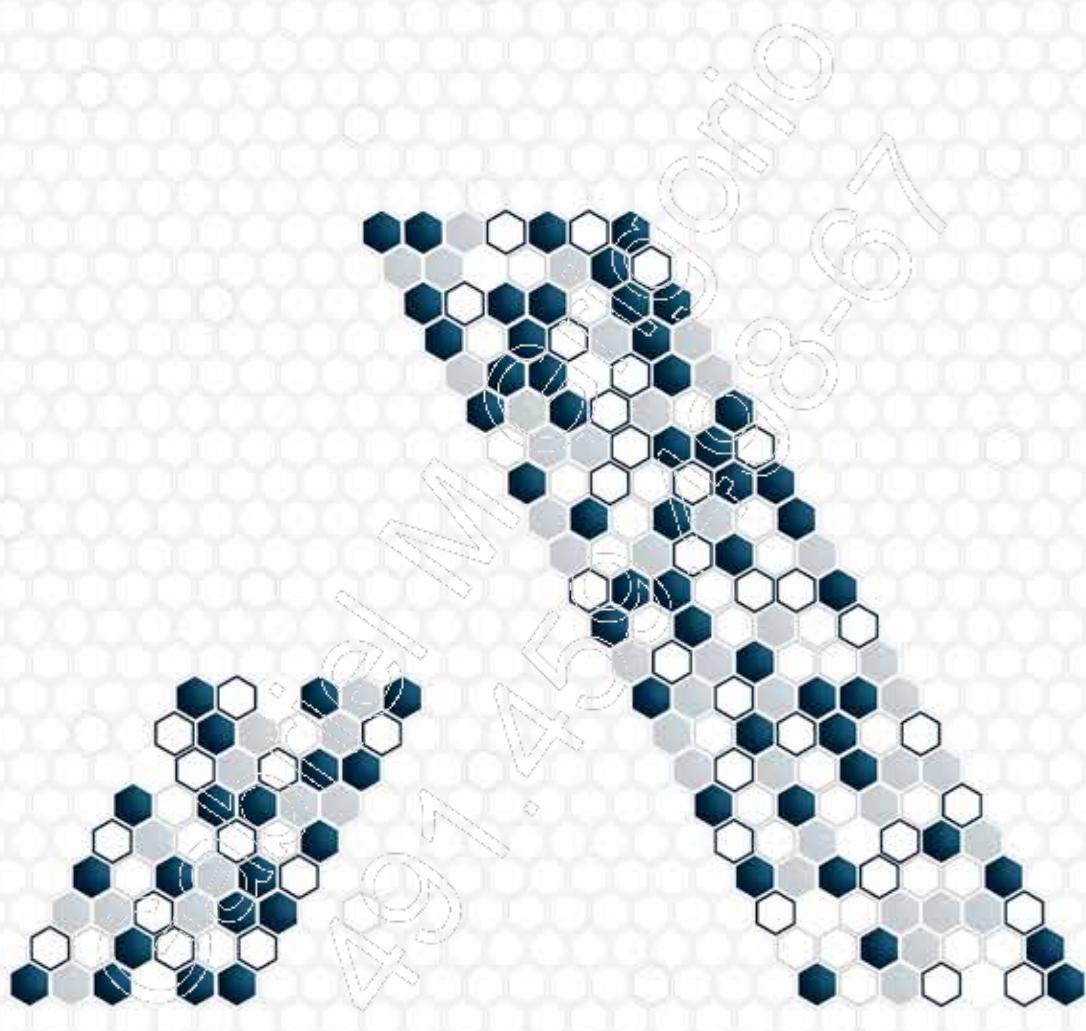
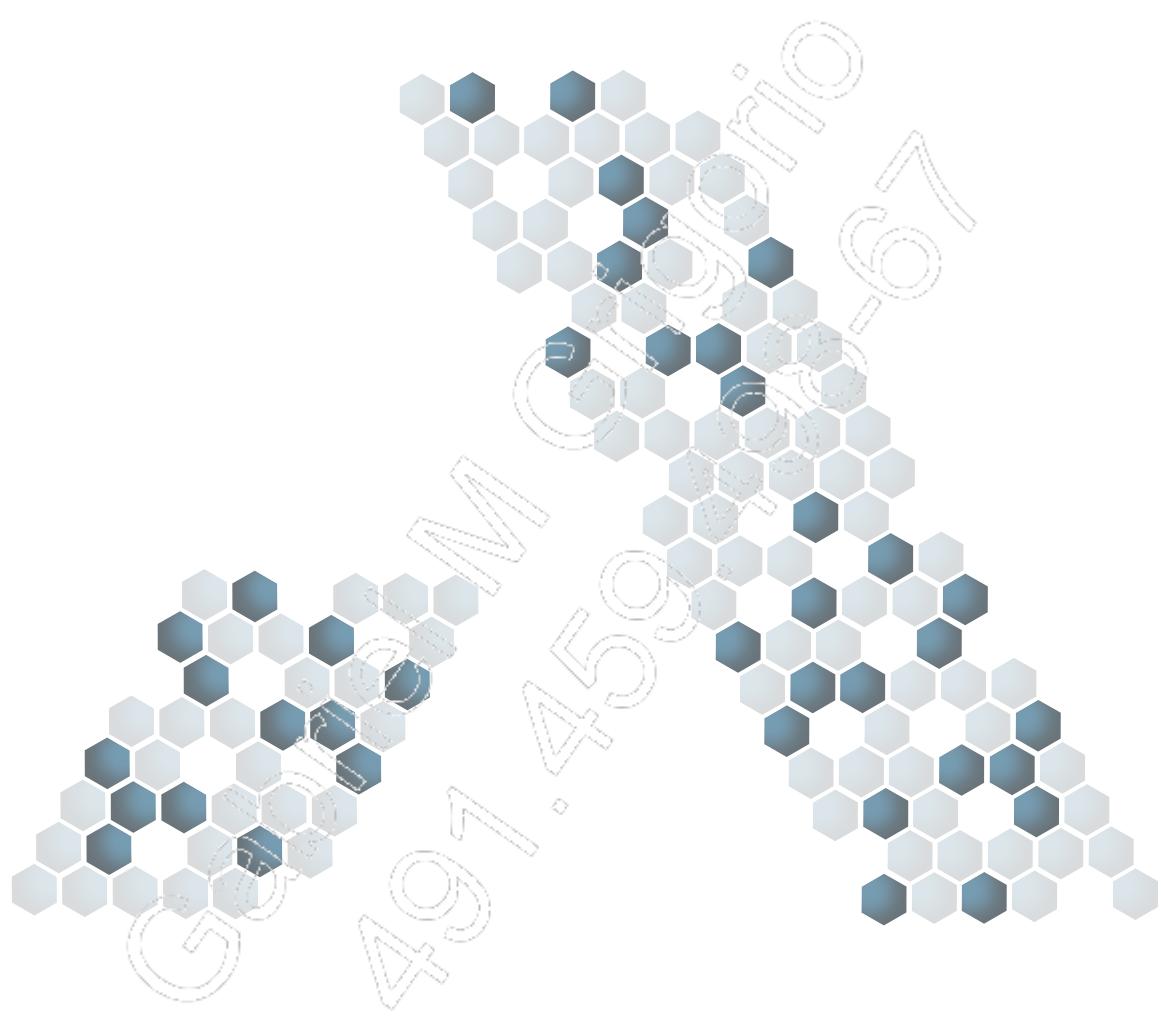


IMPACTA



Editora
IMPACTA



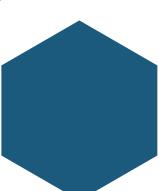


SQL 2016 - Programação em T-SQL (online)

Gabriel M. Prígorio

67.450°

497.450°



Editora
IMPACTA





Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

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

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

SQL 2016 - Programação em T-SQL (online)

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Supervisão de Desenvolvimento Digital

Alexandre Hideki Chicaoka

Produção, Gravação, Edição de Vídeo e Finalização

Xandros Luiz de Oliveira Almeida
Edcléia Souza de Lima

Roteirização

Hélio de Almeida Monteiro Júnior

Curso ministrado por

Hélio de Almeida Monteiro Júnior

Edição e Revisão final

Alexandre Hideki Chicaoka

Edição nº 1 | 1828_0_EAD

Abril/ 2017

Sobre o instrutor do curso:

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

Sumário

Apresentação	10
Aula 1 - Conceitos básicos	11
1.1. Introdução	12
1.2. Arquitetura física	12
1.3. Modelos de armazenamento	13
1.3.1. Modelo OLTP.....	13
1.3.2. Modelo OLAP	14
1.3.3. Comparativo entre OLTP e OLAP	15
1.4. Bancos de dados	16
1.4.1. Bancos de dados do sistema	16
1.4.1.1. Master	17
1.4.1.2. TEMPDB	17
1.4.1.3. MODEL.....	18
1.4.1.4. MSDB	18
1.4.1.5. Resource.....	18
1.4.2. Bancos de dados SNAPSHOT.....	19
1.4.3. Bancos de dados de usuários	19
1.4.4. Visualizando bancos de dados	19
1.5. Objetos de gerenciamento	19
1.5.1. Metadados	20
1.5.2. Catálogos	21
1.5.2.1. Catálogos do sistema.....	21
1.5.2.2. Catálogos do banco de dados	23
1.5.2.3. Procedures que retornam metadados	24
1.5.2.4. Funções que retornam metadados	25
1.5.6. Grupos de comandos T-SQL	26
1.7. Referenciando objetos do SQL Server.....	27
Pontos principais	29
Teste seus conhecimentos.....	31
Mãos à obra!	33
Aulas 2 a 7 - Comandos adicionais	37
1.1. SELECT.....	38
1.2. IIF/CHOOSE.....	46
1.3. LAG e LEAD.....	48
1.4. Paginação (FETCH e OFFSET)	49
1.5. Funções úteis para campos IDENTITY	50
1.6. MERGE	55
1.6.1. OUTPUT em uma instrução MERGE	58
1.7. Consultas cruzadas	59
1.7.1. PIVOT ()	61
1.7.2. UNPIVOT().....	64
1.8. Common Table Expressions (CTE)	66
1.8.1. CTE Recursiva	68
1.9. CROSS APPLY e OUTER APPLY	75
Pontos principais	79
Teste seus conhecimentos.....	81
Mãos à obra!	85

SQL 2016 - Programação em T-SQL (online)

Aulas 8 a 10 - Opções de definição de tabelas	89
1.1. Introdução	90
1.2. Tipos de dados	90
1.2.1. Tipos de dados nativos (Built-in)	90
1.2.2. Tipos de dados definidos pelo usuário	93
1.2.2.1. CREATE TYPE	94
1.2.3. DROP TYPE.....	95
1.2.4. CREATE RULE	95
1.2.5. sp_bindrule.....	96
1.2.6. sp_unbindrule.....	96
1.2.7. CREATE DEFAULT	97
1.2.8. sp_bindefault.....	97
1.2.9. sp_unbindefault	98
1.2.10. Tabelas de sistema	99
1.2.11. Tabela systypes	99
1.2.12. Tabela sysobjects.....	100
1.2.13. Tabela syscomments.....	100
1.2.14. Trabalhando com UDDT	101
1.3. Sequências.....	104
1.4. Sinônimos	106
1.5. Trabalhando com objetos binários	107
1.5.1. Campos binários	107
1.6. FILETABLE	109
1.7. Colunas computadas.....	117
Pontos principais	118
Teste seus conhecimentos.....	119
Mãos à obra!.....	123
Aulas 11 e 12 - Customizando consultas	129
1.1. Introdução	130
1.2. Plano de execução	130
1.3. Dicas para construir consultas	131
1.4. Índices	133
1.4.1. Criando índices	133
1.4.2. Índices compostos	135
1.4.3. INCLUDE	135
1.4.4. Excluindo índices	136
1.5. Otimizando consultas	136
1.5.1. Hints.....	137
1.5.2. Customizando bloqueios na seção	141
Pontos principais	143
Teste seus conhecimentos.....	145
Mãos à obra!.....	147
Aulas 13 a 15 - Acesso a recursos externos	151
1.1. Introdução	152
1.2. OPENROWSET	152
1.3. BULK INSERT	158
1.4. XML	161
1.4.1. FOR XML	161
1.4.2. Métodos XML	177
1.4.2.1. Query.....	178

Sumário

1.4.2.2. Value	180
1.4.2.3. Exists	180
1.4.2.4. Nodes	181
1.4.3. Gravando um arquivo XML	181
1.4.4. Abrindo um arquivo XML	182
1.5. JSON	183
1.5.1. FOR JSON	183
1.5.2. OPENJSON	185
1.5.3. JSON_VALUE	186
1.5.4. JSON_QUERY	187
1.5.5. ISJSON	188
1.5.6. Exportação para arquivo JSON	189
1.5.7. Importação de arquivo JSON	190
Pontos principais	192
Teste seus conhecimentos.	193
Mãos à obra!	195
Aulas 16 a 18 - Views	199
1.1. Introdução	200
1.2. Utilizando views	200
1.2.1. Vantagens oferecidas pelas views	201
1.2.2. Restrições	201
1.2.3. Tabela syscomments	202
1.2.4. Views de catálogo	203
1.3. CREATE VIEW	203
1.3.1. Utilizando WITH ENCRYPTION	204
1.3.2. Utilizando WITH SCHEMABINDING	205
1.3.3. Utilizando WITH CHECK OPTION	206
1.4. ALTER VIEW	207
1.5. DROP VIEW	208
1.6. Visualizando informações sobre views	208
1.7. Views atualizáveis	209
1.8. Retornando dados tabulares	210
Pontos principais	213
Teste seus conhecimentos.	215
Mãos à obra!	217
Aulas 19 a 21 - Introdução à programação	219
1.1. Introdução	220
1.2. Variáveis	220
1.2.1. Atribuindo valores às variáveis	220
1.3. Operadores	221
1.3.1. Operadores aritméticos	221
1.3.2. Operadores relacionais	222
1.3.3. Operadores lógicos	222
1.3.4. Precedência	223
1.4. Controle de fluxo	223
1.4.1. BEGIN/END	224
1.4.2. IF/ELSE	224
1.5. WHILE	226
1.5.1. BREAK	226

SQL 2016 - Programação em T-SQL (online)

1.5.2.	CONTINUE.....	227
1.5.3.	Exemplos	227
1.6.	Outros comandos	228
1.6.1.	GOTO	228
1.6.2.	RETURN	229
1.6.3.	WAITFOR.....	230
1.6.4.	EXISTS.....	230
1.6.5.	Atribuição de valor de uma consulta	230
	Pontos principais	231
	Teste seus conhecimentos.....	233
	Mãos à obra!.....	237
Aula 22 - Funções		241
1.1.	Introdução	242
1.2.	Funções e stored procedures	242
1.3.	Funções escalares	242
1.3.1.	Funções determinísticas e não determinísticas	244
1.4.	Funções tabulares	244
1.4.1.	Funções tabulares in-line	245
1.4.2.	Funções tabulares com várias instruções	245
1.5.	Funções nativas (built-in)	246
1.5.1.	Funções de texto	247
1.5.2.	Funções de data e hora	249
1.5.3.	Funções de conversão	249
1.6.	Funções de classificação	254
1.6.1.	ROW_NUMBER	254
1.6.2.	RANK	256
1.6.3.	DENSE_RANK	258
1.6.4.	NTILE	260
1.6.5.	ROW_NUMBER, RANK, DENSE_RANK e NTILE	262
1.7.	Funções definidas pelo usuário	263
1.7.1.	Funções escalares	264
1.7.2.	Funções tabulares	273
1.8.	Campos computados com funções	274
	Pontos principais	276
	Teste seus conhecimentos.....	277
	Mãos à obra!.....	281
Aulas 23 e 24 - Stored procedures		285
1.1.	Introdução	286
1.2.	Stored procedures.....	286
1.2.1.	Vantagens.....	287
1.2.2.	Considerações	287
1.2.3.	CREATE PROCEDURE.....	289
1.2.4.	Alterando stored procedures.....	291
1.2.5.	Excluindo stored procedures.....	291
1.3.	Declarando parâmetros	291
1.3.1.	Exemplos	292
1.3.2.	Passagem de parâmetros posicional.....	293
1.3.3.	Passagem de parâmetros nominal	294
1.4.	Retornando valores	294

Sumário

1.4.1.	PRINT	295
1.4.2.	SELECT	296
1.4.3.	Parâmetros de saída (OUTPUT)	299
1.5.	Cursor	300
1.6.	Depurando stored procedures	301
1.6.1.	Parâmetros tabulares (table-valued)	305
1.6.2.	Boas práticas	306
1.7.	Recompilando stored procedures	308
1.8.	Query dinâmicas	308
1.9.	Tratamento de erros	309
1.9.1.	Severidade de um erro	309
1.9.2.	@@ERROR	311
1.9.3.	TRY...CATCH	311
1.9.4.	Funções para tratamento de erros	313
1.10.	Trabalhando com mensagens de erro	314
1.10.1.	SP_ADDMESSAGE	314
1.10.2.	RAISERROR	316
1.10.3.	THROW	318
1.10.4.	Exemplo de tratamento de erros	321
	Pontos principais	324
	Teste seus conhecimentos	327
	Mãos à obra!	331

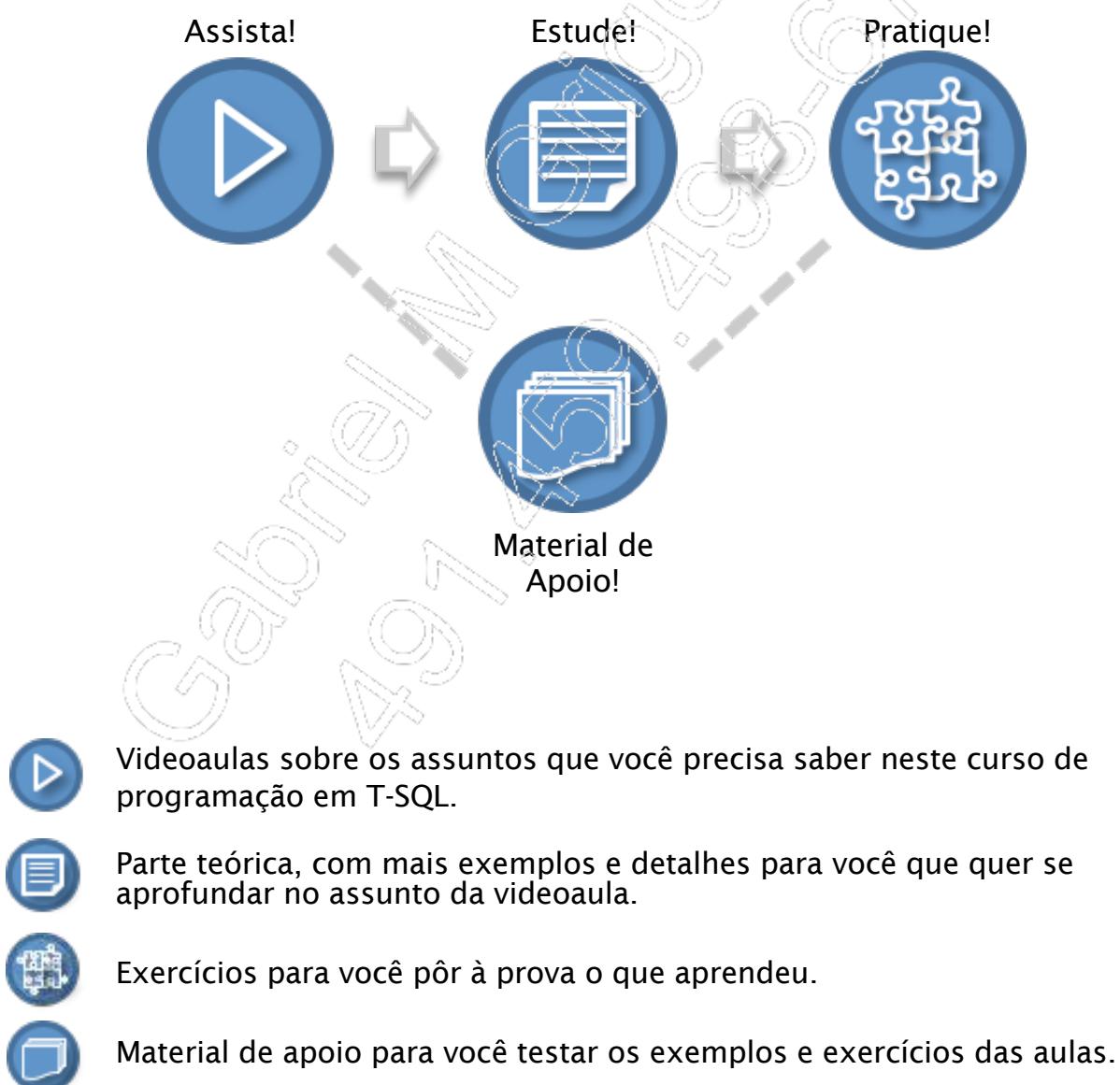
	Aula 25 - Triggers	335
1.1.	Introdução	336
1.2.	Triggers	336
1.2.1.	Diferenças entre triggers e constraints	338
1.2.2.	Considerações	338
1.2.3.	Visualizando triggers	339
1.2.4.	Alterando triggers	340
1.2.5.	Desabilitando e excluindo triggers	340
1.2.5.1.	DISABLE TRIGGER	340
1.2.5.2.	ENABLE TRIGGER	341
1.2.5.3.	DROP TRIGGER	341
1.3.	Triggers DML	342
1.3.1.	Tabelas INSERTED e DELETED	343
1.3.2.	Triggers de inclusão	344
1.3.3.	Triggers de exclusão	344
1.3.4.	Trigger de alteração	344
1.3.5.	Trigger INSTEAD OF	345
1.4.	Triggers DDL	350
1.4.1.	Criando triggers DDL	351
1.5.	Triggers de logon	355
1.6.	Aninhamento de triggers	357
1.6.1.	Habilitando e desabilitando aninhamento	358
1.7.	Recursividade de triggers	359
	Pontos principais	360
	Teste seus conhecimentos	361
	Mãos à obra!	363

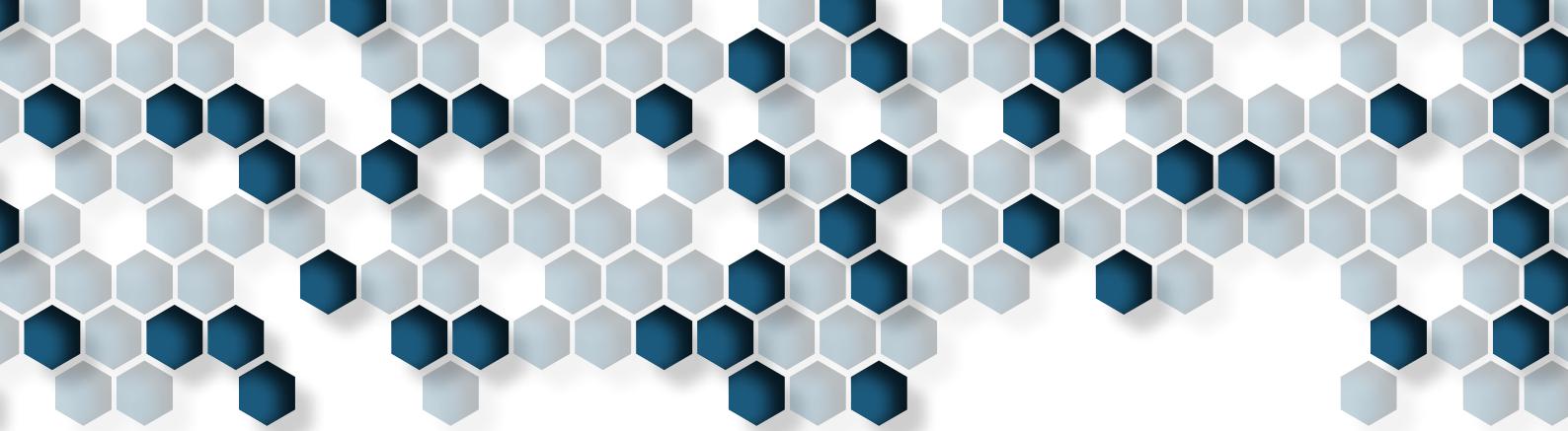
Apresentação

Bem-vindo!

