **WHEN NOT MATCHED** no mesmo **MERGE**. Neste caso, as cláusulas iniciais terão que ter uma condição adicional, obrigatoriamente. Apenas o primeiro **WHEN NOT MATCHED** verdadeiro será executado;
- **WHEN NOT MATCHED BY SOURCE:** Define instruções **UPDATE** ou **DELETE** que serão executadas quando existirem registros na tabela alvo que não estão presentes na tabela fonte, de acordo com a condição **ON**. Uma condição adicional pode ser inserida com **AND <condicao\_busca\_clausula>**. Podemos ter várias **WHEN NOT MATCHED BY SOURCE** no mesmo **MERGE**. Neste caso, as cláusulas iniciais terão que ter uma condição adicional, obrigatoriamente. Apenas o primeiro **WHEN NOT MATCHED BY SOURCE** verdadeiro será executado.

O exemplo a seguir demonstra a utilização da instrução **MERGE**. Primeiramente, selecionamos o banco de dados **PEDIDOS**:

```
USE db_Ecommerce;
```

Em seguida, vamos gerar uma cópia da tabela **TB\_EMPREGADO** chamada **EMP\_TEMP** e testá-la:

```
--Apagar a tabela caso ela exista  
DROP TABLE EMP_TEMP  
  
--Cria a tabela a partir da cláusula INTO  
SELECT * INTO EMP_TEMP FROM TB_EMPREGADO;  
  
-- Testando  
SELECT * FROM EMP_TEMP;
```

Em seguida, excluiremos de **EMP\_TEMP** o funcionário de código 3:

```
DELETE FROM EMP_TEMP WHERE ID_CARGO = 3;
```

O próximo passo é alterar os salários dos funcionários do departamento 2 da tabela **EMP\_TEMP**:

```
UPDATE EMP_TEMP SET SALARIO *= 1.2  
WHERE ID_DEPARTAMENTO = 2
```

Agora, habilitaremos a inserção de dados no campo identidade da tabela em questão:

```
SET IDENTITY_INSERT EMP_TEMP ON
```

Feito isso, utilizamos **MERGE** a fim de fazer com que a tabela **EMP\_TEMP** fique igual à tabela **TB\_EMPREGADO**:

```
MERGE EMP_TEMP AS ET    -- Tabela alvo
USING TB_EMPREGADO AS E  -- Tabela fonte
ON ET.ID_EMPREGADO = E.ID_EMPREGAO -- Condição de comparação
```

Então, desejamos o seguinte: Quando o registro existir nas duas tabelas e (**AND**) o salário (**SALARIO**) for diferente, o campo de salário de **EMP\_TEMP** será atualizado. Para isso, utilizamos o código a seguir:

```
WHEN MATCHED AND E.SALARIO <> ET.SALARIO THEN
    UPDATE SET ET.SALARIO = E.SALARIO
```

Também, desejamos o seguinte: Quando o registro não existir em **EMP\_TEMP**, este terá o registro inexistente inserido. Para isso, utilizamos o código a seguir:

```
WHEN NOT MATCHED THEN
    INSERT
        (ID_EMPREGADO,NOME, DEPARTAMENTO, ID_CARGO, DATA_ADMISSAO,
        DATA_NASCIMENTO, SALARIO, NUM_DEPEND, SINDICALIZADO, OBS,
        FOTO)
    VALUES (ID_EMPREGADO,NOME, ID_DEPARTAMENTO, ID_CARGO, DATA_
        ADMISSAO,
        DATA_NASCIMENTO,SALARIO, NUM_DEPEND, SINDICALIZADO, OBS,
        FOTO);
```

Por fim, desabilitamos a inserção de dados no campo identidade:

```
SET IDENTITY_INSERT EMP_TEMP OFF;
```

O comando completo fica da seguinte forma:

```
SET IDENTITY_INSERT EMP_TEMP ON

MERGE EMP_TEMP           AS ET          -- Tabela alvo
USING TB_EMPREGADO AS E      -- Tabela fonte
ON ET.ID_EMPREGADO = E.ID_EMPREGADO -- Condição de comparação
WHEN MATCHED AND E.SALARIO <> ET.SALARIO THEN
    UPDATE SET ET.SALARIO = E.SALARIO
WHEN NOT MATCHED THEN
    INSERT (ID_EMPREGADO,NOME, ID_DEPARTAMENTO, ID_CARGO, DATA_
        ADMISSAO,
        DATA_NASCIMENTO,
        SALARIO, NUM_DEPEND, SINDICALIZADO, OBS, FOTO)
    VALUES (ID_EMPREGADO,NOME, ID_DEPARTAMENTO, ID_CARGO, DATA_
        ADMISSAO,
        DATA_NASCIMENTO,
        SALARIO, NUM_DEPEND, SINDICALIZADO, OBS, FOTO);

SET IDENTITY_INSERT EMP_TEMP OFF;
```