É um prazer tê-lo como aluno do nosso curso online SQL 2016 - Programação em T-SQL. Este curso é ideal para você que deseja se aperfeiçoar na programação com a linguagem Transact - SQL por meio do gerenciador de banco de dados Microsoft SQL Server. Você fará uso de técnicas avançadas para criar rotinas que facilitarão tarefas como automatização de processos, integração de dados e otimização de consultas. Bom aprendizado!

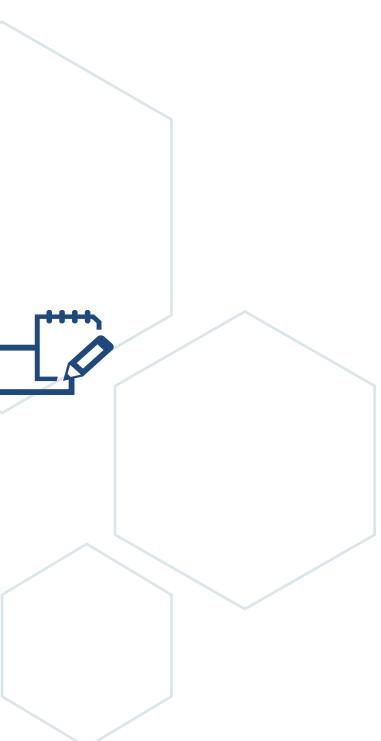
Como estudar?





Conceitos básicos

- 
- 
- 
- ◆ Arquitetura física;
 - ◆ Modelos de armazenamento;
 - ◆ Bancos de dados;
 - ◆ Objetos de gerenciamento;
 - ◆ Grupos de comandos T-SQL;
 - ◆ Referenciando objetos do SQL Server.



Esta Leitura Complementar refere-se ao conteúdo da Aula 1.

1.1. Introdução

Definimos o SQL Server como sendo um gerenciador de banco de dados relacional que envia solicitações entre o cliente e o servidor por meio da arquitetura cliente/servidor. Para enviar essas solicitações, o SQL Server também utiliza: o XML, a linguagem Transact-SQL, o MDX, que são as expressões multidimensionais, e o SQL DMO, que são os objetos de gerenciamento distribuído.

O SQL Server realiza o gerenciamento dos bancos de dados e aloca os recursos disponíveis no servidor. A tecnologia utilizada pelo SQL Server permite, ainda, realizar o armazenamento dos dados em ambientes OLAP e OLTP, a respeito dos quais estudaremos a seguir.

1.2. Arquitetura física

Os processos de desenvolvimento e de gerenciamento de um aplicativo são afetados pela arquitetura utilizada com a finalidade de implementar um sistema. Vejamos, a seguir, quais são os tipos de arquitetura utilizados:

- **Servidor inteligente:** Este tipo de arquitetura utiliza um sistema de duas camadas. Com ela, o processamento, em sua grande maioria, é realizado no servidor de dados, sendo que os clientes exercem apenas o papel de interface, ou seja, apresentação dos dados;
- **Cliente inteligente:** Este tipo de arquitetura também utiliza um sistema de duas camadas. Com ela, o processamento do sistema é distribuído entre cliente e servidor. A parte do sistema que é executada no servidor recebe o nome de aplicação **back-end**, ao passo que a parte do sistema que é executada no cliente recebe o nome de aplicação **front-end**;
- **Internet:** Este tipo de arquitetura utiliza um sistema de três camadas. Com ela, temos o servidor de dados, no qual a parte do banco de dados da aplicação é executada por meio de procedures, triggers e functions; o servidor Web, em que as aplicações e as apresentações são executadas; e os clientes, os quais acessam o sistema a partir de um browser;
- **Sistemas de N camadas:** Esta arquitetura apresenta a figura do servidor de negócios a fim de valorizar o desempenho do processamento de sistemas que gerenciam um grande volume de dados e um grande volume de regras de negócios. Dessa forma, esse esquema apresenta: o servidor de dados, que tem a finalidade de processar a aplicação por meio de procedures, triggers e functions; o servidor de aplicação, que tem a função de processar as regras de negócios; e os clientes, que são responsáveis por processar a interface com o usuário final.

1.3. Modelos de armazenamento

O SQL gerencia bancos de dados baseados em dois modelos de armazenamento: o sistema **OLTP (On-line Transaction Processing)** e o sistema **OLAP (On-line Analytical Processing)**, que detalharemos adiante.

1.3.1. Modelo OLTP

Os sistemas **OLTP (On-line Transaction Processing)**, isto é, de processamento de transações em tempo real, são aqueles que dão origem aos dados. Os sistemas OLTP foram desenvolvidos para processar as diversas transações recebidas ao mesmo tempo. A principal finalidade dos dados desses sistemas é suportar as transações. Os dados com os quais este sistema trabalha são normalizados e armazenados em tabelas inter-relacionadas.

Os sistemas OLTP são voltados para a manipulação destes dados, realizando as operações de gravação (**INSERT**, **UPDATE** e **DELETE**). Têm como finalidade reduzir a redundância de informações e garantir a integridade, consistência e bom desempenho nas operações de inclusão, exclusão e alteração de dados.

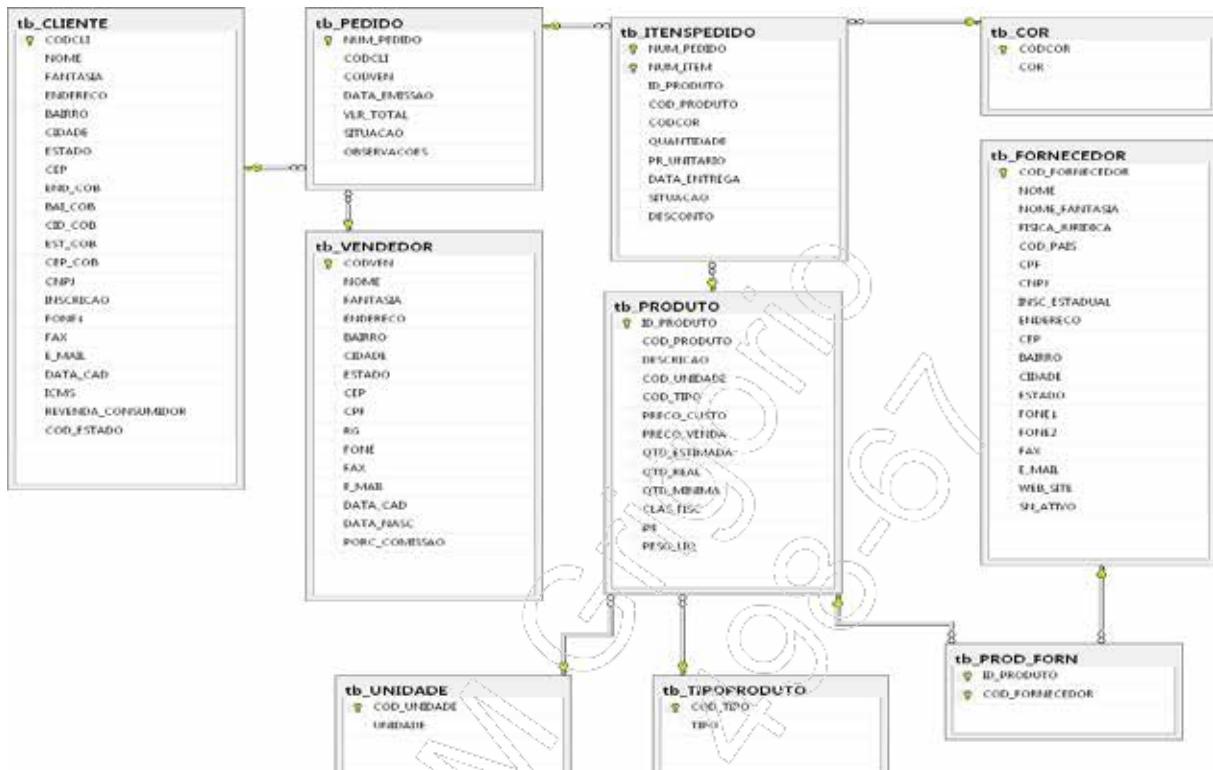
Um banco de dados OLTP apresenta estruturas complexas e contém uma grande quantidade de dados. Ele oferece a infraestrutura tecnológica capaz de suportar as operações realizadas diariamente por uma organização, como uma grande quantidade de usuários manipulando os dados de forma simultânea e em tempo real. Por isso, é importante determinar o modelo de acesso a dados e planejar deixar juntos os dados frequentemente acessados.

Tal banco de dados transacional é capaz, também, de representar a frequente mudança de estado de uma organização, embora não se possa salvar o histórico. Quando trabalhamos com um banco de dados OLTP, as transações individuais são completadas de forma rápida.

Exemplos:

- Sistema de transações bancárias que registra todas as operações efetuadas em um banco;
- Caixas de multibanco;
- Reservas de viagens ou hotel on-line;
- Cartões de crédito;
- ERP de uma empresa;
- Sistema de vendas de uma loja, entre outros.

A seguir, um exemplo de modelagem para sistemas OLTP:



1.3.2. Modelo OLAP

Os sistemas **OLAP** (On-line Analytical Processing) utilizam dados sem normalização, permitindo a criação de estruturas tridimensionais a fim de otimizar o desempenho no acesso a informações.

Os sistemas OLAP têm por objetivo realizar a leitura de grandes volumes de dados para a geração de relatórios gerenciais que dão suporte a decisões. Entretanto, para os recursos do sistema, é comum que a leitura dos dados seja um trabalho dispendioso, visto que os bancos de dados costumam chegar a um tamanho muito grande, correspondente a centenas de gigabytes ou terabytes.

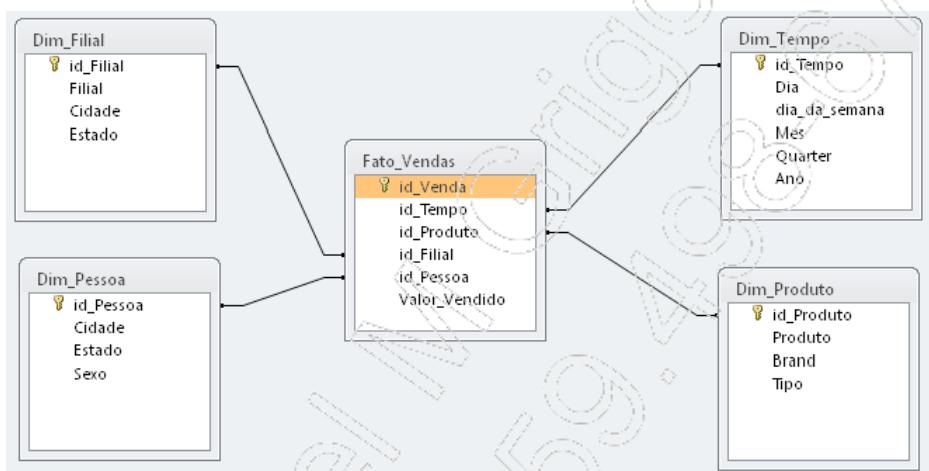
Neste tipo de processamento, chamado de **processamento analítico em tempo real**, devemos dar maior importância aos dados dos históricos que estejam totalizados e consolidados do que aos dados detalhados ou individualizados.

O **SSAS (SQL Server 2016 Analysis Services)** oferece alguns recursos e ferramentas que permitem desenvolver, distribuir e fazer a manutenção de diversos objetos de suporte.

As características do sistema OLAP são as seguintes:

- Gera informações gerenciais e estratégicas para a diretoria da empresa;
- Pouco normalizado;
- Periodicamente os dados do banco de dados OLTP são transferidos para o banco de dados OLAP em um processo chamado de CARGA;
- Trouxe uma grande capacidade de efetuar cálculos complexos como previsões, percentuais de crescimento e médias diversas.

Vejamos, a seguir, um exemplo de modelo para uma base OLAP:



1.3.3. Comparativo entre OLTP e OLAP

Uma comparação entre os sistemas OLTP e OLAP pode ser observada na tabela a seguir:

Características	OLTP – Bancos de dados operacionais	OLAP – Data Warehouse
Objetivo	Operações diárias do negócio	Analizar o negócio
Utilização	Operacional	Informativo
Tipo de processamento	OLTP	OLAP
Unidade de trabalho	Inclusão, alteração e exclusão	Carga e consulta

Características	OLTP – Bancos de dados operacionais	OLAP – Data Warehouse
Quantidade de usuários	Milhares	Dezenas
Tipo de usuário	Usuários em geral	Comunidade gerencial
Condições dos dados	Dados operacionais	Dados analíticos
Volume	Megabytes – Gigabytes	Gigabytes – Terabytes
Granularidade	Detalhados	Detalhados e resumidos
Redundância	Não ocorre	Ocorre
Manutenção desejada	Mínima	Constante
Acesso a registros	Dezenas	Milhares
Atualização	Contínua (em tempo real)	Periódica (em batch)
Número de índices	Poucos e simples	Muitos e complexos
Intenção dos índices	Localizar um registro	Aperfeiçoar consultas

1.4. Bancos de dados

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.

1.4.1. 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 com que a integridade e a segurança dos dados sejam 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.

1.4.1.1. 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 de dados 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.

1.4.1.2. 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.

1.4.1.3. 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.

1.4.1.4. 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.

1.4.1.5. 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 databases no SQL Server Management Studio. Seus arquivos de dados e de transações estão disponíveis em: **C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data**.

1.4.2. 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.

1.4.3. Bancos de dados de usuários

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.

O comando básico para a criação de um banco é:

```
CREATE DATABASE {nome do banco}
```

1.4.4. Visualizando bancos de dados

Para visualizar os bancos de dados, basta abrir o SQL Server Management Studio e, em seguida, na janela **Object Explorer**, expandir a opção **Databases**.

1.5. Objetos de gerenciamento

Os objetos de gerenciamento utilizados no SQL Server serão descritos nos subtópicos a seguir.

1.5.1. 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:

Descrição	Tipo
Função de agregação	AF
CONSTRAINT: CHECK	C
CONSTRAINT: DEFAULT	D
CONSTRAINT: FOREIGN KEY	F
Função escalar	FN
Função escalar: Assembly (CLR)	FS
Função tabular: Assembly (CLR)	FT
Função IN-LINED	IF
Tabela Interna	IT
LOG	L
Procedimento armazenado	P
Procedimento armazenado: Assembly (CLR)	PC
CONSTRAINT: PRIMARY KEY	PK
Procedimento armazenado: Replication filter	RF
Tabela de Sistema	S
Sinônimo	SN

Tipo	Descrição
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

1.5.2. Catálogos

São chamados de **catálogos** os recursos existentes para extrairmos metadados do banco de dados, como as tabelas de catálogo, as views de catálogo e as procedures de catálogo.

1.5.2.1. 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 PEDIDOS

-- 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
-- colunas 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
```

- Exemplo de procedures de catálogo

```
-- 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_ITENSPEDIDO
-- Estrutura de uma tabela
exec sp_help TB_PEDIDO
```

1.5.2.2. Catálogos do banco de dados

O banco de dados **model** possui alguns objetos que são copiados em cada um dos bancos de dados de usuários quando eles são criados. Esses objetos, que podem ser tabelas, procedures, views e roles, são utilizados com a finalidade de permitir que o SQL realize o gerenciamento de seu sistema.

A fim de que o SQL Server seja capaz de gerenciar as ações realizadas pelos usuários sobre seus dados, os objetos que são criados em seu banco de dados são registrados de forma automática nos objetos do Catálogo do banco de dados, o qual está presente em todo o banco de dados.

1.5.2.3. Procedures que retornam metadados

Há diversas stored procedures disponibilizadas pelo SQL Server que retornam metadados ao serem executadas.

A seguir, destacamos algumas delas:

Procedure	Descrição
SP_HELPDEVICE	Dispositivos de backup.
SP_HELP	Objetos do banco de dados.
SP_HELPCONSTRAINT	CONSTRAINTS.
sp_helpextendedproc	Procedimentos armazenados estendidos atualmente definidos, bem como o nome da biblioteca de vínculo dinâmico (DLL).
sp_helpfile	Os nomes físicos e os atributos de arquivos do banco de dados.
sp_helpdb	Banco de dados.
sp_helpdbfixedrole	Roles do banco de dados.
sp_helpfilegroup	Nomes e os atributos de grupos de arquivos.
sp_helpsrvrole	Roles do servidor de banco de dados.
sp_helptrigger	Trigger DML definidos para a tabela.
sp_helpuser	Usuários do banco de dados.
sp_helpstats	Estatísticas sobre colunas e índices.
sp_helptext	A definição de procedures, funções, views etc.
sp_helplogins	Logons e usuários associados em cada banco de dados.
sp_helprolmemer	Membros de uma função no banco de dados atual.
sp_helpntgroup	Grupos do Windows com contas no banco de dados atual.
sp_helprotect	Permissões para um objeto ou permissões de instrução, no banco de dados atual.
sp_helpremotelogin	Logons remotos para um determinado servidor remoto, ou para todos os servidores, definido no servidor local.
sp_helpserver	Servidor remoto ou réplica, ou sobre todos os servidores de ambos os tipos.
sp_helprole	Funções no banco de dados atual.

Procedure	Descrição
sp_helpsort	COLLATION.
sp_helpindex	Índices em uma tabela ou view.
sp_HELPLANGUAGE	Idioma alternativo específico ou sobre todos os idiomas.
sp_HELPLINKEDSRVLOGIN	Mapeamentos de logon definidos em um servidor vinculado e procedimentos armazenados remotos.

1.5.2.4. Funções que retornam metadados

Há diversas funções disponibilizadas pelo SQL Server que retornam metadados ao serem executadas. A seguir, citamos algumas delas:

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

Função	Descrição
OBJECT_SCHEMA_NAME	Nome do schema do objeto.
OBJECTPROPERTY	Propriedades de um objeto.
SCHEMA_ID	ID do schema.
TYPE_ID	ID de um tipo de dados.
TYPE_NAME	Nome de um tipo de dados.

1.6. Grupos de comandos T-SQL

Os comandos da linguagem Transact-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.

1.7. Referenciando objetos do SQL Server

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**

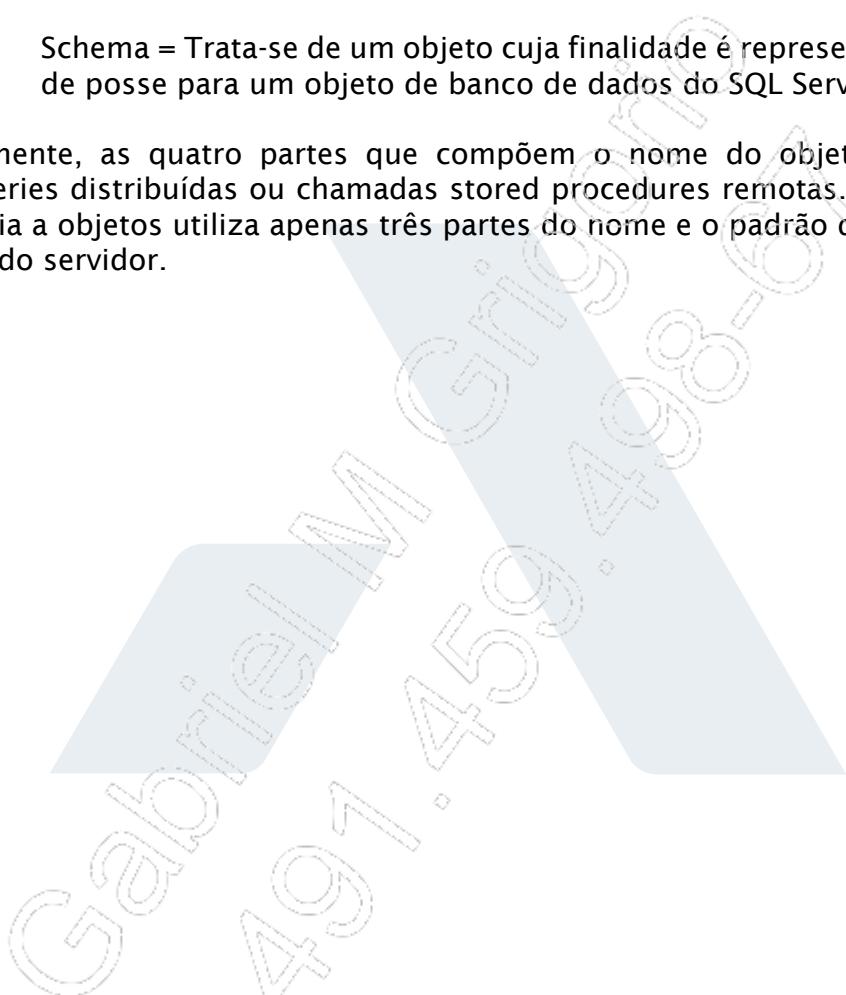
Embora tenhamos os nomes totalmente qualificados para os objetos do SQL Server, podemos omitir identificadores intermediários desses nomes, desde que eles sejam indicados por pontos, conforme podemos observar nas formas descritas a seguir:

- **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.

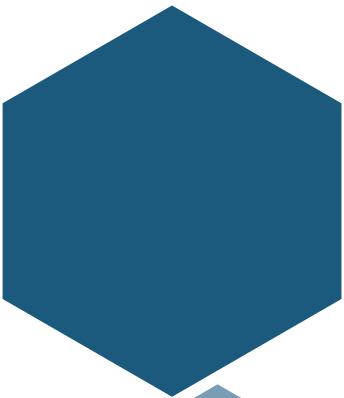
Normalmente, as quatro partes que compõem o nome do objeto são utilizadas para queries distribuídas ou chamadas stored procedures remotas. Grande parte da referência a objetos utiliza apenas três partes do nome e o padrão determinado para o nome do servidor.



Pontos principais

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

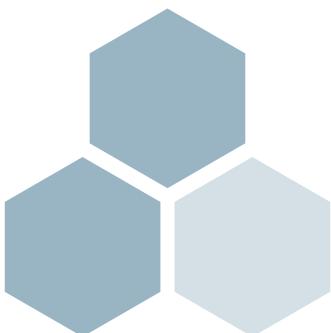
- O SQL Server realiza o gerenciamento dos bancos de dados e aloca os recursos disponíveis no servidor. O armazenamento dos dados é realizado em ambientes **OLTP (On-line Transaction Processing)**, que são aqueles que dão origem aos dados e são voltados para a manipulação desses dados, realizando operações de gravação (**INSERT, UPDATE e DELETE**) e **OLAP (On-line Analytical Processing)** que têm por objetivo realizar a leitura de grandes volumes de dados para a geração de relatórios gerenciais que dão suporte a decisão;
- 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 extrairmos 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**;
- Os comandos da linguagem Transact-SQL são divididos em quatro grupos: **DCL (Data Control Language), DDL (Data Definition Language) e DML (Data Manipulation Language)**;
- 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.



Conceitos básicos

Teste seus conhecimentos

Estes testes referem-se ao conteúdo da Aula 1.



1. Ao instalarmos um servidor de banco de dados, é recomendado definir um modelo para armazenar os dados. Com esta premissa, ao preparamos um projeto de um sistema ERP, qual o melhor modelo a ser utilizado?

- a) OLAP.
- b) OLTP.
- c) OLAP e OLTP.
- d) OLAP com modelo tabular.
- e) Qualquer um dos modelos atende à necessidade.

2. A empresa que você trabalha iniciou um projeto de BI (Business Intelligence), que consiste em armazenar as informações históricas de vendas. Qual modelo deve ser utilizado?

- a) OLAP.
- b) OLTP.
- c) OLAP e OLTP.
- d) OLTP com tabelas multidimensionais.
- e) Qualquer um dos modelos atende à necessidade.

3. Os bancos de dados de sistemas possuem funcionalidades distintas. Qual delas gerencia o SQL e controla os logins?

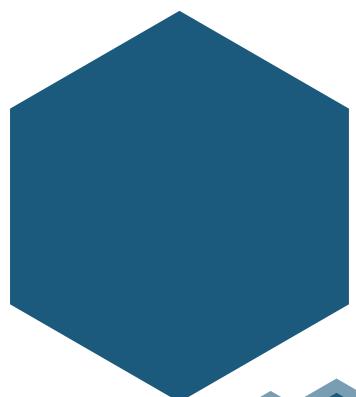
- a) MODEL
- b) TEMPDB
- c) MSDB
- d) RESOURCE
- e) MASTER

4. Qual afirmação está errada com relação ao Catálogo de Sistema?

- a) Apresenta informação das tabelas.
- b) Apresenta informação de views.
- c) Mostra as restrições de acesso a um determinado recurso.
- d) Apresenta as informações de metadados.
- e) É um recurso antigo que foi substituído pelo Object Explorer.

5. Qual item a seguir não é um grupo de comandos do SQL?

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



Conceitos básicos



Mãos à obra!

Gabriel Grigorio
497-A59-A8-67

Este laboratório refere-se ao conteúdo da Aula 1.



Editora
IMPACTA



Laboratório 1

A – Criação de um banco de dados

1. Acesse o **SQL Server Management Studio (SSMS)**;
2. Abra uma nova Query;
3. Crie um banco de dados de nome **DB_Aula_Impacta**;
4. Coloque o banco de dados em uso.

B – Consulta de objetos do servidor

1. 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.
2. Realize o mesmo procedimento utilizando o **Object Explorer** do SSMS.

Laboratório 2

A – Anexando o banco de dados Pedidos

1. Abra o **SQL Server Management Studio**;
2. Conecte a instância **Default**;
3. Com o botão direito do mouse, clique em **Database** no **Object Explorer**;
4. Clique em **Attach**;
5. Clique em **Add** e localize o arquivo **PEDIDOS_TABELAS.MDF** na pasta **BcoDadosPedidos_2016**;
6. Clique em **OK**;
7. Verifique se todos os arquivos para o banco de dados já estão marcados na lista de detalhes;
8. Clique em **OK** para finalizar o processo.

Caso ocorra algum problema ao anexar o banco, realize os passos a seguir:

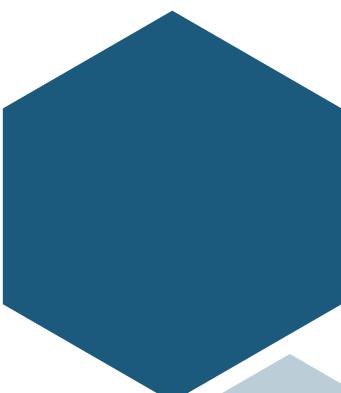
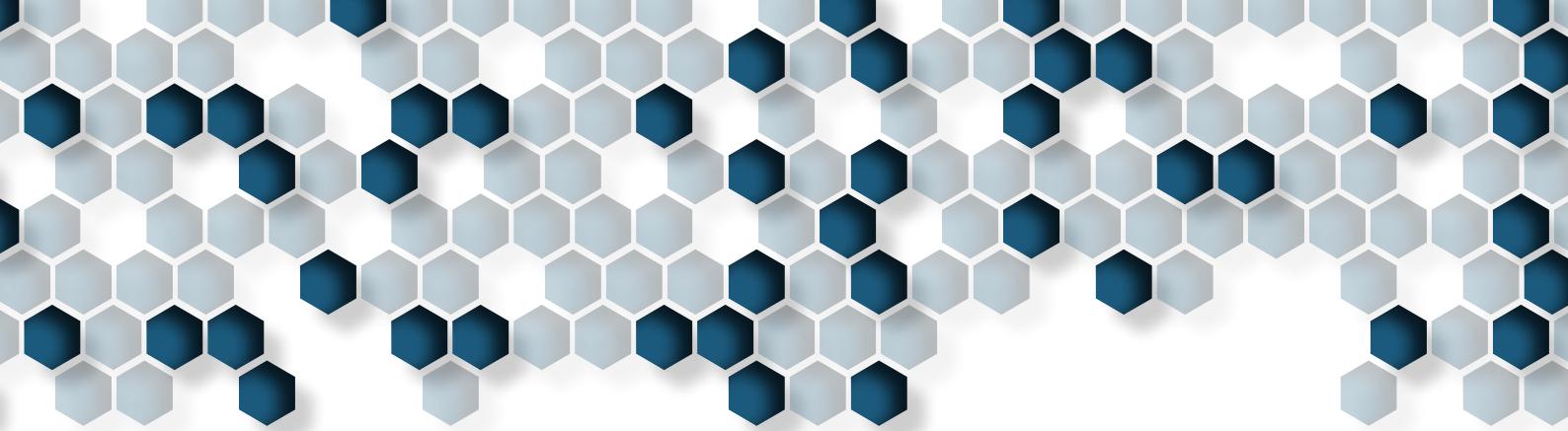
1. Abra o Windows Explorer;
2. Clique com o botão direito na pasta **BcoDadosPedidos_2016**;
3. Clique em **Propriedades** e, depois, na aba **Segurança**;
4. Clique no botão **Editar**;
5. Marque as opções **Controle total** e **Modificar** para o usuário **USER**.

Laboratório 3

A – Utilizando os objetos de catálogo

1. 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**.



Comandos adicionais



- ◆ SELECT;
- ◆ IIF/CHOOSE;
- ◆ LAG e LEAD;
- ◆ Paginação (FETCH e OFFSET);
- ◆ Funções úteis para campos IDENTITY;
- ◆ MERGE;
- ◆ Consultas cruzadas;
- ◆ Common Table Expressions (CTE);
- ◆ CROSS APPLY e OUTER APPLY.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 2 a 7.



1.1. SELECT

O comando **SELECT** é utilizado para consultar todos os dados de uma fonte de dados ou apenas uma parte específica deles.

A seguir, temos a sintaxe de **SELECT**:

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

Em que:

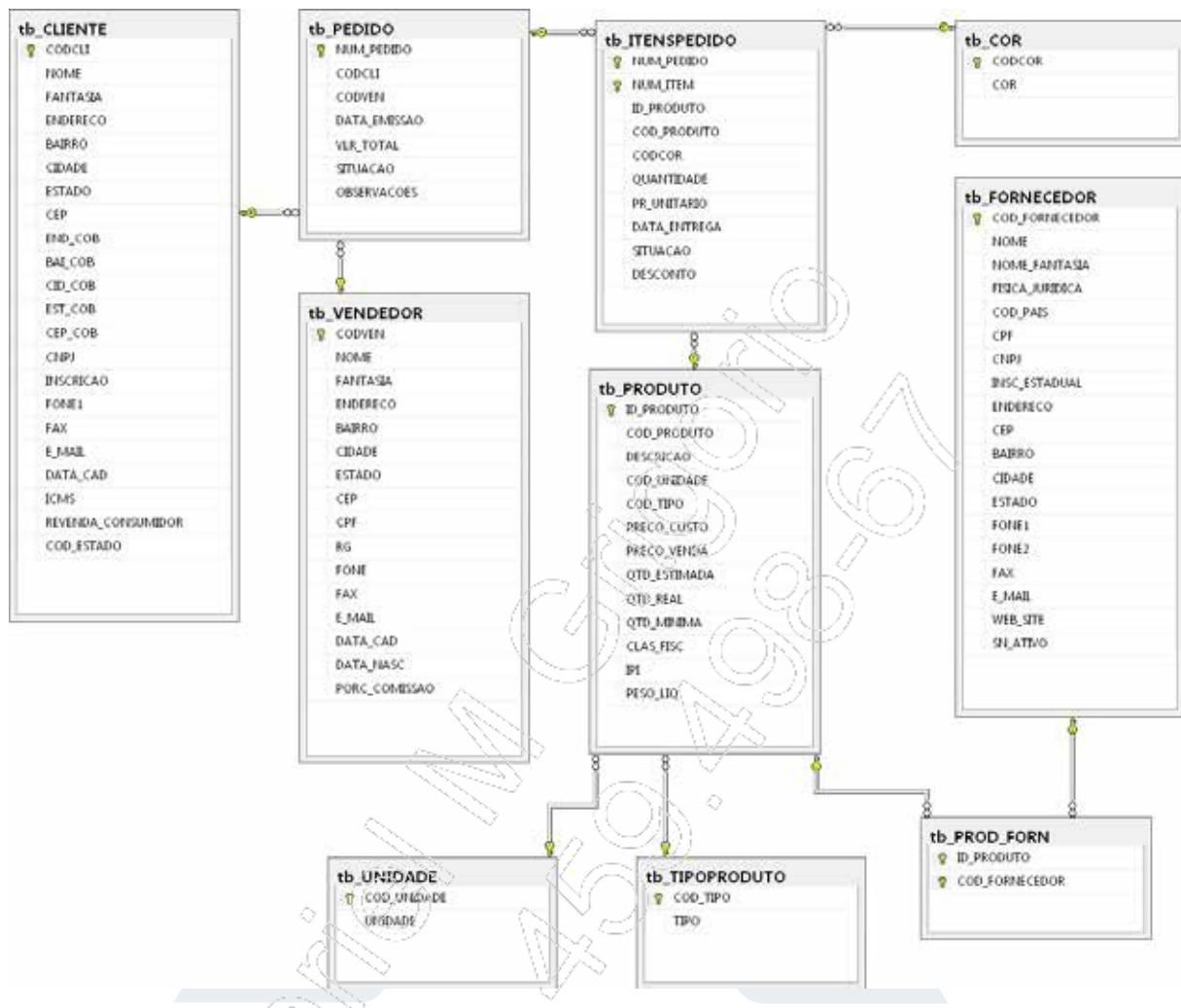
- **DISTINCT**: Palavra que especifica que apenas uma única instância de cada linha faça parte do conjunto de resultados. **DISTINCT** é utilizada com o objetivo de evitar a existência de linhas duplicadas no resultado da seleção;
- **[TOP (N) [PERCENT] [WITH TIES]]**: Especifica que apenas um primeiro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

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

Para a demonstração de como utilizar o comando SELECT, vamos primeiramente registrar no SQL Server o banco de dados PEDIDOS. A procedure SP_ATTACH_DB anexa um banco de dados:

```
EXEC sp_attach_db
@dbname      = 'PEDIDOS'
@filename1   = 'c:\Dados\PEDIDOS_TABELAS.mdf',
@filename2   = 'c:\Dados\PEDIDOS_INDICES.ndf',
@filename3   = 'c:\Dados\PEDIDOS_log.ldf'
```

Na imagem a seguir, temos as estruturas das tabelas desse banco de dados:





Para colocar em uso o banco de dados **PEDIDOS**, devemos utilizar a seguinte linha de código:

```
USE PEDIDOS;
```

A seguir, temos diversos exemplos do uso do comando **SELECT** para realizar consultas no banco de dados **PEDIDOS**:

- **Exemplo 1**

O código adiante consulta os funcionários do departamento de código **2** que ganham acima de 5000 reais:

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

- **Exemplo 2**

O código adiante consulta os funcionários do departamento de código **2** ou (**OR**) aqueles que ganham acima de 5000 reais:

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

- **Exemplo 3**

O código adiante consulta os funcionários com salário entre 3000 e 5000 reais:

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

O código a seguir pratica a mesma consulta feita pelo código anterior. A diferença é que aqui utilizamos a palavra **BETWEEN** em vez de **>=** e **<=**:

```
SELECT * FROM TB_EMPREGADO  
WHERE SALARIO BETWEEN 3000 AND 5000  
ORDER BY SALARIO
```

- **Exemplo 4**

A instrução **SELECT** a seguir consulta os funcionários com salário abaixo de 3000 ou acima de 5000 reais:

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

O mesmo resultado do código anterior pode ser obtido com a instrução **SELECT** a seguir:

```
SELECT * FROM TB_EMPREGADO  
WHERE SALARIO NOT BETWEEN 3000 AND 5000  
ORDER BY SALARIO
```

- **Exemplo 5**

O exemplo a seguir consulta os funcionários admitidos no ano 2000:

```
SELECT * FROM TB_Empregado  
WHERE DATA ADMISSAO BETWEEN '2000.1.1' AND '2000.12.31'  
ORDER BY DATA ADMISSAO
```

O mesmo resultado do código anterior pode ser obtido com a instrução **SELECT** a seguir:

```
SELECT * FROM TB_Empregado  
WHERE YEAR(DATA ADMISSAO) = 2000  
ORDER BY DATA ADMISSAO
```

- **Exemplo 6**

A instrução **SELECT** adiante consulta os funcionários cujo nome seja iniciado por **MARIA**:

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

- **Exemplo 7**

A instrução **SELECT** adiante consulta os funcionários cujo nome contenha **SOUZA**:

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

- **Exemplo 8**

A instrução **SELECT** a seguir consulta os funcionários cujo nome contenha **SOUZA** ou **SOUSA**:

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

- **Exemplo 9**

A instrução **SELECT** a seguir consulta os clientes dos estados do Amazonas, Paraná, Rio de Janeiro e São Paulo:

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

- **Exemplo 10**

Para sabermos a quantidade de dias desde que cada funcionário foi admitido, utilizamos a instrução **SELECT** com a função de data **GETDATE()**:

```
SELECT CODFUN, NOME,  
       CAST(GETDATE() - DATA ADMISSAO AS INT) AS DIAS NA EMPRESA  
FROM TB_EMPREGADO
```

- **Exemplo 11**

Vejamos como utilizar o **SELECT** com outras funções de data. No código a seguir, filtramos os funcionários admitidos em uma sexta-feira:

```
SELECT
    CODFUN, NOME, DATA_ADMISSAO,
    DATENAME(WEEKDAY, DATA_ADMISSAO) AS DIA_SEMANA,
    DATENAME(MONTH, DATA_ADMISSAO) AS MES
FROM TB_EMPREGADO
WHERE DATEPART(WEEKDAY, DATA_ADMISSAO) = 6
```

- **Exemplo 12**

O código adiante consulta duas tabelas na mesma instrução **SELECT**:

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

A escrita do código anterior pode ser simplificada se utilizarmos alias para os nomes das tabelas:

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

- **Exemplo 13**

O código a seguir consulta três tabelas na mesma instrução **SELECT**:

```
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO
FROM TB_EMPREGADO E
JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO
JOIN TB_CARGO C ON E.COD_CARGO = C.COD_CARGO;
```

A mesma consulta do código anterior pode ser feita da maneira mostrada a seguir:

```
SELECT
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO
FROM TB_DEPARTAMENTO D
JOIN TB_EMPREGADO E ON E.COD_DEPTO = D.COD_DEPTO
JOIN TB_CARGO C ON E.COD_CARGO = C.COD_CARGO;
```

Também podemos utilizar uma terceira maneira para realizar a mesma consulta anterior:

```
SELECT  
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO, C.CARGO  
FROM TB_CARGO C  
JOIN TB_EMPREGADO E ON E.COD_CARGO = C.COD_CARGO  
JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO;
```

- **Exemplo 14**

A instrução **SELECT** a seguir consulta os funcionários que não constam em departamento algum:

```
SELECT  
    E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_CARGO, D.DEPTO  
FROM TB_EMPREGADO E  
LEFT JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO  
WHERE D.COD_DEPTO IS NULL
```

- **Exemplo 15**

A instrução **SELECT** a seguir consulta os departamentos sem funcionários:

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

O mesmo resultado do código anterior pode ser obtido com a instrução a seguir:

```
SELECT * FROM TB_DEPARTAMENTO  
WHERE COD_DEPTO NOT IN (SELECT DISTINCT COD_DEPTO FROM TB_EMPREGADO  
WHERE COD_DEPTO IS NOT NULL)
```

- **Exemplo 16**

A instrução a seguir consulta os clientes que compraram em janeiro de 2015:

```
SELECT * FROM TB_CLIENTE  
WHERE CODCLI IN (SELECT CODCLI FROM TB_PEDIDO  
WHERE CODCLI = TB_CLIENTE.CODCLI AND  
DATA_EMISSAO BETWEEN '2015.1.1' AND  
'2015.1.31'))
```

- **Exemplo 17**

O código a seguir soma os salários de cada departamento:

```
SELECT E.COD_DEPTO, D.DEPTO, SUM( SALARIO ) AS TOT_SAL
FROM TB_EMPREGADO E JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_
DEPTO
GROUP BY E.COD_DEPTO, D.DEPTO
ORDER BY TOT_SAL
```

- **Exemplo 18**

A instrução a seguir consulta os departamentos que gastam mais de 15000 reais em salários:

```
SELECT E.COD_DEPTO, D.DEPTO, SUM( SALARIO ) AS TOT_SAL
FROM TB_EMPREGADO E JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_
DEPTO
GROUP BY E.COD_DEPTO, D.DEPTO
HAVING SUM(E.SALARIO) > 15000
ORDER BY TOT_SAL
```

- **Exemplo 19**

A instrução **SELECT** a seguir consulta os cinco departamentos que mais gastam com salários:

```
SELECT TOP 5 E.COD_DEPTO, D.DEPTO, SUM( E.SALARIO ) AS TOT_SAL
FROM TB_EMPREGADO E
JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO
GROUP BY E.COD_DEPTO, D.DEPTO
ORDER BY TOT_SAL DESC
```

1.2.IIF/CHOOSE

O comando **IIF** retorna um dos dois argumentos passados, dependendo do valor obtido em uma expressão booleana.

A sintaxe para utilização de IIF é a seguinte:

```
IIF(<expressao_booleana>, <valor_positivo>, <valor_negativo>)
```

O argumento **expressao_booleana** retorna TRUE ou FALSE. Caso o resultado da expressão seja TRUE, o argumento **valor_positivo** será retornado. Caso contrário, se o resultado da expressão for FALSE, o argumento **valor_negativo** será retornado.

No trecho a seguir, o valor do resultado da consulta será **VERDADEIRO**, pois a expressão booleana retorna o valor **TRUE**:

```
DECLARE @a INT = 15, @b INT = 10;
SELECT IIF (@a > @b, 'VERDADEIRO', 'FALSO') AS Resultado
```

O comando **CHOOSE** age com um índice em uma lista de valores. O argumento **índice** determina qual dos valores seguintes será retornado.

A sintaxe para a utilização do **CHOOSE** é a seguinte:

```
CHOOSE(<índice>, <valor_1>, <valor_2> [, <valor_n>] )
```

Em que:

- **<índice>**: É uma expressão que representa um índice na lista de valores passados como argumentos. Se o valor do argumento **<índice>** não for numérico do tipo INT, será necessária a conversão desse valor. Se o valor passado como índice for 0 ou um número negativo, o comando **CHOOSE** retornará o valor **NUL**L;
- **<valor_1>, <valor_2>, <valor_n>**: Representam a lista de valores em que é aceito qualquer tipo de dado. A lista permite a inserção de n valores.

Os exemplos a seguir demonstram a utilização de **CHOOSE**:

- **Exemplo 1**

No trecho seguinte, o resultado da expressão será o segundo valor passado na lista após o índice:

```
DECLARE @índice INT = 4;
SELECT CHOOSE (@índice/2, 1, 2, 3) AS Resultado
```

- **Exemplo 2**

Podemos utilizar o **CHOOSE** em conjunto com o comando **IIF**:

```
USE PEDIDOS

SELECT CODFUN, NOME, DATA ADMISSAO,
-- Substitui o S por SIM e o N por NÃO
    IIF(SINDICALIZADO = 'S', 'SIM', 'NÃO') AS SINDICALIZADO,
        -- Pega o número do dia da semana e devolve o nome que
        -- está na posição correspondente
            CHOOSE(DATEPART(WEEKDAY, DATA ADMISSAO),
                'DOMINGO',
                'SEGUNDA', 'TERÇA', 'QUARTA', 'QUINTA', 'SEXTA', 'SÁBADO')
                AS DIA_SEMANA
FROM TB_EMPREGADO
```

1.3. LAG e LEAD

Em uma consulta (**SELECT**), os operadores **LAG** e **LEAD** permitem recuperar um campo de N linhas anteriores à atual (**LAG**) ou posteriores à atual (**LEAD**).