O resultado obtido com a sequência de códigos anterior é mostrado adiante:

Messages  
(15 rows affected)

## 10.6.1. OUTPUT em uma instrução MERGE

A cláusula **OUTPUT**, quando utilizada em uma instrução **MERGE**, tem a função de retornar uma linha para cada linha inserida, excluída ou atualizada na tabela de destino. As linhas são retornadas sem nenhuma ordem específica.

Primeiramente, criaremos uma cópia da tabela **TB\_EMPREGADO** chamada **EMP\_TEMP**:

```
IF OBJECT_ID('EMP_TEMP','U') IS NOT NULL
    DROP TABLE EMP_TEMP;
SELECT * INTO EMP_TEMP FROM TB_EMPREGADO;
-- Testando
SELECT * FROM EMP_TEMP;
GO
```

Em seguida, excluiremos da tabela **EMP\_TEMP** o funcionário de código 3 e alteraremos os salários dos funcionários do departamento 2 da mesma tabela:

```
-- Exclui os funcionários de código 3, 5 e 7 de EMP_TEMP
DELETE FROM EMP_TEMP WHERE ID_CARGO IN (3,5,7);
-- Altera os salários de EMP_TEMP dos funcionários do depto. 2
UPDATE EMP_TEMP SET SALARIO *= 1.2
WHERE ID_DEPARTAMENTO = 2
```

Depois, acrescentaremos registros na tabela alvo:

```
INSERT INTO EMP_TEMP
(NOME, ID_DEPARTAMENTO, ID_CARGO, SALARIO, DATA ADMISSAO)
VALUES ('MARIA ANTONIA',1,2,2000,GETDATE()),
       ('ANTONIA MARIA',2,1,3000,GETDATE());
```

Feita essa inserção, habilitaremos a inserção de dados no campo identidade e faremos com que a tabela **EMP\_TEMP** fique igual à tabela **TB\_EMPREGADO**:

```
SET IDENTITY_INSERT EMP_TEMP ON

-- Faz com que a tabela EMP_TEMP fique igual à tabela TB_
EMPREGADO
MERGE EMP_TEMP AS ET      -- Tabela alvo
USING TB_EMPREGADO AS E   -- Tabela fonte
ON ET.ID_EMPREGADO = E.ID_EMPREGADO -- Condição de comparação
-- Quando o registro existir nas 2 tabelas e o SALARIO for
-- diferente...
WHEN MATCHED AND E.SALARIO <> ET.SALARIO THEN
    -- ...Alterar o campo salário de EMP_TEMP
    UPDATE SET ET.SALARIO = E.SALARIO
-- Quando o registro não existir em EMP_TEMP...
WHEN NOT MATCHED THEN
    -- ...Inserir o registro em EMP_TEMP
    INSERT (ID_EMPREGADO,NOME,ID_DEPARTAMENTO, ID_CARGO,DATA_
ADMISSAO, DATA_NASCIMENTO,
        SALARIO, NUM_DEPEND, SINDICALIZADO, OBS, FOTO)
    VALUES (ID_EMPREGADO,NOME, ID_DEPARTAMENTO, ID_CARGO,DATA_
ADMISSAO, DATA_NASCIMENTO,
        SALARIO, NUM_DEPEND, SINDICALIZADO, OBS, FOTO)
WHEN NOT MATCHED BY SOURCE THEN
    -- Excluir o que existe na tabela alvo mas não existe
    -- na tabela fonte
    DELETE
OUTPUT $ACTION AS [Ação], INSERTED.ID_EMPREGADO AS [Código
Após],
DELETED.ID_EMPREGADO AS [Código Antes],
INSERTED.NOME AS [Nome Após],
DELETED.NOME AS [Nome Antes],
INSERTED.SALARIO AS [Salário Após],
DELETED.SALARIO AS [Salário Antes];
-- Desabilita a inserção de dados no campo identidade

SET IDENTITY_INSERT EMP_TEMP OFF;
```

**\$ACTION** retorna a ação executada pela instrução **MERGE** em cada registro.



### Pontos principais

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

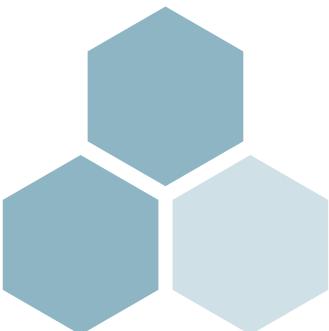
- Os dados pertencentes a múltiplas linhas de uma tabela podem ser alterados por meio do comando **UPDATE**;
- O comando **DELETE** deve ser utilizado quando desejamos excluir os dados de uma tabela;
- Transações são unidades de programação capazes de manter a consistência e a integridade dos dados. Devemos considerar uma transação como uma coleção de operações que executa uma função lógica única em uma aplicação de banco de dados;
- Todas as alterações de dados realizadas durante a transação são submetidas e tornam-se parte permanente do banco de dados caso a transação seja executada com êxito. No entanto, caso a transação não seja finalizada com êxito por conta de erros, são excluídas quaisquer alterações feitas sobre os dados;
- O comando **MERGE** realiza as operações de **INSERT**, **UPDATE** e **DELETE** em uma única instrução.



10

# Atualizando e excluindo dados

Teste seus conhecimentos



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

```
UPDATE TB_EMPREGADO  
SET SALARIO = SALARIO * 1.10  
WHERE ID_DEPARTAMENTO = (SELECT ID_DEPARTAMENTO FROM TB_DEPAR-  
TAMENTO  
WHERE DEPARTAMENTO = 'CPD');
```

- a) Atualiza o campo salário de todos os empregados.
- b) A sintaxe está errada, pois deve ser utilizado o operador IN em vez do igual.
- c) Nenhum salario será alterado devido à cláusula errada.
- d) Os empregados do departamento CPD terão um aumento de 10% no salário.
- e) Para o comando UPDATE não podemos utilizar subconsultas.

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

```
DELETE FROM TB_PEDIDO  
WHERE DATA_EMISSAO BETWEEN '2017.12.1' AND '2017.12.15' AND  
ID_EMPREGADO = (SELECT ID_EMPREGADO FROM TB_EMPREGADO  
WHERE NOME = 'MARCELO');
```

- a) Não podemos utilizar subconsultas 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 2017.