```
LAG( coluna, offset[, default])
```

```
LEAD( coluna, offset[, default])
```

Em que:

- **coluna**: Nome da coluna que queremos recuperar;
- **offset**: Quantidades de linhas acima ou abaixo da atual;
- **default**: Valor a retornar caso a linha não exista. Se omitido, o valor retornado será **NULL**.

A seguir, temos um exemplo da utilização das funções analíticas **LAG** e **LEAD** para comparar os salários dos funcionários:

```
SELECT CODFUN, NOME, SALARIO,
LAG(SALARIO, 1, 0) OVER (ORDER BY CODFUN) AS SALARIO_ANTERIOR,
LEAD(SALARIO, 1, 0) OVER (ORDER BY CODFUN) AS PROXIMO_SALARIO
FROM TB_EMPREGADO
ORDER BY CODFUN
```

O resultado a seguir será mostrado:

	CODFUN	NOME	SALARIO	SALARIO_DE_CIMA	SALARIO_DE_BAIXO
1	1	OLAVO TRINDADE	3000.00	0.00	600.00
2	2	JOSE REIS	600.00	3000.00	2400.00
3	3	MARCELO SOARES	2400.00	600.00	600.00
4	4	PAULO CESAR JUNIOR	600.00	2400.00	1200.00
5	5	JOAO LIMA MACHADO DA SILVA	1200.00	600.00	4500.00
6	7	CARLOS ALBERTO SILVA	4500.00	1200.00	1200.00
7	8	ELIANE PEREIRA	1200.00	4500.00	800.00
8	9	RUDGE RAMOS SANTANA DA PENHA	800.00	1200.00	1200.00
9	10	MARIA CARMEM	1200.00	800.00	1200.00
10	11	FERNANDO OLIVEIRA	1200.00	1200.00	1200.00
11	12	JOAO ROBERTO OLIVEIRA	1200.00	1200.00	2400.00
12	13	OSMAR PRADO	2400.00	1200.00	1200.00

1.4. Paginação (FETCH e OFFSET)

Utilizando as cláusulas **FETCH** e **OFFSET**, é possível dividir os resultados das consultas em várias páginas numeradas. Com este novo recurso, podemos selecionar N linhas (**FETCH**) a partir de qualquer posição da tabela. A cláusula **ORDER BY** é necessária para a utilização das cláusulas **FETCH** e **OFFSET**.

Na instrução **SELECT** a seguir, visualizamos que os operadores dividem os dados em duas páginas, cada uma com 20 clientes:

```
SELECT * FROM TB_CLIENTE
ORDER BY CODCLI
OFFSET 0 ROWS FETCH NEXT 20 ROWS ONLY;

-- seleciona os próximos 20 clientes
SELECT * FROM TB_CLIENTE
ORDER BY CODCLI
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

O resultado a seguir será mostrado:

Result	Messages							
CODCLI	NOME	FANTASIA	ENDERECO	BAIRRO	CIDADE	ESTADO	CEP	END_C
13	BIASI MARQUETTI LTDA	BIASI	AV. MAL DEODORO, 14	V. MARILEIA	CASTANHAL	PA	68745000	AV. MA
14	BRAGA BRINDES LTDA	BRAGA BRINDES	R. SINAL CORREIA, 12	NULL	JUIZ DE FO...	MG	36020310	R. SINA
15	BRINDES TATUAPE LTDA ME	BRINDES TATUAPE	R. ENGENHEIRO PEGAD...	V. CARRAO	SAO PAULO	SP	03430000	R. ENG
16	BRINDY COM E IND. DE PROD. P/ PRESENTES LTDA	BRINDICE	AVALBERTO MIGUEL,172	CAMPINAS	GOIANIA	GO	74510010	AV. ALE
17	B.MATTOS & CIA. LTDA.	B.MATTOS	R. ENGENHEIRO SAMPAI...	IPIRANGA	SAO PAULO	SP	04261000	R. ENG
18	BANCO AMERICA DO SUL SA	BCO AMERICA DO	AL. RIBEIRAO PRETO, 82...	NULL	SAO PAULO	SP	01331001	AL. RIB
19	BRINDES GOLDEN FOX LTDA	BRINDES GOLDE	R. DO INVERNO, 129	JD. RIUCE	DIADEMA	SP	05990070	R. DO
20	BRESSER BRINDES LTDA	BRINDES BRESSER	R. 21 DE ABRIL, 250	BRAS	SAO PAULO	SP	03047000	R. 21
Query executed successfully.								
SQLSERVER (11.0 RTM) SA (55) PEDIDOS 00:00:00 20 rows								

É possível também especificar variáveis para os valores dos operadores **FETCH** e **OFFSET**. O código a seguir mostra como isso pode ser feito:

```
DECLARE @OFFSET INT = 0, @FETCH INT = 20;
SELECT * FROM TB_CLIENTE
ORDER BY CODCLI
OFFSET @OFFSET ROWS FETCH NEXT @FETCH ROWS ONLY;
```

Desta forma, com a ajuda de uma linguagem de programação, podemos facilmente criar uma interface gráfica para tornar dinâmica a visualização dos dados das consultas.

1.5. Funções úteis para campos IDENTITY

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
DBCC CHECKIDENT	DBCC CHECKIDENT('Tabela')
IDENTITYCOL	SELECT IDENTITYCOL FROM Tabela
IDENT_INCR	SELECT IDENT_INCR('Tabela')
IDENT_SEED	SELECT IDENT_SEED('Tabela')
@@IDENTITY	SELECT @@IDENTITY
SCOPE_IDENTITY	SELECT SCOPE_IDENTITY
IDENT_CURRENT	SELECT IDENT_CURRENT('Tabela')
SET IDENTITY_INSERT	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:

The screenshot shows the SSMS Results window with two tables displayed:

COD_PROF	NOME
1	MAGNO
2	AGNALDO
3	ROBERTO
4	RENATA
5	EDUARDO
6	MARCIO

COD_ALU	NOME
10	ZÉ DA SILVA
12	CARLOS SILVA
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:

The screenshot shows the SSMS Results window with the PROFESSOR table displayed, showing only five rows of data:

COD_PROF	NOME
1	MAGNO
3	ROBERTO
4	RENATA
5	EDUARDO
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	MAGNO
2	GODOFREDO
3	ROBERTO
4	RENATA
5	EDUARDO
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	7	MAGNO

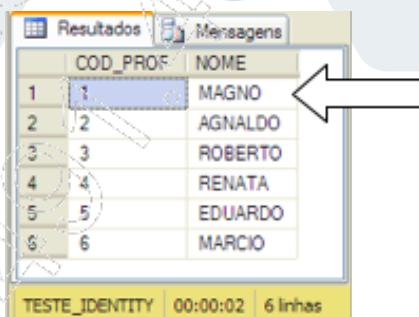
Podemos perceber que o código gerado (7) para MAGNO seguiu a mesma sequência anteriormente adotada. Agora, vamos excluir novamente os registros da tabela **PROFESSOR** e zerar, por meio de **DBCC CHECKIDENT**, o contador do **IDENTITY**:

```
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

TESTE_IDENTITY | 00:00:02 | 6 linhas

1.6. 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 PEDIDOS;
```

Em seguida, vamos gerar uma cópia da tabela **TB_EMPREGADO** chamada **EMP_TEMP** e testá-la:

```
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 COD_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 COD_DEPTO = 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.CODFUN = E.CODFUN -- 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
```

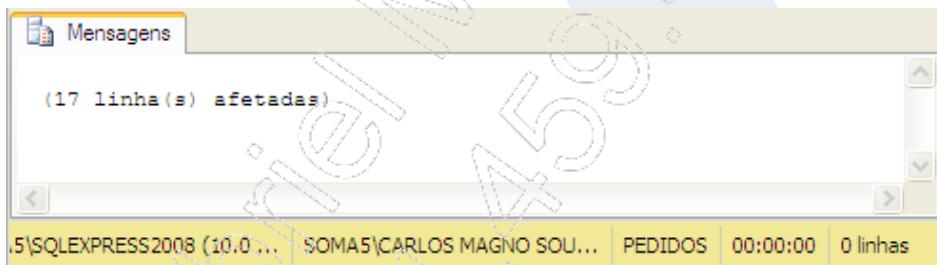
Também, desejamos o seguinte: Quando o registro não existir em **EMP_TEMP**, este terá o registro inexistente inserido. Para isso, utilizamos o código a seguir:

```
WHEN NOT MATCHED THEN  
  
    INSERT (CODFUN,NOME,COD_DEPTO,COD_CARGO,DATA_ADMISSAO,  
            DATA_NASCIMENTO,  
            SALARIO, NUM_DEPEND, SINDICALIZADO, OBS, FOTO)  
    VALUES (CODFUN,NOME,COD_DEPTO,COD_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 resultado obtido com a sequência de códigos anterior é mostrado adiante:



	CODFUN	NOME	COD_DEP...	SALARIO_ANTIGO	SALARIO_NOVO
1	2	JOSE REIS	2	600.00	900.00
2	9	RUDGE RAMOS SANTANA DA PENHA	2	800.00	1200.00
3	19	SEBASTIÃO SILVA	2	8300.00	12450.00
4	20	EURICO BRANDÃO	2	800.00	1200.00
5	25	MARIA DA PENHA	2	4500.00	6750.00
6	28	MARIANO DE OLIVEIRA	2	3330.00	4995.00
7	38	LUIS FERNANDO LEMOS	2	600.00	900.00
8	40	JOAQUIM ALBERTO	2	500.00	750.00

1.6.1. 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 COD_CARGO IN (3,5,7);
-- Altera os salários de EMP_TEMP dos funcionários do depto. 2
UPDATE EMP_TEMP SET SALARIO *= 1.2
WHERE COD_DEPTO = 2
```

Depois, acrescentaremos registros na tabela aí:

```
INSERT INTO EMP_TEMP
(NOME, COD_DEPTO, COD_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.CODFUN = E.CODFUN -- 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 (CODFUN,NOME,COD_DEPTO,COD_CARGO,DATA_ADMISSAO, DATA_
NASCIMENTO,
        SALARIO, NUM_DEPEND, SINDICALIZADO, OBS, FOTO)
VALUES (CODFUN,NOME,COD_DEPTO,COD_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
```

Em seguida, vamos gerar uma saída que mostre as alterações realizadas:

```
OUTPUT $ACTION AS [Ação], INSERTED.CODFUN AS [Código Após],
       DELETED.CODFUN 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.

1.7. Consultas cruzadas

A criação de uma consulta cruzada consiste em rotacionar os resultados de uma consulta de maneira que as linhas sejam exibidas verticalmente e as colunas, horizontalmente.

Para que possamos criar consultas cruzadas, será necessário contar com a ajuda da plataforma, caso contrário, teríamos uma tarefa trabalhosa e passível de erros. Por isso, os operadores **PIVOT** e **UNPIVOT** são disponibilizados pelo SQL, sendo que o primeiro tem como finalidade facilitar o processo de criação de consultas cruzadas, e o segundo tem a capacidade de reverter os dados que já tiverem passado pelo processo feito pelo **PIVOT**.

Imaginemos uma situação em que precisássemos totalizar as vendas de cada vendedor em cada um dos meses do ano. Bastaria utilizar um **GROUP BY**, como mostra o exemplo a seguir:

```
SELECT CODVEN, MONTH(DATA_EMISSAO) AS MES,
       YEAR(DATA_EMISSAO) AS ANO,
       SUM(VLR_TOTAL) AS TOT_VENDIDO
  FROM TB_PEDIDO
 WHERE YEAR(DATA_EMISSAO) = 2014
 GROUP BY
       CODVEN, MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
 ORDER BY 1,2,3
```

O resultado obtido seria semelhante à imagem mostrada a seguir. Seria um resultado muito longo e, como podemos notar, mostraria apenas os totais do vendedor de código 1. Os outros vendedores não apareceriam na mesma tela, o que significa que precisaríamos rolar o grid para consultar tudo.

	CODVEN	MES	ANO	TOT_VENDIDO
1	1	1	2014	73444.52
2	1	2	2014	61123.23
3	1	3	2014	59815.03
4	1	4	2014	69612.15
5	1	5	2014	53147.85
6	1	6	2014	75630.23
7	1	7	2014	51593.74
8	1	8	2014	56174.34
9	1	9	2014	17849.42
10	1	10	2014	28491.52
11	1	11	2014	44370.92
12	1	12	2014	29908.86
13	2	1	2014	111177.53
14	2	2	2014	64058.59
15	2	3	2014	79435.16
16	2	4	2014	66724.83
17	2	5	2014	47390.07
18	2	6	2014	72522.24

Vejamos, então, o resultado seguinte, que é mais compacto:

CODVEN	MES1	MES2	MES3	MES4	MES5	MES6	MES7	MES8	MES9	MES10	MES11	MES12
1	73444.52	61123.23	59815.03	69612.15	53147.85	75630.23	51593.74	56174.34	17849.42	28491.52	44370.92	29908.86
2	111177.53	64058.59	79435.16	66724.83	47390.07	72522.24	51839.08	60837.07	41643.92	33518.25	45212.35	52893.10
3	74104.79	59699.02	64064.99	51760.71	63538.69	60886.21	82300.62	54767.02	53137.94	45989.69	47667.24	45171.16
4	74255.05	67774.55	67498.30	60758.37	78109.67	80365.60	76414.80	59402.64	35246.49	44611.69	26125.06	35753.63
5	80295.20	56184.31	75553.14	58400.34	57958.24	71775.86	77848.27	68584.78	53232.24	31022.90	39659.19	41442.59
6	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	40924.02	45439.26	52872.04
7	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	67108.15	59831.71	37354.18
8	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	12520.38	47158.34	48277.21

O que aparecia como conteúdo da coluna **MES**, no primeiro exemplo, passou a ser título de coluna. Esta é a finalidade da função **PIVOT()**.

1.7.1. PIVOT ()

O operador **PIVOT** tem como função girar uma expressão table-valued, transformando os valores únicos provenientes de uma coluna da expressão em várias colunas na saída. Além disso, realiza agregações em todos os valores de colunas restantes que sejam necessários ao final da saída.

PIVOT transforma valores em colunas e é bastante utilizado para gerar relatórios de tabela cruzada.

A seguir, temos diversos exemplos do uso de **PIVOT**. Neste primeiro, temos o total vendido por cada vendedor em cada um dos meses de 2014.

```
SELECT CODVEN, [1] AS MES1, [2] AS MES2, [3] AS MES3,
       [4] AS MES4, [5] AS MES5, [6] AS MES6,
       [7] AS MES7, [8] AS MES8, [9] AS MES9,
       [10] AS MES10, [11] AS MES11, [12] AS MES12
  FROM (SELECT CODVEN, VLR_TOTAL, MONTH(DATA_EMISSAO) AS MES
        FROM TB_PEDIDO
       WHERE YEAR(DATA_EMISSAO) = 2014) P
 PIVOT( SUM(VLR_TOTAL) FOR MES IN ([1], [2], [3], [4], [5], [6],
                                     [7], [8], [9], [10], [11],
                                     [12])) ) AS PVT
 ORDER BY 1
```

Agora temos a mesma informação incluindo também o nome do vendedor:

```
SELECT CODVEN, NOME, [1] AS MES1, [2] AS MES2, [3] AS MES3, [4] AS
MES4, [5] AS MES5,
       [6] AS MES6, [7] AS MES7, [8] AS MES8, [9] AS MES9,
       [10] AS MES10,
       [11] AS MES11, [12] AS MES12
  FROM (SELECT P.CODVEN, V.NOME, P.VLR_TOTAL, MONTH(P.DATA_EMISSAO) AS
MES
        FROM TB_PEDIDO P JOIN TB_VENDEDOR V ON P.CODVEN = V.CODVEN
       WHERE YEAR(P.DATA_EMISSAO) = 2014) P
 PIVOT( SUM(VLR_TOTAL) FOR MES IN ([1],[2],[3],[4],[5],[6],[7],
[8],[9],[10],[11],[12])) ) AS PVT
 ORDER BY 1
```

Já neste outro exemplo, temos o total comprado por cada cliente em cada um dos meses de 2014:

```
SELECT CODCLI, [1] AS MES1, [2] AS MES2, [3] AS MES3, [4] AS MES4,
       [5] AS MES5, [6] AS MES6, [7] AS MES7, [8] AS MES8,
       [9] AS MES9, [10] AS MES10, [11] AS MES11,
       [12] AS MES12
  FROM (SELECT CODCLI, VLR_TOTAL, MONTH(DATA_EMISSAO) AS MES
        FROM TB_PEDIDO
       WHERE YEAR(DATA_EMISSAO) = 2014) P
     PIVOT( SUM(VLR_TOTAL) FOR MES IN ([1],[2],[3],[4],[5],[6],[7],[8],
      ,[9],[10],[11],[12])) AS PVT
    ORDER BY 1
```

Podemos também ver o quanto cada cliente comprou em cada ano:

```
SELECT CODCLI, NOME, [2008] AS '2008', [2009] AS '2009',
       [2010] AS '2010', [2011] AS '2011',
       [2012] AS '2012', [2013] AS '2013',
       [2014] AS '2014', [2015] AS '2015'
  FROM (SELECT P.CODCLI, C.NOME, P.VLR_TOTAL,
              YEAR(P.DATA_EMISSAO) AS ANO
         FROM TB_PEDIDO P
        JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI) AS X
     PIVOT( SUM(VLR_TOTAL) FOR ANO IN ([2008],[2009],[2010],[2011],
      ,[2012],[2013],[2014],[2015])) AS Y
```

No exemplo seguinte, temos o total vendido de cada produto em cada um dos meses de 2014:

```
SELECT ID_PRODUTO, [1] AS MES1, [2] AS MES2, [3] AS MES3,
       [4] AS MES4, [5] AS MES5, [6] AS MES6,
       [7] AS MES7, [8] AS MES8, [9] AS MES9,
       [10] AS MES10, [11] AS MES11, [12] AS MES12
  FROM (SELECT I.ID_PRODUTO, I.QUANTIDADE*I.PR_UNITARIO AS VALOR,
              MONTH(P.DATA_EMISSAO) AS MES
         FROM TB_PEDIDO P
        JOIN TB_ITENSPEDIDO I ON P.NUM_PEDIDO = I.NUM_PEDIDO
       WHERE YEAR(P.DATA_EMISSAO) = 2014) I
     PIVOT( SUM(VALOR) FOR MES IN ([1],[2],[3],[4],[5],[6],[7],[8],[9],
      ,[10],[11],[12])) AS PVT
    ORDER BY 1
```

Neste outro exemplo, temos o total vendido em cada um dos meses de cada ano:

```
SELECT ANO, [1] AS MES1, [2] AS MES2, [3] AS MES3, [4] AS MES4,
       [5] AS MES5, [6] AS MES6, [7] AS MES7, [8] AS MES8,
       [9] AS MES9, [10] AS MES10,
       [11] AS MES11, [12] AS MES12
  FROM (SELECT YEAR(DATA_EMISSAO) AS ANO, VLR_TOTAL, MONTH(DATA_
EMISSAO) AS MES
        FROM TB_PEDIDO) P
PIVOT( SUM(VLR_TOTAL) FOR MES IN ([1],[2],[3],[4],[5],[6],[7],[8]
,[9],[10],[11],[12])) AS PVT
ORDER BY 1
```

Por fim, temos uma situação igual à do exemplo anterior, porém, com disposição inversa:

```
SELECT MES, [2012] AS ANO_2012, [2013] AS ANO_2013, [2014] AS
ANO_2014, [2015] AS ANO_2015
  FROM ( SELECT MONTH(DATA_EMISSAO) AS MES, VLR_TOTAL, YEAR(DATA_
EMISSAO) AS ANO FROM TB_PEDIDO ) P
PIVOT( SUM(VLR_TOTAL) FOR ANO IN ([2012],[2013],[2014],[2015]) )
AS PVT
ORDER BY 1
```

A informação que define os agrupamentos nas colunas, aquela que se transforma em título de coluna, precisa ser uma informação numérica. No exemplo a seguir, tentaremos contar quantos funcionários temos em cada departamento com nomes começando com as letras A, B, C, D e E:

```
-- NÃO FUNCIONA
SELECT COD_DEPTO, ['A'], ['B'], ['C'], ['D'], ['E']
  FROM (SELECT CODFUN,COD_DEPTO, LEFT(NOME,1) AS LETRA FROM TB_
EMPREGADO) AS P
PIVOT (COUNT(CODFUN) FOR LETRA IN ('A', 'B', 'C', 'D',
['E'])) AS PVT
```

Ao executar o trecho anterior, verificamos que não ocorre nenhum erro, mas não obtemos o resultado esperado. Se substituirmos a letra pelo seu código, verificamos que o resultado correto será mostrado:

```
-- FUNCIONA
SELECT COD_DEPTO, [65], [66], [67], [68], [69]
FROM (SELECT CODFUN,COD_DEPTO, ASCII(UPPER(LEFT(NOME,1))) AS LETRA
      FROM TB_EMPREGADO) AS P
PIVOT (COUNT(CODFUN) FOR LETRA IN ([65], [66], [67], [68], [69])) AS PVT
```

COD_DEPTO	65	66	67	68	69
NULL	0	0	0	0	0
1	1	0	1	0	0
2	0	0	0	0	1
3	4	0	0	0	0
4	0	0	2	0	0
5	3	0	0	0	0
6	2	0	0	0	1
7	0	0	1	0	0
8	0	0	0	0	0
9	1	0	0	0	0
11	0	0	1	0	0
12	0	0	0	0	0
14	0	0	0	0	0

1.7.2. UNPIVOT()

O operador **UNPIVOT** realiza uma ação contrária à de **PIVOT**, ou seja, transforma colunas de uma expressão table-valued em valores de colunas. Porém, **UNPIVOT** não oferece como resultado uma expressão table-valued idêntica à original, uma vez que as linhas já foram fundidas. Isso quer dizer que o resultado obtido com **UNPIVOT** é diferente da entrada com a qual **PIVOT** começou a lidar.

Consideremos o seguinte exemplo, que cria uma tabela com a quantidade de pessoas que frequentam o cinema, em diferentes sessões (horários), em cada dia da semana:

```
CREATE TABLE FREQ_CINEMA
( DIA_SEMANA TINYINT,
  SEC_14HS INT,
  SEC_16HS INT,
  SEC_18HS INT,
  SEC_20HS INT,
  SEC_22HS INT )
```

```
INSERT FREQ_CINEMA VALUES ( 1, 80, 100, 130, 90, 70 )
INSERT FREQ_CINEMA VALUES ( 2, 20, 34, 75, 50, 30 )
INSERT FREQ_CINEMA VALUES ( 3, 25, 40, 80, 70, 25 )
INSERT FREQ_CINEMA VALUES ( 4, 30, 45, 70, 50, 30 )
INSERT FREQ_CINEMA VALUES ( 5, 35, 40, 60, 60, 40 )
INSERT FREQ_CINEMA VALUES ( 6, 25, 34, 70, 90, 110 )
INSERT FREQ_CINEMA VALUES ( 7, 30, 80, 130, 150, 180 )

SELECT * FROM FREQ_CINEMA
```

O resultado é o seguinte:

	DIA_SEMANA	SEC_14HS	SEC_16HS	SEC_18HS	SEC_20HS	SEC_22HS
1	1	80	100	130	90	70
2	2	20	34	75	50	30
3	3	25	40	80	70	25
4	4	30	45	70	50	30
5	5	35	40	60	60	40
6	6	25	34	70	90	110
7	7	30	80	130	150	180

Agora, vamos utilizar o operador **UNPIVOT** para converter as sessões de cinema em valores de coluna:

```
SELECT DIA_SEMANA, HORARIO, QTD_PESSOAS
FROM
(
  SELECT DIA_SEMANA, SEC_14HS, SEC_16HS, SEC_18HS, SEC_20HS, SEC_22HS
  FROM FREQ_CINEMA
) P
UNPIVOT ( QTD_PESSOAS FOR HORARIO IN (SEC_14HS, SEC_16HS, SEC_18HS,
SEC_20HS, SEC_22HS)) AS UP
```

O resultado é o seguinte:

	DIA_SEMANA	HORARIO	QTD_PESSOAS
1	1	SEC_14HS	80
2	1	SEC_16HS	100
3	1	SEC_18HS	130
4	1	SEC_20HS	90
5	1	SEC_22HS	70
6	2	SEC_14HS	20
7	2	SEC_16HS	34
8	2	SEC_18HS	75
9	2	SEC_20HS	50
10	2	SEC_22HS	30
11	3	SEC_14HS	25
12	3	SEC_16HS	40
13	3	SEC_18HS	80
14	3	SEC_20HS	70
15	3	SEC_22HS	25
16	4	SEC_14HS	30
17	4	SEC_16HS	45
18	4	SEC_18HS	70
19	4	SEC_20HS	50
20	4	SEC_22HS	30

1.8. Common Table Expressions (CTE)

Chamamos de **common table expression** (CTE) o conjunto de resultados temporários que se define no escopo de execução de uma instrução **SELECT**, **INSERT**, **UPDATE**, **DELETE** ou **CREATE VIEW**. As common table expressions não são armazenadas como objetos e conservam-se apenas durante a consulta, sendo, por estes motivos, parecidas com as tabelas derivadas.

O uso de common table expressions está voltado para as seguintes finalidades:

- Criar uma consulta recursiva;
- Substituir uma view nas situações em que seu uso geral não é exigido;
- Habilitar o agrupamento por uma coluna derivada de um subquery escalar;
- Referenciar diversas vezes na mesma instrução a tabela resultante.

Podendo ser definidas em rotinas determinadas pelo usuário (como funções, procedures armazenadas, triggers ou views), as common table expressions garantem os benefícios da boa legibilidade e da fácil manutenção de queries complexas.

Uma CTE é composta de um nome de expressão (que a representa), de uma lista de colunas – opcional – e de uma consulta (que a define). A lista de colunas é opcional quando, na definição da consulta, forem fornecidos nomes diferentes para todas as colunas resultantes.

As CTEs já definidas podem ser referenciadas nas instruções **SELECT**, **INSERT**, **UPDATE** ou **DELETE**, da mesma maneira como referenciamos tabelas ou views em tais instruções.

Os procedimentos para a criação e utilização de uma common table expression são: escolher um nome e uma lista de colunas, criar a consulta **SELECT** da expressão e, então, usá-la em uma consulta.

O resultado a seguir mostra o maior pedido (de maior valor) vendido em cada um dos meses de 2014:

	MES	ANO	MAIOR_PEDIDO
1	1	2006	17998.70
2	2	2006	6785.85
3	3	2006	13233.50
4	4	2006	22871.25
5	5	2006	11134.00
6	6	2006	5287.36
7	7	2006	5221.90
8	8	2006	5050.74
9	9	2006	5156.75
10	10	2006	4958.93
11	11	2006	5161.89
12	12	2006	5318.93

Esse resultado foi gerado pelo seguinte **SELECT**, o qual utiliza a cláusula **GROUP BY**:

```
SELECT MONTH( DATA_EMISSAO ) AS MES,
       YEAR( DATA_EMISSAO ) AS ANO,
       MAX( VLR_TOTAL ) AS MAIOR_PEDIDO
  FROM TB_PEDIDO
 WHERE YEAR(DATA_EMISSAO) = 2014
 GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
 ORDER BY 1;
```

Vamos supor que desejamos incluir uma quarta coluna no resultado para informar o número do pedido tido como o maior do mês. No entanto, não é possível incluí-la na consulta **SELECT** exibida porque estaríamos quebrando o agrupamento. Uma solução para este caso seria criar uma view com essa consulta e depois fazer uma associação (join) entre a view e a tabela **PEDIDOS**. Mas, se assim for feito, a view não terá utilidade posterior, já que ela só funciona para o ano de 2014. O mais indicado, então, é utilizar uma Common Table Expression (CTE):

```
WITH CTE( MES, ANO, MAIOR_PEDIDO )
AS
(
    -- Membro âncora
    SELECT MONTH( DATA_EMISSAO ) AS MES,
           YEAR( DATA_EMISSAO ) AS ANO,
           MAX( VLR_TOTAL ) AS MAIOR_PEDIDO
      FROM TB_PEDIDO
     WHERE YEAR(DATA_EMISSAO) = 2014;
     GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
)

    -- Utilização da CTE fazendo JOIN com a tabela TB_PEDIDO
    SELECT CTE.MES, CTE.ANO, CTE.MAIOR_PEDIDO, P.NUM_PEDIDO
      FROM CTE JOIN TB_PEDIDO P ON CTE.MES = MONTH(P.DATA_EMISSAO) AND
                                   CTE.ANO = YEAR(P.DATA_EMISSAO) AND
                                   CTE.MAIOR_PEDIDO = P.VLR_TOTAL;
```

1.8.1. CTE Recursiva

Com as common table expressions, também é possível escrever consultas recursivas, as quais são formadas por três elementos: a invocação da rotina, a invocação recursiva da rotina e verificação de finalização.

Uma common table expression recursiva é capaz de retornar diversas linhas, diferente de uma rotina recursiva de outras linguagens, que retorna um valor escalar.

É importante lembrar que a estrutura de uma common table expression recursiva precisa possuir, no mínimo, um **membro âncora** e um **membro recursivo**. O membro âncora é executado somente uma vez. Já o membro recursivo executa a própria CTE.

Enquanto criamos a CTE, podemos modificar sua consulta **SELECT**. Para isso, basta criar a consulta do membro âncora, adicionar o operador **UNION ALL** e, então, criar a consulta do membro recursivo que faça autorreferência à common table expression.

Vejamos:

```
-- Contador  
WITH CONTADOR ( N )  
AS  
(  
    -- Membro âncora  
    SELECT 1  
    UNION ALL  
    -- Membro recursivo  
    SELECT N+1 FROM CONTADOR WHERE N < 100  
)  
-- Execução da CTE  
SELECT * FROM CONTADOR;
```

O resultado do código anterior é uma sequência de números inteiros de 1 até 100.

A seguir, temos uma CTE que retorna potências de 5 como resultado:

```
-- Potências de 5  
  
WITH POTENCIAS_DE_5 ( EXPOENTE, POTENCIA )  
AS  
(  
    SELECT 1,5  
    UNION ALL  
    SELECT EXPOENTE+1, POTENCIA * 5  
    FROM POTENCIAS_DE_5  
    WHERE EXPOENTE < 10  
)  
SELECT * FROM POTENCIAS_DE_5;
```

O resultado é o seguinte:

EXPOENTE	POTENCIA
1	5
2	25
3	125
4	625
5	3125
6	15625
7	78125
8	390625
9	1953125
10	9765625

Adiante, temos um exemplo de CTE que calcula a evolução do total a ser pago para uma dívida de R\$ 1000,00, considerando uma taxa de juros de 5% ao mês (juros compostos):

```
WITH JUROS( MES, VALOR )
AS
(
    SELECT 0, CAST(1000 AS NUMERIC(10,2))
    UNION ALL
    SELECT MES + 1,
           CAST(VALOR * 1.05 AS NUMERIC(10,2))
    FROM JUROS WHERE MES < 12
)
SELECT * FROM JUROS;
```

O resultado é o seguinte:

	MES	VALOR
1	0	1000.00
2	1	1050.00
3	2	1102.50
4	3	1157.63
5	4	1215.51
6	5	1276.29
7	6	1340.10
8	7	1407.11
9	8	1477.47
10	9	1551.34
11	10	1628.91
12	11	1710.36
13	12	1795.88

No exemplo a seguir, temos uma CTE que aplica a sequência de Fibonacci. Descrita primeiramente por Leonardo de Pisa (também conhecido como Fibonacci), essa sequência é iniciada por 0 e 1, sendo que o elemento seguinte será sempre a soma dos dois últimos números. Como os dois números iniciais são 0 e 1, o número seguinte é 1, ou seja, $0 + 1 = 1$. Agora, temos 0, 1 e 1. Como continuação da sequência, temos 2, isto é, $1 + 1 = 2$, e assim por diante. Vejamos:

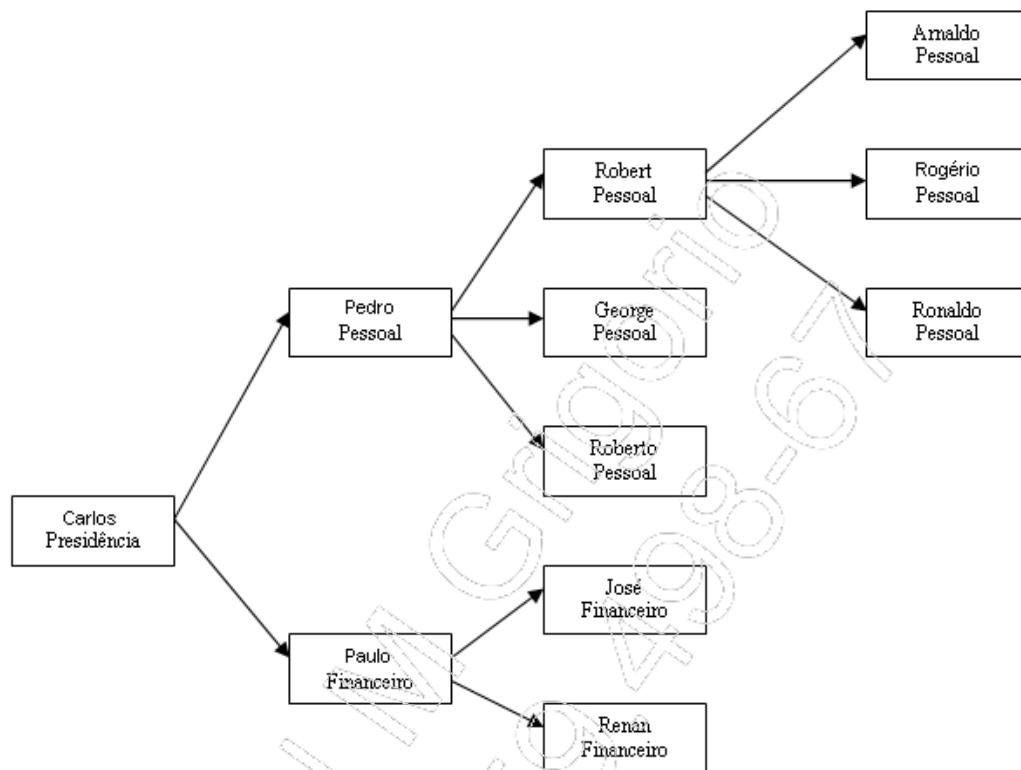
```
WITH FIBO( N1, N2, PROX)
AS
(
    SELECT 0,1,1
    UNION ALL
    SELECT N2, PROX, N2+PROX FROM FIBO WHERE PROX < 10000
)
SELECT * FROM FIBO;
```

O resultado é o seguinte:

	N1	N2	PROX
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
6	5	8	13
7	8	13	21
8	13	21	34
9	21	34	55
10	34	55	89
11	55	89	144
12	89	1...	233
13	1...	2...	377
14	2...	3...	610
15	3...	6...	987
16	6...	9...	1597
17	9...	1...	2584
18	1...	2...	4181
19	2...	4...	6765
20	4...	6...	10946

A sequência de Fibonacci é esta: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946 etc.

Vejamos outro exemplo. Na tabela **TB_EMPREGADO**, do banco de dados **PEDIDOS**, existe um campo chamado **COD_SUPERVISOR**, que indica quem é o supervisor do empregado, que está cadastrado na mesma tabela **TB_EMPREGADO**. No entanto, a quantidade de subníveis do organograma é variável, como mostra a figura a seguir:



Para gerar um relatório como o mostrado adiante, ou criamos uma stored procedure ou, então, criamos uma CTE recursiva. Não há como gerá-lo com view ou com join apenas:

	Nome Funcionário	Nível	Nome Supervisor	Anvare
1	CARLOS ALBERTO SILVA	1	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA
2	JORGE DOS SANTOS ROCHA JUNIOR	1	JORGE DOS SANTOS ROCHA JUNIOR	JORGE DOS SANTOS ROCHA JUNIOR
3	SEVERINO CARLOS MACIEIRA	1	SEVERINO CARLOS MACIEIRA	SEVERINO CARLOS MACIEIRA
4	PAULO CESAR JUNIOR	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← PAULO CESAR JUNIOR
5	SEBASTIÃO SILVA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← SEBASTIÃO SILVA
6	ANA MARIA OLIVEIRA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← ANA MARIA OLIVEIRA
7	CARLOS MAGNO P SOUZA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← CARLOS MAGNO P SOUZA
8	MANOEL SANTOS	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← MANOEL SANTOS
9	MARIANA DA SILVA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← MARIANA DA SILVA
10	ARLINDO SOARES	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← ARLINDO SOARES
11	ALBERTO HELENA SILVA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← ALBERTO HELENA SILVA
12	LÚCIO MITSUI	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← LÚCIO MITSUI
13	JOÃO JOSÉ DE SOUZA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← JOÃO JOSÉ DE SOUZA
14	PEDRO PAULO SOUZA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA
15	ROBERTO MARILDO	3	PEDRO PAULO SOUZA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO
16	JORGE ROBERTO SOUZA	3	PEDRO PAULO SOUZA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← JORGE ROBERTO SOUZA
17	ROBERTO CARLOS DA SILVA	3	PEDRO PAULO SOUZA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO CARLOS DA SILVA
18	ARNALDO MOURA	4	ROBERTO MARILDO	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ARNALDO MOURA
19	ROGÉRIO FREITAS	4	ROBERTO MARILDO	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ROGÉRIO FREITAS
20	RONALDO MATIAS	4	ROBERTO MARILDO	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← RONALDO MATIAS
21	CASSIANO OLIVEIRA	5	ROGÉRIO FREITAS	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ROGÉRIO FREITAS ← CASSIANO OLIV.
22	JOSÉ CARLOS SILVA	5	ROGÉRIO FREITAS	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ROGÉRIO FREITAS ← JOSÉ CARLOS
23	ROBERTO PINHEIRO	6	CASSIANO OLIVEIRA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ROGÉRIO FREITAS ← CASSIANO OLIV.
24	JOÃO CARLOS DE OLIVEIRA	6	ARNALDO MOURA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ARNALDO MOURA ← JOÃO CARLOS
25	JOSE CARLOS MOREIRA	5	ARNALDO MOURA	CARLOS ALBERTO SILVA ← PEDRO PAULO SOUZA ← ROBERTO MARILDO ← ARNALDO MOURA ← JOSE CARLOS M.

Comandos adicionais

Aulas 2 a 7

Para obter o resultado anterior, primeiramente criamos a seguinte CTE:

```
WITH CTE( CODFUN, NOME, COD_DEPTO, CODSUP, NOME_SUP )  
AS  
(  
    -- Membro âncora  
    SELECT CODFUN, NOME, COD_DEPTO, COD_SUPERVISOR, NOME  
    FROM TB_Empregado WHERE COD_SUPERVISOR = 0  
    UNION ALL  
    -- Membro recursivo  
    SELECT E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_SUPERVISOR, CTE.  
    NOME  
    FROM TB_Empregado E JOIN CTE ON E.COD_SUPERVISOR = CTE.  
    CODFUN  
)  
    -- Execução da CTE  
    SELECT CTE.CODFUN AS [Código], CTE.NOME AS [Funcionário],  
        D.DEPTO AS [Departamento], CTE.NOME_SUP AS [Nome Supervisor]  
    FROM CTE JOIN TB_EMPREGADO E ON CTE.CODFUN = E.CODFUN  
        JOIN TB_DEPARTAMENTO D ON E.COD_DEPTO = D.COD_DEPTO  
    ORDER BY CTE.COD_DEPTO;
```

No código anterior, o membro recursivo utiliza a própria CTE para acessar os subníveis. Estamos subutilizando o poder da recursão. A consulta **SELECT** adiante faria a mesma coisa:

```
SELECT E.CODFUN, E.NOME, E.COD_DEPTO, E.COD_SUPERVISOR, CTE.NOME  
FROM TB_Empregado E JOIN TB_EMPREGADO CTE ON E.COD_SUPERVISOR = CTE.  
CODFUN  
ORDER BY COD_DEPTO;
```

Em seguida, vamos acrescentar uma coluna **NIVEL**, para sabermos em qual nível do organograma o funcionário está localizado. Para executar essa tarefa, utilizamos recursão:

```
WITH CTE( CODFUN, NOME, NIVEL, COD_DEPTO, CODSUP, NOME_SUP )  
AS  
(  
    -- Membro âncora  
    SELECT CODFUN, NOME, 1, COD_DEPTO, COD_SUPERVISOR, NOME  
    FROM TB_Empregado WHERE COD_SUPERVISOR = 0  
    UNION ALL  
    -- Membro recursivo  
    SELECT E.CODFUN, E.NOME,  
        CTE.NIVEL+1, E.COD_DEPTO, E.COD_SUPERVISOR, CTE.NOME  
    FROM TB_Empregado E JOIN CTE ON E.COD_SUPERVISOR = CTE.  
    CODFUN  
)  
    -- Execução da CTE  
    SELECT CTE.CODFUN AS [Código], CTE.NOME AS [Funcionário], CTE.NIVEL  
        AS [Nível], CTE.NOME_SUP AS [Nome Supervisor]  
    FROM CTE;
```

O resultado é o seguinte:

	Código	Funcionário	Nível	Nome Supervisor
1	7	CARLOS ALBERTO SILVA	1	CARLOS ALBERTO SILVA
2	44	JORGE DOS SANTOS ROCHA JUNIOR	1	JORGE DOS SANTOS ROCHA JUNIOR
3	68	SEVERINO CARLOS MACIEIRA	1	SEVERINO CARLOS MACIEIRA
4	4	PAULO CESAR JUNIOR	2	CARLOS ALBERTO SILVA
5	19	SEBASTIÃO SILVA	2	CARLOS ALBERTO SILVA
6	26	ANA MARIA OLIVEIRA	2	CARLOS ALBERTO SILVA
7	30	CARLOS MAGNO P SOUZA	2	CARLOS ALBERTO SILVA
8	31	MANOEL SANTOS	2	CARLOS ALBERTO SILVA
9	35	MARIANA DA SILVA	2	CARLOS ALBERTO SILVA
10	55	ARLINDO SOARES	2	CARLOS ALBERTO SILVA
11	58	ALBERTO HELENA SILVA	2	CARLOS ALBERTO SILVA
12	64	LÚCIO MITSUI	2	CARLOS ALBERTO SILVA
13	67	JOÃO JÓSE DE SOUZA	2	CARLOS ALBERTO SILVA
14	70	PEDRO PAULO SOUZA	2	CARLOS ALBERTO SILVA
15	49	ROBERTO MARILDO	3	PEDRO PAULO SOUZA
16	51	JORGE ROBERTO SOUZA	3	PEDRO PAULO SOUZA
17	72	ROBERTO CARLOS DA SILVA	3	PEDRO PAULO SOUZA

Por fim, vamos gerar a árvore que mostra toda a hierarquia ascendente de um funcionário:

```
WITH CTE( CODFUN, NOME, NIVEL, COD_DEPTO, CODSUP, NOME_SUP, ARVORE )
AS
(
    -- Membro âncora
    SELECT CODFUN, NOME, 1, COD_DEPTO, COD_SUPERVISOR, NOME,
           CAST(NOME AS VARCHAR(MAX))
    FROM TB_Empregado WHERE COD_SUPERVISOR = 0
    UNION ALL
    -- Membro recursivo
    SELECT E.CODFUN, E.NOME,
           CTE.NIVEL+1, E.COD_DEPTO, E.COD_SUPERVISOR, CTE.NOME,
           CTE.ARVORE + ' <- ' + E.NOME
    FROM TB_Empregado E JOIN CTE ON E.COD_SUPERVISOR = CTE.
CODFUN
)
-- Execução da CTE
SELECT CTE.CODFUN AS [Código], CTE.NOME AS [Funcionário], CTE.NIVEL
AS [Nível],
       CTE.NOME_SUP AS [Nome Supervisor], CTE.ARVORE AS [Árvore]
FROM CTE;
```

O resultado final é o seguinte:

	Funcionário	Nível	Nome Supervisor	Anexo
1	CARLOS ALBERTO SILVA	1	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA
2	JORGE DOS SANTOS ROCHA JUNIOR	1	JORGE DOS SANTOS ROCHA JUNIOR	JORGE DOS SANTOS ROCHA JUNIOR
3	SEVERINO CARLOS MACIEIRA	1	SEVERINO CARLOS MACIEIRA	SEVERINO CARLOS MACIEIRA
4	PAULO CESAR JUNIOR	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- PAULO CESAR JUNIOR
5	SEBASTIÃO SILVA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- SEBASTIÃO SILVA
6	ANA MARIA OLIVEIRA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- ANA MARIA OLIVEIRA
7	CARLOS MAGNO P SOUZA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- CARLOS MAGNO P SOUZA
8	MARCELO SANTOS	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- MARCELO SANTOS
9	MARIANA DA SILVA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- MARIANA DA SILVA
10	ARLINDO SOARES	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- ARLINDO SOARES
11	ALBERTO HELENA SILVA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- ALBERTO HELENA SILVA
12	LÓCIO MITSUI	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- LÓCIO MITSUI
13	JOÃO JOSÉ DE SOUZA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- JOÃO JOSÉ DE SOUZA
14	PEDRO PAULO SOUZA	2	CARLOS ALBERTO SILVA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA
15	ROBERTO MARILDO	3	PEDRO PAULO SOUZA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO
16	JORGE ROBERTO SOUZA	3	PEDRO PAULO SOUZA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- JORGE ROBERTO SOUZA
17	ROBERTO CARLOS DA SILVA	3	PEDRO PAULO SOUZA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO CARLOS DA SILVA
18	ARNALDO MOURA	4	ROBERTO MARILDO	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ARNALDO MOURA
19	ROGÉRIO FREITAS	4	ROBERTO MARILDO	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ROGÉRIO FREITAS
20	RONALDO MATIAS	4	ROBERTO MARILDO	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- RONALDO MATIAS
21	CASSIANO OLIVEIRA	5	ROGÉRIO FREITAS	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ROGÉRIO FREITAS <- CASSIANO OLIVEIRA
22	JOSÉ CARLOS SILVA	5	ROGÉRIO FREITAS	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ROGÉRIO FREITAS <- JOSÉ CARLOS SILVA
23	ROBERTO PINHEIRO	6	CASSIANO OLIVEIRA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ROGÉRIO FREITAS <- CASSIANO OLIVEIRA
24	JOÃO CARLOS DE OLIVEIRA	5	ARNALDO MOURA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ARNALDO MOURA <- JOÃO CARLOS DE OLIVEIRA
25	JOSÉ CARLOS MOREIRA	5	ARNALDO MOURA	CARLOS ALBERTO SILVA <- PEDRO PAULO SOUZA <- ROBERTO MARILDO <- ARNALDO MOURA <- JOSÉ CARLOS MOREIRA

1.9. CROSS APPLY e OUTER APPLY

O **CROSS APPLY** tem funcionalidade parecida com o **INNER JOIN**, que permite a relação entre consultas. Uma das vantagens é a correlação com uma subconsulta ou função tabular, permitindo o retorno de várias colunas. Este recurso pode aumentar a performance em consultas mais complexas. Vejamos o exemplo a seguir:

A área de negócio solicitou uma consulta que apresente as seguintes informações: código e nome do cliente, número, valor e data de emissão.

```
--Consulta apresentando o código e nome do cliente, Valor total e
data da -- emissão.
SELECT C.CODCLI, C.NOME,P.NUM_PEDIDO,P.VLR_TOTAL, P.DATA_EMISSAO
FROM TB_CLIENTE AS C
JOIN TB_PEDIDO AS P ON P.CODCLI = C.CODCLI
ORDER BY C.CODCLI,P.NUM_PEDIDO
```

Após apresentar o resultado, a área também solicitou que fossem apresentadas, na mesma consulta, as seguintes informações: quantidade de pedidos, maior e menor pedido e a data da última compra.

Para realizar essa consulta, podemos utilizar o recurso de subconsulta:

```
--Adicionando as colunas quantidade de pedidos, maior e menor valor
de compra e a última data de pedido.
```

```
SELECT C.CODCLI, C.NOME,P.NUM_PEDIDO,P.VLR_TOTAL, P.DATA_EMISSAO,
(SELECT COUNT(*) FROM TB_PEDIDO
 WHERE CODCLI = C.CODCLI) AS QTD_PED,
```

```
(SELECT MAX(VLR_TOTAL) FROM TB_PEDIDO
     WHERE CODCLI = C.CODCLI) AS MAIOR_VALOR,
(SELECT MIN(VLR_TOTAL) FROM TB_PEDIDO
     WHERE CODCLI = C.CODCLI) AS MENOR_VALOR,
(SELECT MAX(DATA_EMISSAO) FROM TB_PEDIDO
     WHERE CODCLI = C.CODCLI) AS ULTIMA_COMPRA
FROM TB_CLIENTE AS C
JOIN TB_PEDIDO AS P ON P.CODCLI = C.CODCLI
ORDER BY C.CODCLI,P.NUM_PEDIDO
```

A mesma consulta pode ser realizada por meio do **CROSS APPLY**:

```
--Utilizando o CROSS APPLY

SELECT      C.CODCLI,C.NOME,P.NUM_PEDIDO,P.VLR_TOTAL, P.DATA_EMISSAO,
           CR.QTD_PED, CR.MAIOR_VALOR , CR.MENOR_VALOR ,
           CR.DATAMAXIMA
  FROM TB_CLIENTE AS C
 JOIN TB_PEDIDO AS P ON P.CODCLI = C.CODCLI

CROSS APPLY

( SELECT      COUNT(*) AS QTD_PED,
              MAX(VLR_TOTAL) AS MAIOR_VALOR,
              MIN(VLR_TOTAL) AS MENOR_VALOR,
              MAX(DATA_EMISSAO) AS DATAMAXIMA
    FROM TB_PEDIDO AS P
   WHERE C.CODCLI = P.CODCLI ) AS CR
 ORDER BY C.CODCLI,P.NUM_PEDIDO
```

Note que, ao adicionar um filtro por ano, a subconsulta torna-se mais complexa:

```
-- Consulta com filtro dos pedidos de 2014.

SELECT  C.CODCLI, C.NOME,P.NUM_PEDIDO,P.VLR_TOTAL, P.DATA_EMISSAO,
(SELECT COUNT(*) FROM TB_PEDIDO
     WHERE YEAR(DATA_EMISSAO) = 2014
     AND CODCLI = C.CODCLI) AS QTD_PED,
(SELECT MAX(VLR_TOTAL) FROM TB_PEDIDO
     WHERE YEAR(DATA_EMISSAO) = 2014
     AND CODCLI = C.CODCLI) AS MAIOR_VALOR,
(SELECT MIN(VLR_TOTAL) FROM TB_PEDIDO
     WHERE YEAR(DATA_EMISSAO) = 2014
     AND CODCLI = C.CODCLI) AS MENOR_VALOR,
(SELECT MAX(DATA_EMISSAO) FROM TB_PEDIDO
     WHERE YEAR(DATA_EMISSAO) = 2014
     AND CODCLI = C.CODCLI) AS ULTIMA_COMPRA
FROM TB_CLIENTE AS C
JOIN TB_PEDIDO AS P ON P.CODCLI = C.CODCLI
WHERE YEAR(DATA_EMISSAO) = 2014
```

A mesma consulta pode ser realizada por meio do **CROSS APPLY**:

```
--UTILIZANDO O CROSS APPLY

SELECT C.CODCLI, C.NOME, P.NUM_PEDIDO, P.VLR_TOTAL, P.DATA_EMISSAO,
       CR.QTD_PED, CR.MAIOR_VALOR, CR.MENOR_VALOR, CR.DATAMAXIMA
  FROM TB_CLIENTE AS C
 JOIN TB_PEDIDO AS P ON P.CODCLI = C.CODCLI
 CROSS APPLY
  ( SELECT COUNT(*) AS QTD_PED,
            MAX(VLR_TOTAL) AS MAIOR_VALOR,
            MIN(VLR_TOTAL) AS MENOR_VALOR,
            MAX(DATA_EMISSAO) AS DATAMAXIMA
      FROM TB_PEDIDO AS PC
     WHERE C.CODCLI = PC.CODCLI AND YEAR(PC.DATA_EMISSAO) = 2014 )AS CR
 WHERE YEAR(P.DATA_EMISSAO) = 2014
 ORDER BY C.CODCLI, P.NUM_PEDIDO
```

Já o **OUTER APPLY** possui uma característica parecida com a do **LEFT JOIN**. Vejamos o exemplo adiante:

A área de RH solicitou uma listagem com o nome do departamento e o nome do funcionário. Podemos efetuar a consulta com **LEFT JOIN**:

```
SELECT D.DEPTO, E.NOME
  FROM TB_DEPARTAMENTO AS D
 LEFT JOIN TB_EMPREGADO AS E ON E.COD_DEPTO = D.COD_DEPTO
 ORDER BY 2
```

A mesma consulta pode ser realizada com **CROSS APPLY**:

```
SELECT D.DEPTO, CA.NOME
  FROM TB_DEPARTAMENTO AS D
 CROSS APPLY
  (SELECT E.NOME FROM TB_EMPREGADO AS E WHERE E.COD_DEPTO = D.COD_DEPTO ) AS CA
 ORDER BY 2
```

Para apresentar todos os departamentos, mesmo que não possuam funcionários, podemos realizar essa consulta com **LEFT JOIN**:

```
SELECT D.DEPTO, E.NOME
  FROM TB_DEPARTAMENTO AS D
 LEFT JOIN TB_EMPREGADO AS E ON E.COD_DEPTO = D.COD_DEPTO
 ORDER BY 2
```

A mesma consulta pode ser realizada com **OUTER APPLY**:

```
SELECT D.DEPTO , CA.NOME
FROM TB_DEPARTAMENTO AS D
OUTER APPLY
(SELECT E.NOME FROM TB_EMPREGADO AS E WHERE E.COD_DEPTO = D.COD_
DEPTO ) AS CA
ORDER BY 2
```



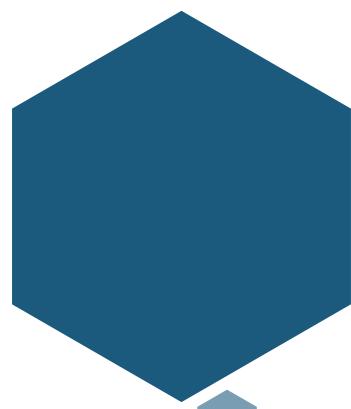
Pontos principais

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

- O comando **SELECT** é utilizado para consultar todos os dados de uma fonte de dados ou apenas uma parte específica deles;
- 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;
- A instrução **MERGE** realiza 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;
- Como resultado de uma consulta a uma tabela de banco de dados, podemos especificar um conjunto de linhas que será retornado, a partir da primeira linha selecionada. Para isso, utilizamos a cláusula **TOP**. Podemos especificar uma quantidade de linhas ou uma porcentagem de linhas a serem retornadas;
- A criação de uma consulta cruzada consiste em rotacionar os resultados de uma consulta de maneira que as linhas sejam exibidas verticalmente e as colunas, horizontalmente;
- O operador **PIVOT** tem como função girar uma expressão table-valued, transformando os valores únicos provenientes de uma coluna da expressão em várias colunas na saída. Além disso, realiza agregações em todos os valores de colunas restantes que sejam necessários ao final da saída;
 -
- O operador **UNPIVOT** realiza uma ação contrária à de **PIVOT**, ou seja, transforma colunas de uma expressão table-valued em valores de colunas. Porém, uma vez que as linhas já foram fundidas, o resultado obtido com **UNPIVOT** é diferente da entrada com a qual **PIVOT** começou a lidar;
- **Common table expression (CTE)** é o conjunto de resultados temporário que se define no escopo de execução de uma instrução **SELECT**, **INSERT**, **UPDATE**, **DELETE** ou **CREATE VIEW**. As common table expressions não são armazenadas como objetos e conservam-se apenas durante a consulta, sendo, por esses motivos, parecidas com as tabelas derivadas;
- **CROSS APPLY** e **OUTER APPLY** possuem funcionalidades parecidas com **INNER JOIN** e **LEFT JOIN**. Em determinados casos, possuem uma melhor performance.

SQL 2016 - Programação em T-SQL (online)

Gabriel M Grigorio
497.459.498-67



Comandos adicionais

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 2 a 7.



1. Verifique a consulta a seguir:

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

Quantos registros serão retornados?

- a) Todos os empregados com salário menor que 3000 e maior que 5000.
- b) Todos os empregados com salário menor e igual a 3000 e maior e igual a 5000.
- c) A sintaxe está errada e retornará um erro.
- d) Todos os empregados.
- e) Nenhum registro.

2. Verifique a consulta a seguir:

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

Qual a finalidade da consulta?

- a) Retornar os clientes.
- b) Retornar os clientes que não compraram em janeiro de 2014.
- c) Retornar os pedidos dos clientes.
- d) Retornar os clientes que compraram em janeiro de 2014.
- e) A sintaxe está errada.

3. Com relação às funcionalidades do SQL, qual afirmação está incorreta?

- a) LEAD recupera um valor de campo em N linhas posteriores ao registro atual.
- b) LAG recupera um valor de campo em N linhas anteriores ao registro atual.
- c) Podemos realizar paginação do retorno de dados com FETCH e OFFSET.
- d) O comando CHOOSE retorna o valor de uma lista conforme o argumento.
- e) Não podemos utilizar um comando IIF em uma instrução TSQL.

4. Qual a função do MERGE?

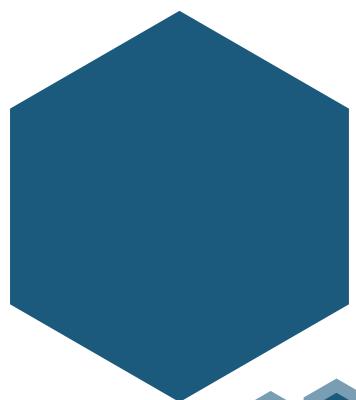
- a) Igualar uma tabela alvo com informações de uma tabela fonte.
- b) Igualar uma tabela fonte com informações de uma tabela alvo.
- c) Apagar registros.
- d) Inserir e apagar registros.
- e) Este comando não tem funcionalidade dentro das áreas de negócio.

5. Podemos transformar uma consulta rotacionando seu resultado. Como realizamos esta ação?

- a) Utilizando o UNPIVOT.
- b) Por meio do comando PIVOT ou UNPIVOT.
- c) Utilizando o PIVOT.
- d) Utilizando tabela temporária com UNION.
- e) O melhor é utilizar outra ferramenta, como o EXCEL.

SQL 2016 - Programação em T-SQL (online)

Gabriel M Grigorio
497.459.498-67



Comandos adicionais



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 2 a 7.



Laboratório 1

A - Revisando comandos

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Liste todos os pedidos (**TB_PEDIDO**) do vendedor ‘**MARCELO**’ em janeiro de 2014;
3. Liste todos os pedidos de janeiro de 2014, mostrando o nome do cliente e do vendedor em cada pedido;
4. Liste todos os itens de **TB_PEDIDO** de janeiro de 2014 com desconto superior a 7%. Devem ser mostrados **NUM_PEDIDO**, **DESCRICAO** do produto, **NOME** do cliente, nome do **VENDEDOR** e **QUANTIDADE** vendida;
5. Calcule a quantidade de pedidos cadastrados em janeiro de 2014 e o maior e o menor valor de pedido (**VLR_TOTAL**);
6. Calcule o valor total vendido (soma de **TB_PEDIDO.VLR_TOTAL**) e o valor da comissão (soma de **TB_PEDIDO.VLR_TOTAL * TB_VENDEDOR.PORC_COMISSAO /100**) de cada vendedor em janeiro de 2014;
7. Liste os nomes e o total comprado pelos 10 clientes que mais compraram em janeiro de 2014;
8. Liste os nomes dos clientes que não compraram em janeiro de 2014;
9. Reajuste os preços de venda de todos os produtos com **COD_TIPO = 5**, de modo que fiquem 20% acima do preço de custo;

! Para calcular a porcentagem de 20%, devemos, na tabela **TB_PRODUTO**, atribuir a **PRECO_VENDA** o valor **PRECO_CUSTO** multiplicado por 1.2. Lembrando que esse cálculo só deve ser feito para produtos com **COD_TIPO = 5**.

10. Reajuste os preços de venda de todos os produtos com descrição do tipo igual a **REGUA**, de modo que fiquem 40% acima do preço de custo.

! Para calcular a porcentagem de 40%, devemos, na tabela **TB_PRODUTO**, atribuir a **PRECO_VENDA** o valor **PRECO_CUSTO** multiplicado por 1.4. Lembrando que esse cálculo só deve ser feito para produtos com **TB_TIPOPRODUTO.TIPO** igual a **REGUA**. Porém, para realizar esse cálculo, devemos fazer um **JOIN** da tabela **TB_PRODUTO** com a tabela **TB_TIPOPRODUTO**.

Laboratório 2

A – Trabalhando com opções adicionais e com o comando MERGE

1. Coloque em uso o banco de dados **PEDIDOS**;
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**.

B – Trabalhando com consultas cruzadas

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Desenvolva uma consulta que retorne o Estado do cliente, Valor total de compra e o mês de todas as compras do ano de 2013;
3. Realize uma rotação da consulta para apresentar a soma dos meses por Estado;
4. Utilizando a consulta anterior, acrescente a Cidade do cliente.

C – Utilizando Common Table Expressions (CTE)

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Utilizando uma consulta com expressão CTE, apresente: mês, ano, o maior pedido, número do pedido e o nome do cliente, mas somente do ano de 2013.

D – Utilizando APPLY

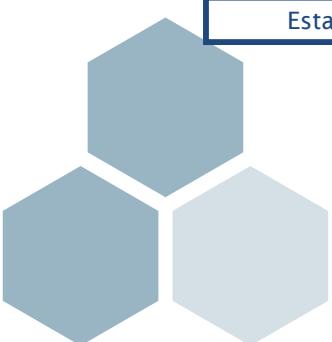
1. Coloque em uso o banco de dados **PEDIDOS**;
2. Realize uma consulta que apresente as informações adiante:
 - Código, nome, número do pedido, valor total e estado do cliente;
 - Quantos pedidos os cliente realizou;
 - A soma do valor total dos pedidos do cliente;
 - A quantidade dos pedidos do estado do cliente;
 - A soma do valor total dos pedidos do estado do cliente;
 - O maior e menor valor dos pedidos do estado do cliente;
 - A data da última compra do pedido do estado do cliente;
 - Percentual da compra sobre o total do mês: ($VLR_Total / Total\ do\ mês *100$);
 - Quantidade de dias entre data de emissão com a última compra;
 - Os registros devem ser apenas de Janeiro de 2014;
 - Ordene pelo nome do cliente e número de pedido.



Opções de definição de tabelas

- ◆ Tipos de dados;
- ◆ Sequências;
- ◆ Sinônimos;
- ◆ Trabalhando com objetos binários;
- ◆ FILETABLE;
- ◆ Colunas computadas.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 8 a 10.



1.1. Introdução

No desenvolvimento de um projeto, é necessária a definição de como armazenar as informações. Para isso, utilizamos o recurso de tabelas. Existem recursos que ampliam as possibilidades desse tipo de armazenamento. Nesta leitura, iremos explorar esses recursos, que ajudarão a incrementar o desenvolvimento de tabelas.

1.2. Tipos de dados

Para a construção do modelo que armazenará os dados de uma aplicação, é necessário planejamento, visto que os tipos de dados determinam os tipos de valores que podem ser inseridos em uma coluna. Determinando um tipo de valor específico, podemos manter a integridade e a consistência dos dados inseridos em uma coluna.

1.2.1. Tipos de dados nativos (Built-in)

Os **tipos de dados nativos** são aqueles fornecidos pelo próprio SQL Server. As principais categorias desses dados estão relacionadas a seguir:

Numéricos exatos
BIGINT
INT
SMALLINT
TINYINT
BIT
DECIMAL
NUMERIC
MONEY
SMALLMONEY
Números aproximados
FLOAT
REAL
Alfanuméricos
CHAR
VARCHAR
TEXT

Opções de definição de tabelas

Aulas 8 a 10

Alfanuméricos com UNICODE
NCHAR
NVARCHAR
NTEXT
Binários
BINARY
VARBINARY
IMAGE
Outros tipos de dados
CURSOR
HIERARCHYID
SPATIAL TYPES
SQL_VARIANT
TABLE
TIMESTAMP
UNIQUEIDENTIFIER

Há, ainda, os **tipos de dados baseados em data e hora** e o **tipo de dados XML**:

- **Tipos de dados baseados em data e hora**

Existem seis tipos de dados no SQL Server relacionados a data e hora. São eles: **time**, **date**, **datetime**, **datetime2**, **smalldatetime** e **datetimeoffset**. Com exceção de **time** e **date**, que se referem, respectivamente, a hora e data, os outros tipos de dados comportam os dois tipos de informações. O tipo de dado **datetimeoffset** também comporta data e hora, porém reconhece diferenças de fuso horário.

Se, ao utilizar **datetime**, **datetime2** ou **smalldatetime**, especificarmos apenas um valor para data, um valor zero será aplicado para a hora. Se for especificado apenas um valor para hora, o valor aplicado pelo SQL Server para data será **1900-01-01**.

! É preciso ressaltar que, como **datetime**, **datetime2** e **smalldatetime** comportam dois tipos de informação, data e hora, algumas vezes isso pode gerar problemas.

Para realizar consultas sobre valores de data e hora, podem ser utilizados operadores numéricos (=, <, >), bem como funções de data e hora. Ao utilizar **datetime**, **datetime2** e **datetimeoffset**, as condições da query devem incluir tanto valores de data quanto valores de hora.

As diferenças entre os tipos de dados de data e hora estão relacionadas na tabela a seguir:

Tipo de dado	Formato	Extensão	Precisão	bytes	Precisão
time	hh:mm:ss[.nnnnnnnn]	De 00:00:00.0000000 a 23:59:59.9999999	100 nanossegundos	3 a 5	Sim
date	YYYY-MM-DD	De 0001-01-01 a 9999-12-31	1 dia	3	Não
datetime	YYYY-MM-DD	De 1753-01-01 a 9999-12-31	0.00333 segundos	8	Não
	hh:mm:ss[.nnn]				
datetime2	YYYY-MM-DD	De 0001-01-01	100 nanossegundos	6 a 8	Sim
	hh:mm:ss[.nnnnnnnn]	00:00:00.0000000 a 9999-12-31			
		23:59:59.9999999			
smalldatetime	YYYY-MM-DD	De 1900-01-01 a 2079-06-06	1 minuto	4	Não
	hh:mm:ss				
datetimeoffset	YYYY-MM-DD	De 0001-01-01	100 nanossegundos	8 a 10	Sim
	hh:mm:ss[.nnnnnnnn]	00:00:00.0000000 a 9999-12-31			
	[+ -]hh:mm	23:59:59.9999999 (em UTC, Universal Time, Coordinated)			

É fundamental que as configurações de formato e linguagem dos valores de data e hora a serem inseridos em uma tabela ou view estejam corretas. Por isso, é recomendável a utilização de formatos que independem da linguagem. O uso de formato dependente de linguagem não é aconselhável mesmo no uso de instruções **SET**. Para isso, pode-se usar o formato ISO **datetime**, que sempre funciona, independentemente da linguagem e formato de data, e cujos valores são representados da seguinte maneira: **yyyymmdd [hh:mm:ss]** ou **yyyy-mm-ddThh:mm:ss**.

- **Tipo de dados XML**

O tipo de dados XML é um tipo de dado nativo cuja função é armazenar e interagir com dados XML. Assim, podemos armazenar documentos e fragmentos XML em um banco de dados SQL. Esses fragmentos correspondem a instâncias XML que não possuem um determinado elemento de nível superior. Elas são armazenadas ao criarmos variáveis e colunas com tipo de dados XML.

Ao criar uma tabela, o XML pode ser usado como um tipo de coluna. Também pode servir como um tipo de variável, um tipo de parâmetro ou um tipo de função de retorno. Podemos, ainda, associar uma coleção de esquema XML com uma coluna, parâmetro ou variável de tipo XML.

Para trabalhar com os dados suportados pelo XML, dispomos de cinco métodos:

- **Query**: Permite instruções XQuery utilizadas para retornar elementos de dados XML;
- **Exists**: Especifica se uma instrução XQuery retorna resultados;
- **Value**: Retorna um valor de tipo SQL de uma instância XML;
- **Modify**: Especifica instruções de atualização para modificação de dados XML;
- **Nodes**: Divide o XML em diversas linhas.

Os métodos **Query**, **Exists**, **Value** e **Nodes** suportam instruções XQuery. Já o método **Modify** suporta instruções XML Data Modification Language.

1.2.2. 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.

1.2.2.1. 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_udt>
    FROM <tipo_builtin> <tipo_null>;
```