**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.ID_DEPARTAMENTO = D.ID_DEPARTAMENTO  
WHERE D.DEPARTAMENTO = 'RH';
```

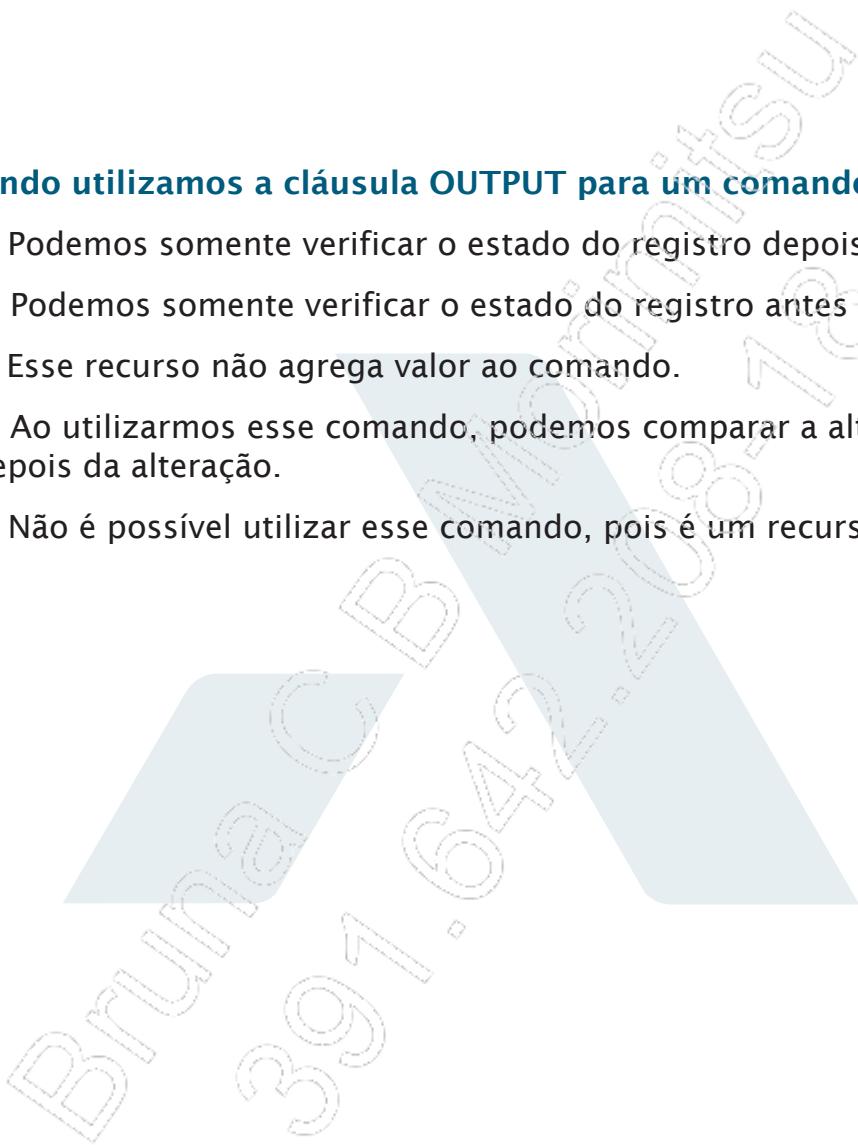
- a) Os empregados do RH terão um aumento de 10%.
- b) Os empregados do RH terão um aumento de 110%.
- c) Somente os empregados do RH terão aumento, inclusive os que possuírem COD\_DEPTO nulo.
- d) Nenhum empregado terá aumento.
- e) Deve ser utilizada uma subconsulta em vez de JOIN.

**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.ID_EMPREGADO = V.ID_EMPREGADO  
WHERE P.DATA_EMISSAO BETWEEN '2017.12.1' AND '2017.12.15' 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 2017.

### 5. Quando utilizamos a cláusula OUTPUT para um comando UPDATE:

- a) Podemos somente verificar o estado do registro depois da alteração.
  - b) Podemos somente verificar o estado do registro antes da alteração.
  - c) Esse recurso não agrega valor ao comando.
  - d) Ao utilizarmos esse comando, podemos comparar a alteração antes e depois da alteração.
  - e) Não é possível utilizar esse comando, pois é um recurso antigo.
- 

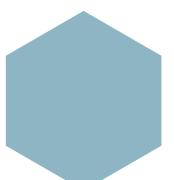


10

# Atualizando e excluindo dados



Mãos à obra!



## Laboratório 1

### A – Atualizando tabelas

1. Coloque em uso o banco de dados **DB\_ECOMMERCE**;
2. Altere a tabela **TB\_CARGO**, mudando o salário inicial do cargo QUÍMICO para 2500,00;
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 2017;
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;
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.

## Laboratório 2

### A – Trabalhando com opções adicionais e com o comando MERGE

1. Coloque em uso o banco de dados **DB\_ECOMMERCE**;
2. Gere uma cópia da tabela **TB\_PRODUTO** chamada **PRODUTOS\_COPIA**;
3. Exclua da tabela **PRODUTOS\_COPIA** os produtos que sejam do tipo '**CANETA**', exibindo os registros que foram excluídos (**OUTPUT**);
4. Aumente em 10% os preços de venda dos produtos do tipo **REGUA**, mostrando com **OUTPUT** as seguintes colunas: **ID\_PRODUTO**, **DESCRICAO**, **PRECO\_VENDA\_ANTIGO** e **PRECO\_VENDA\_NOVO**;
5. Utilizando o comando **MERGE**, faça com que a tabela **PRODUTOS\_COPIA** volte a ser idêntica à tabela **TB\_PRODUTO**, ou seja, o que foi deletado de **PRODUTOS\_COPIA** deve ser reinserido, e os produtos que tiveram seus preços alterados devem ser alterados novamente para que voltem a ter o preço anterior. O **MERGE** deve possuir uma cláusula **OUTPUT** que mostre as seguintes colunas: ação executada pelo **MERGE** (**DELETE**, **INSERT**, **UPDATE**), **ID\_PRODUTO**, **PRECO\_VENDA\_ANTIGO**, **PRECO\_VENDA\_NOVO**.