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 novo 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.

1.2.3. 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.

1.2.4. CREATE RULE

Essa instrução cria um objeto chamado **rule** (ou regra), que é responsável por especificar os valores que podem ser inseridos em uma coluna à qual está associado, ou seja, criar regras de validação. O rule também pode ser associado a um tipo de dados de usuário. Não se pode associar um rule a tipos de dados do sistema.

Somente uma regra de validação pode ser associada a uma coluna ou tipo de dados de usuário. A uma coluna podemos associar uma regra e restrições de verificação, que serão avaliadas.

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

```
CREATE RULE <nome_regra> AS <condicao>
```

Em que:

- **nome_regra**: É o nome da nova regra. Os nomes de regras devem ser compatíveis com as regras para identificadores;
- **condicao**: Representa a condição ou condições que definem a regra. Uma regra pode ser definida por uma expressão válida em uma cláusula **WHERE**, e pode incluir elementos como operadores aritméticos, operadores relacionais e predicados. Não se pode utilizar funções definidas por usuário, mas é possível incluir funções internas que não façam referências a objetos de banco de dados. A regra não pode incluir referência a colunas ou outros objetos de banco de dados.

! Este é um recurso que será removido em uma versão futura do SQL Server. É recomendável, portanto, evitar a sua utilização em novos desenvolvimentos.

1.2.5. sp_bindrule

Essa stored procedure serve para associar uma regra de validação a um tipo de dados definido pelo usuário.

A seguir, apresentamos a sintaxe de **sp_bindrule**:

```
SP_BINDRULE <nome_regra>, <tipo_udt>
```

Em que:

- **nome_regra**: É o nome da regra criada;
- **tipo_udt**: É o nome do tipo de dados do usuário ao qual a regra será associada.

Ao associar uma nova regra a uma coluna ou tipo de dados definido pelo usuário, a regra antiga é substituída, sem necessidade de desassociá-la. Se uma regra é associada a uma coluna, as informações relacionadas são adicionadas à tabela **sys.columns**. Se for associada a um tipo de dados definido pelo usuário, as informações relacionadas são adicionadas à tabela **sys.types**.

! Este é um recurso que será removido em uma versão futura do SQL Server. É recomendável, portanto, evitar a sua utilização em novos desenvolvimentos.

1.2.6. sp_unbindrule

A função dessa stored procedure é desassociar uma regra de uma coluna ou de um tipo de dados de usuário.

A sintaxe de **sp_unbindrule** é a seguinte:

```
SP_UNBINDRULE <tipo_udt>, ['futureonly']
```

Em que:

- **tipo_udt**: É o nome do tipo de dados definido pelo usuário;
- **'futureonly'**: Se utilizado, afetará somente as novas utilizações de **<tipo_udt>**. Os objetos que já foram criados utilizando esse tipo de dados não serão afetados.

Ao desassociar uma regra de uma coluna, as informações da associação são removidas da tabela **sys.columns**, e, ao desassociar uma regra de um tipo de dados de usuário, as informações são removidas da tabela **sys.types**.

 Este é um recurso que será removido em uma versão futura do SQL Server. É recomendável, portanto, evitar a sua utilização em novos desenvolvimentos.

1.2.7. CREATE DEFAULT

Essa instrução cria um objeto **default** (ou padrão). O default atribui automaticamente um valor a uma coluna caso o registro inserido pelo usuário venha a ocultar esse conteúdo. É uma forma de assegurar a integridade dos dados.

Para associarmos um default a uma coluna que possua uma regra de validação, é necessário que ele esteja de acordo com a regra. Caso isso não ocorra, uma mensagem de erro será exibida ao tentar inserir o default.

A sintaxe da instrução **CREATE DEFAULT** é a seguinte:

```
CREATE DEFAULT <nome_def> AS <valor_def>
```

Em que:

- **nome_def**: É o nome do objeto que define o valor default;
- **valor_def**: É a expressão que define o valor default.

 Este é um recurso que será removido em uma versão futura do SQL Server. É recomendável, portanto, evitar a sua utilização em novos desenvolvimentos.

1.2.8. sp_bindefault

Essa stored procedure associa um valor default a uma coluna ou tipo de dados de usuário.

Um default, porém, não pode ser associado a um tipo de dado CLR definido pelo usuário, nem a um tipo de dados de sistema do SQL Server. Se um default não é compatível com uma coluna, ao tentar inseri-lo será exibida uma mensagem de erro.

A sintaxe de **sp_bindefault** é a seguinte:

```
SP_BINDEFAULT <nome_def> , <tipo_udt>
```

Em que:

- **nome_def**: É o nome do objeto que define o valor default;
- **tipo_udt**: É o nome do tipo de dados de usuário.

Quando um default é associado a uma coluna, as informações relacionadas são adicionadas à tabela **sys.columns**. Quando é associado a um tipo de dados de usuário, as informações são adicionadas à tabela **sys.types**.

! Este é um recurso que será removido em uma versão futura do SQL Server. É recomendável, portanto, evitar a sua utilização em novos desenvolvimentos.

1.2.9. sp_unbindefault

Essa stored procedure desassocia um valor default de uma coluna ou de um tipo de dados de usuário.

A sintaxe de **sp_unbindefault** é a seguinte:

```
SP_UNBINDEFAULT <tipo_udt>, ['futureonly']
```

Em que:

- **tipo_udt**: É o nome do tipo de dados do usuário;
- **'futureonly'**: Se utilizado, afetará somente as novas utilizações de **<tipo_udt>**. Os objetos que já foram criados utilizando esse tipo de dados não serão afetados.

Quando desassociamos um default de um tipo de dados de usuário, ele é removido de todas as colunas desse tipo de dados com o mesmo default. Se, porém, um default tiver sido associado diretamente a uma coluna, as colunas não serão afetadas.

! Este é um recurso que será removido em uma versão futura do SQL Server. É recomendável, portanto, evitar a sua utilização em novos desenvolvimentos.

1.2.10. 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**.

1.2.11. 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 query window titled "SQLQuery2.sql - ins...DDT (instrutor (53))". The query "SELECT * FROM SYSTYPES" is run, and the results are displayed in a table. The table has columns: name, xtype, status, xusertype, length, xprec, xscale, tdefault, domain, uid, reserved, ci. The data shows various system types like xml, sysname, and several user-defined types starting with TYPE_. The uid column for system types is 4, while for user-defined types it is 1.

	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	5
35	TYPE_NOME_PESSOA	167	1	257	40	0	0	0	0	1	0	5
36	TYPE_NOME_EMPRESA	167	1	258	60	0	0	0	0	1	0	5
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	5
39	TYPE_DATA_MOVTO	61	0	261	8	23	3	0	0	1	0	N

1.2.12. 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	0	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.

1.2.13. Tabela syscomments

A tabela **syscomments** armazena os textos (comando **CREATE**) de todos os objetos contidos no grupo **Programmability** (Programação) do nosso banco de dados, tais como views, stored procedures, functions, regras de validação, defaults, entre outros.

```
SELECT * FROM SYSCOMMENTS
```

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

1.2.14. 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 TESTE_UDDT
```

2. Coloque-o em uso;

```
USE TESTE_UDDT
```

3. Crie os UDDTs;

```
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 UDDTs 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

Na tela anterior, observe as colunas TDEFAULT e DOMAIN.

5. Crie regras de validação;

```
GO
CREATE RULE R_PRECOS AS @PRECO >= 0
GO
CREATE RULE R_SN AS @SN IN ('S','N')
GO
CREATE RULE R_DATA_MOVTO AS @DT <= GETDATE()
GO
-- Exibir as regras que acabaram de ser criadas (SYSOBJECTS)
SELECT * FROM SYSOBJECTS WHERE XTYPE = 'R'
-- OU
SELECT R.NAME, C.TEXT
FROM SYSOBJECTS R JOIN SYSCOMMENTS C ON C.ID = R.ID
WHERE XTYPE = 'R'
```

	NAME	TEXT
1	R_PRECOS	CREATE RULE R_PRECOS AS @PRECO >= 0
2	R_SN	CREATE RULE R_SN AS @SN IN ('S','N')
3	R_DATA_MOVTO	CREATE RULE R_DATA_MOVTO AS @DT <= GETDATE()

6. Associe as regras de validação aos UDDTs;

```
EXEC SP_BINDRULE 'R_PRECOS', 'TYPE_PRECO'
EXEC SP_BINDRULE 'R_SN', 'TYPE_SN'
EXEC SP_BINDRULE 'R_DATA_MOVTO', 'TYPE_DATA_MOVTO'
-- Observe a coluna DOMAIN da SYSTYPES
SELECT * FROM SYSTYPES WHERE UID = 1
```

	name	xtype	status	xusertype	length	xprec	xscale	tdefault	domain	uid
1	TYPE_NOME_PESSOA	167	1	257	40	0	0	0	0	1
2	TYPE_NOME_EMPRESA	167	1	258	60	0	0	0	0	1
3	TYPE_PRECO	108	1	259	9	12	2	0	2105058535	1
4	TYPE_SN	175	0	260	1	0	0	0	2121058592	1
5	TYPE_DATA_MOVTO	61	0	261	8	23	3	0	2137058649	1

Observe a coluna DOMAIN. Ela armazena o ID da regra de validação que está em SYSOBJECTS.

Opções de definição de tabelas

Aulas 8 a 10

7. Crie os valores **DEFAULTS**;

```
GO
CREATE DEFAULT DEF_SN AS 'S'
GO
CREATE DEFAULT DEF_DATA_MOVTO AS GETDATE()
GO
-- Exibir os DEFAULTS que você acabou de criar (SYSOBJECTS)
SELECT * FROM SYSOBJECTS WHERE XTYPE = 'D'
-- OU
SELECT R.NAME, C.TEXT
FROM SYSOBJECTS R JOIN SYSCOMMENTS C ON C.ID = R.ID
WHERE XTYPE = 'D'
```

	NAME	TEXT
1	DEF_SN	CREATE DEFAULT DEF_SN AS 'S'
2	DEF_DATA_MOVTO	CREATE DEFAULT DEF_DATA_MOVTO AS GETDATE()

8. Associe os **DEFAULTS** aos **UDDTs**;

```
EXEC SP_BINDEFAULT 'DEF_SN', 'TYPE_SN'
EXEC SP_BINDEFAULT 'DEF_DATA_MOVTO', 'TYPE_DATA_MOVTO'
-- Observe a coluna TDEFAULT da SYSTYPES
SELECT * FROM SYSTYPES WHERE UID = 1
```

	name	xtype	status	xusertype	length	xprec	xscale	tdefault	domain
1	TYPE_NOME_PESSOA	167	1	257	40	0	0	0	0
2	TYPE_NOME_EMPRESA	167	1	258	60	0	0	0	0
3	TYPE_PRECO	108	1	259	9	12	2	0	2105058535
4	TYPE_SN	175	0	260	1	0	0	5575058	2121058592
5	TYPE_DATA_MOVTO	61	0	261	8	23	3	21575...	2137058649

9. Mostre todas as características dos **UDDTs**;

```
SELECT UT.NAME AS NOME_UDDT,
       T.NAME AS BUILD_IN_BASE,
       UT.LENGTH AS TAMANHO,
       UT.XPREC AS PRECISAO,
       UT.XSCALE AS DECIMAIS,
       R.NAME AS REGRA_VALID,
       D.NAME AS [DEFAULT],
       RT.TEXT AS CREATE_RULE,
       DT.TEXT AS CREATE_DEFAULT
  FROM SYSTYPES UT JOIN SYSTYPES T ON UT.XTYPE = T.XUSERTYPE
  LEFT JOIN SYSOBJECTS R ON UT.DOMAIN = R.ID
  LEFT JOIN SYSOBJECTS D ON UT.TDEFAULT = D.ID
  LEFT JOIN SYSCOMMENTS RT ON R.ID = RT.ID
  LEFT JOIN SYSCOMMENTS DT ON D.ID = DT.ID
 WHERE UT.UID = 1
```

	NOME_UDDT	BUILD_IN_BASE	TAMANHO	PRECISAO	DECIMAIS	REGRA_VALID	DEFAULT	CREATE_RULE	CREATE_DEFAULT
1	TYPE_NOME_PESSOA	varchar	40	0	0	NULL	NULL	NULL	NULL
2	TYPE_NOME_EMPRESA	varchar	60	0	0	NULL	NULL	NULL	NULL
3	TYPE_PRECO	numeric	9	12	2	R_PRECOS	NULL	CREATE RULE R_PRE...	NULL
4	TYPE_SN	char	1	0	0	R_SN	DEF_SN	CREATE RULE R_SN ...	CREATE DEFAULT
5	TYPE_DATA_MOVTO	datetime	8	23	3	R_DATA_MOVTO	DEF_DATA_MOVTO	CREATE RULE R_DAT...	CREATE DEFAULT

10. Efetue o teste necessário.

```

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) )
-- 
-- Vai gerar os valores de SN_ATIVO e DATA_CADASTRO
-- porque estes UDDTs possuem valor default
INSERT INTO PRODUTOS
( DESCRICAO, PRECO_CUSTO, PRECO_VENDA )
VALUES( 'TESTE 1', 1, 2 )
-- 
SELECT * FROM PRODUTOS
-- VAI DAR ERRO -> Preço NEGATIVO
INSERT INTO PRODUTOS
( DESCRICAO, PRECO_CUSTO, PRECO_VENDA )
VALUES( 'TESTE 2', -1, 2 )

```

1.3. 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.

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.

Opções de definição de tabelas

Aulas 8 a 10

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;
```

1.4. 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 ].| database_
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;
```

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.

1.5. 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.

1.5.1. 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**

```
USE PEDIDOS
GO
--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(DESCRICA0, 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
```

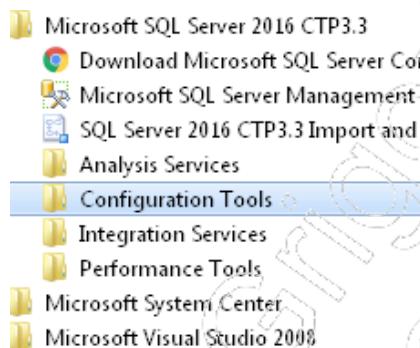
Sempre dê preferência à programação com Stored Procedures ou através do SQL Server Integration Services -SSIS, que permite uma programação mais estruturada.

1.6. FILETABLE

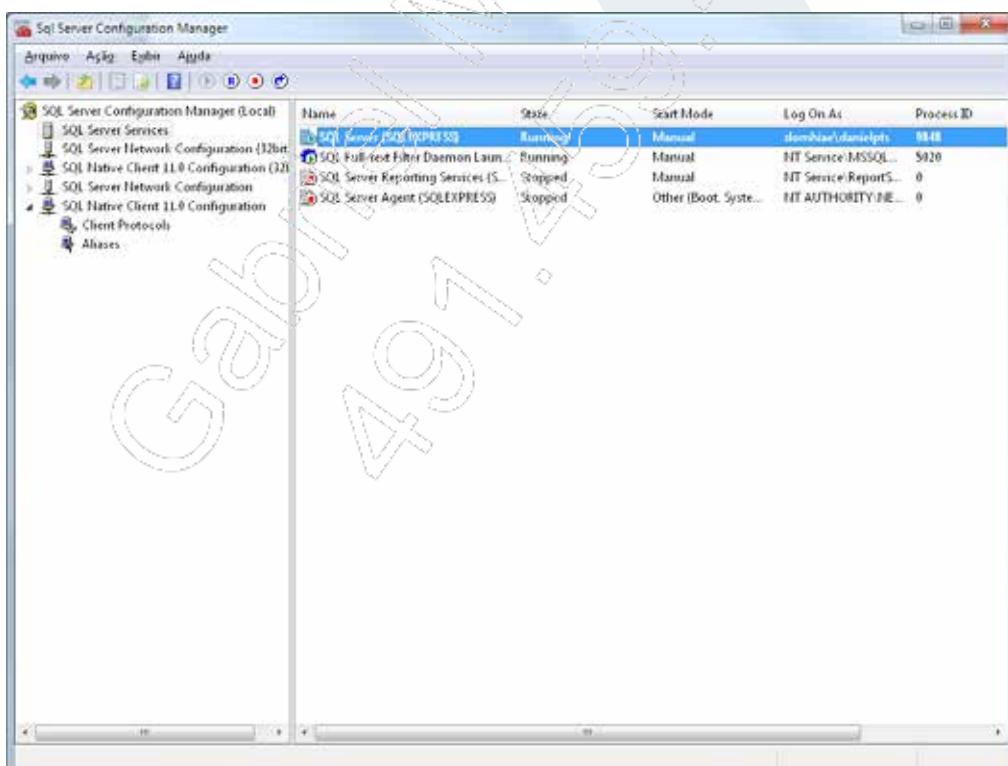
FILETABLE é um recurso que permite carregar objetos binários numa área do sistema do Windows e ser gerenciado pelo SQL Server. A vantagem deste recurso é a separação dos dados estruturados e dos arquivos binários, outra vantagem é a cópia diretamente através do sistema de arquivos.

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

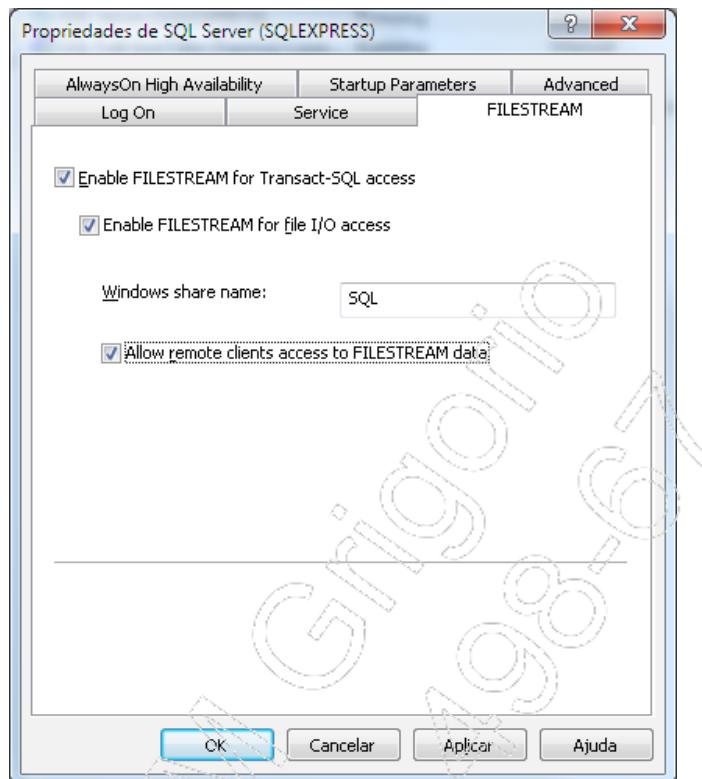
1. Acesse Microsoft SQL Server 2016 CTP3.3 / Configuration Tools;



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

Aulas 8 a 10

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

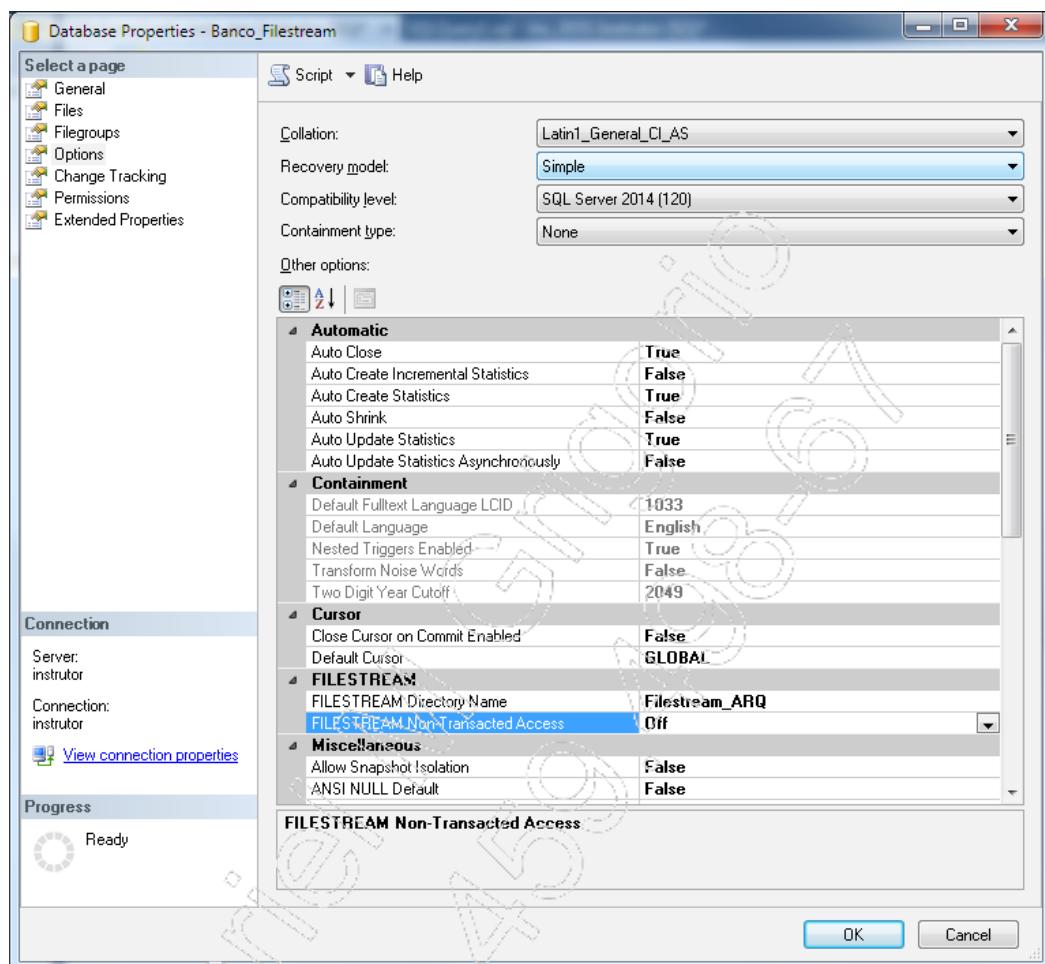
DROP DATABASE BANCO_FILESTREAM
GO

--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:

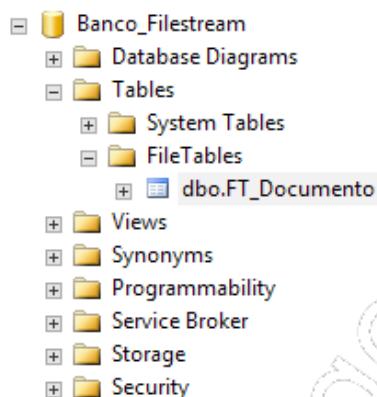
```
USE BANCO_FILESTREAM
GO
CREATE TABLE FT_Documento AS FileTable
--OU
CREATE TABLE FT_Documento AS FileTable
    WITH (
        FileTable_Directory = 'Filestream_ARQ',
        FileTable_Collate_Filename = database_default
    );
GO
```

Opções de definição de tabelas

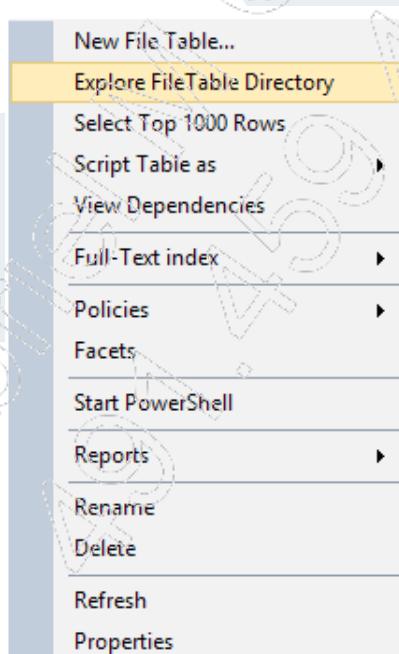
Aulas 8 a 10

- **Visualização e manutenção do modo gráfico**

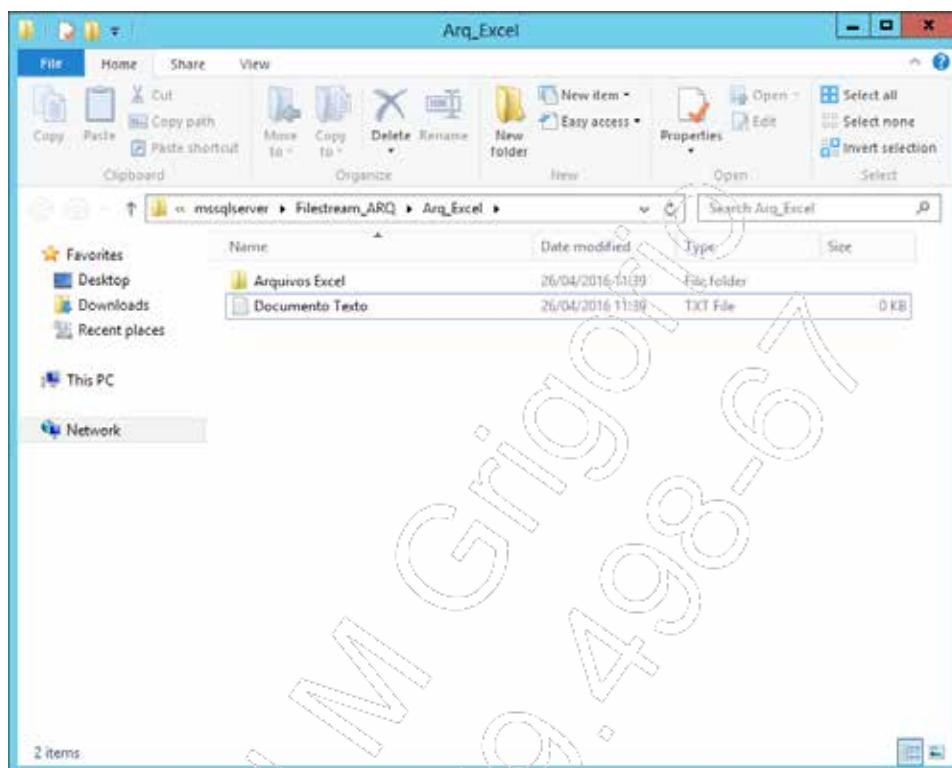
1. Abra o SSMS e, no Object Explorer, expanda o banco de dados, **Tables** e **FileTables**:



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 'Planilha Excel', BulkColumn
from Openrowset (Bulk 'C:\DADOS\PESSOA.XLS', Single_Blob) as
Image
```

- **Inserindo uma subpasta**

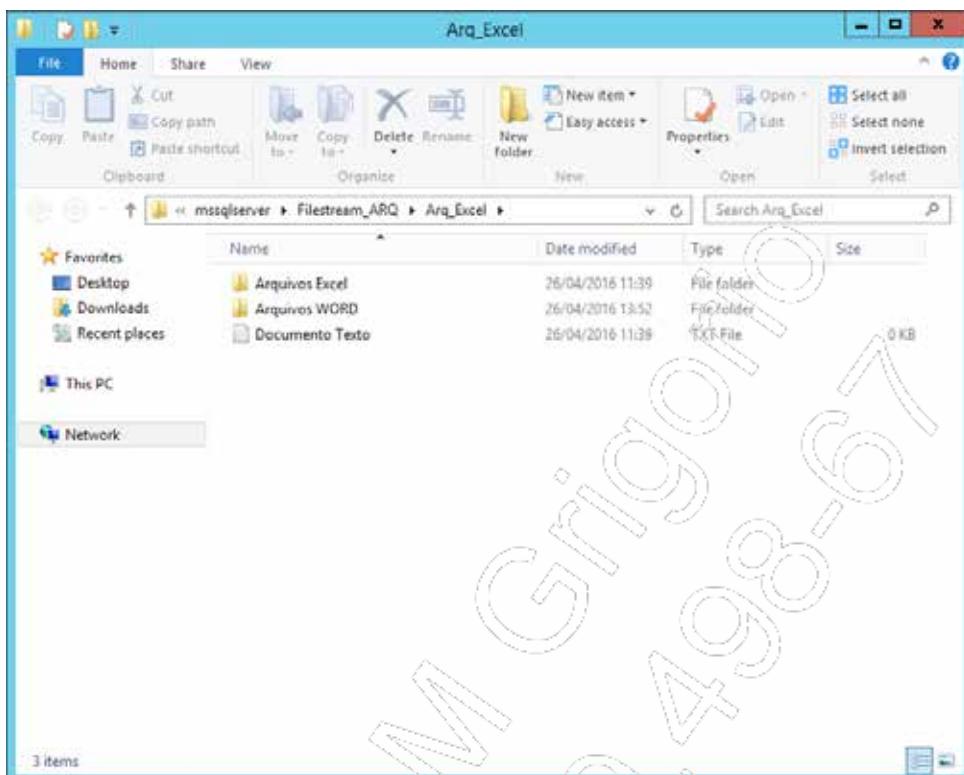
É possível inserir subpastas para melhorar a visualização e organização dos arquivos:

```
INSERT INTO FT_Documento (name,is_directory,is_archive) values
('Arquivos WORD' , 1 ,0)
```

Opções de definição de tabelas

Aulas 8 a 10

Vejamos o resultado após execução do comando:



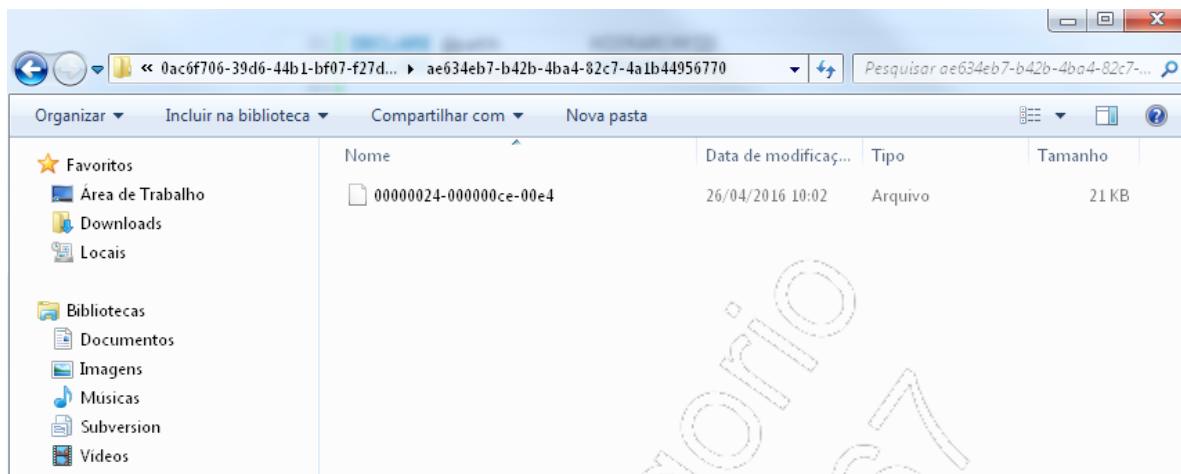
- Consultando uma tabela FILETABLE

```
SELECT * FROM FT_Documento;
```

O resultado apresenta as informações dos arquivos inseridos na tabela:

stream_id	file_stream	name	path_locator	parent_da
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

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'
```

1.7. 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 PEDIDOS  
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)
```

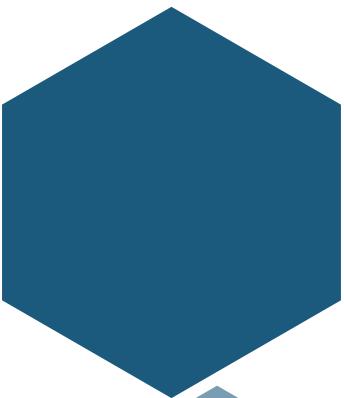
A mesma consulta utilizando a coluna calculada **IDADE**:

```
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 da leitura.

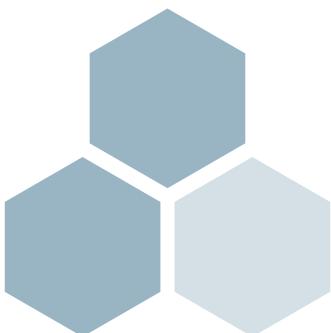
- Os **tipos de dados** determinam os tipos de valores que podem ser inseridos em uma coluna. Os **tipos de dados nativos** são aqueles fornecidos pelo próprio SQL Server. Conhecido como **UDDT**, o **tipo de dados definido pelo usuário** também pode ser considerado um sinônimo de um tipo já disponível;
- A instrução **CREATE RULE** cria um objeto chamado **rule** (ou regra), que é responsável por especificar os valores que podem ser inseridos em uma coluna à qual está associado;
- A instrução **CREATE DEFAULT** cria um objeto **default** (ou padrão). O default atribui automaticamente um valor a uma coluna, caso o registro inserido pelo usuário venha a ocultar esse conteúdo;
- 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.



Opções de definição de tabelas

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 8 a 10.



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. Qual afirmação sobre tipo de dados definido pelo usuário (UDDT) está incorreta?

- a) Para criarmos um tipo de dados, utilizamos o CREATE TYPE.
- b) Não é possível associar regras a uma UDDT.
- c) Associamos regras utilizando a procedure SP_BINDRULE.
- d) Para criar um valor padrão, utilizamos o comando CREATE DEFAULT.
- e) A associação do valor DEFAULT com o tipo é realizada através da procedure SP_BINDEFAULT.

4. Com relação a SEQUENCE, qual afirmação está errada?

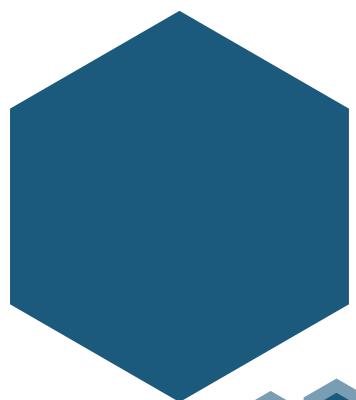
- a) É um objeto que retorna um valor sequencial.
- b) Este objeto não está vinculado a apenas uma tabela.
- c) A sequência pode apresentar “furos”.
- d) Pode ser utilizado em campos que são chaves primárias.
- e) Este recurso não é funcional, pois a tabela já possui um IDENTITY.

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.

SQL 2016 - Programação em T-SQL (online)

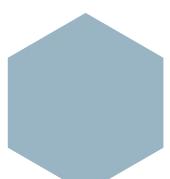
Gabriel M Grigorio
497.459.498-67



Opções de definição de tabelas

Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 8 a 10.



Laboratório 1

A – Criando e associando UDDTs, regras de validação e objetos DEFAULTs

1. Crie os seguintes UDDTs:

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

2. Exiba os UDDTs recém-criados;

3. Crie as seguintes regras de validação:

R_SEXO	Aceita somente 'F' e 'M'
R_ALIQUOTA	Números não negativos
R_PRAZO	Números no intervalo de 1 até 60

4. Exiba as regras de validação recém-criadas;

5. Associe as regras aos UDDTs;

R_SEXO	ao UDDT TIPO_SEXO
R_ALIQUOTA	ao UDDT TIPO_ALIQUOTA
R_PRAZO	ao UDDT TIPO_PRAZO

6. Crie os seguintes objetos DEFAULT:

D_SEXO	"M"
D_ALIQUOTA	0 (ZERO)
D_PRAZO	1

7. Exiba os DEFAULTs recém-criados;

8. Associe os DEFAULTs aos UDDTs;

D_SEXO	a TIPO_SEXO
D_ALIQUOTA	a TIPO_ALIQUOTA
D_PRAZO	a TIPO_PRAZO

9. Crie a tabela PESSOAS, seguindo o modelo adiante:

COD_PESSOA	TIPO_CODIGO	autonumeração chave primária
NOME	VARCHAR(30)	
ENDERECO	TIPO_ENDERECO	
SEXO	TIPO_SEXO	

Opções de definição de tabelas

Aulas 8 a 10

10. 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	

11. Insira cinco registros na tabela **PESSOAS**;

12. Exiba os registros da tabela **PESSOAS**;

13. Crie a tabela **FUNCIONARIO**, seguindo o modelo adiante:

FUNCIONARIO		
COD_FUNC	TIPO_CODIGO	chave primária
NOME	VARCHAR(30)	
ENDERECO	VARCHAR(80)	
SEXO	TIPO_SEXO	

14. Crie um sinônimo de nome **tb_Funcionario** para a tabela **FUNCIONARIO**;

15. Crie uma SEQUENCE de nome **SQ_FUNCIONARIO**, que inicie em 100 com incremento 2;

16. Insira um registro na tabela **FUNCIONARIO** utilizando a SEQUENCE **SQ_FUNCIONARIO** e o sinônimo **tb_Funcionario**.

Laboratório 2

A – Trabalhando com objetos binários

1. Crie a tabela **TB_DOCUMENTO** com as seguintes características:

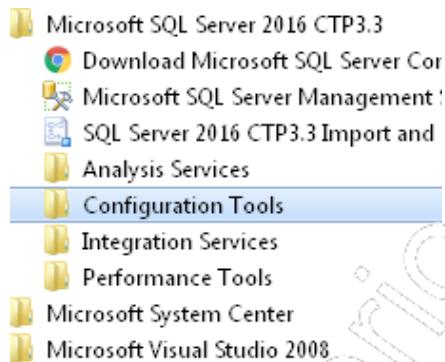
- **ID_DOCUMENTO**, inteiro, autonumerável e chave primária;
- Descrição do documento – Texto livre com até 100 caracteres;
- Data do Cadastro – Deve possuir valor padrão (Data e Hora do servidor);
- Documento – Campo binário.

2. Insira dois documentos na tabela **TB_DOCUMENTO**;

3. Consulte a tabela **TB_DOCUMENTO**.

B – Habilitando FILETABLE

1. Acesse o SQL Configuration Tools;



2. Escolha o SQL Server Configuration Manager;

3. Selecione o serviço do SQL Server e 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. No SQL Server Management Studio, execute o seguinte comando:

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

6. Para criar um banco com FILESTREAM, execute o comando adiante:

```
CREATE DATABASE Banco_LAB3  
ON PRIMARY  
(Name = FG_Filestream_PRIMARY,  
FILENAME = 'C:\DADOS\LAB_Filestream_DATA3.mdf'),  
FILEGROUP FG_Filestream_FS CONTAINS FILESTREAM  
(NAME = Filestream_ARQ,  
FILENAME='C:\DADOS\LAB_Filestream_ARQ3')  
LOG ON  
(Name = Filestream_log,  
FILENAME = 'C:\DADOS\LAB_Filestream_log3.ldf')  
WITH FILESTREAM (NON_TRANSACTED_ACCESS = FULL,  
DIRECTORY_NAME = N'Filestream_ARQ3');  
GO
```

C- Inserindo e visualizando arquivos

1. Coloque o banco **BANCO_LAB3** em uso;
2. Crie uma tabela FILETABLE de nome **FT_Documento**;
3. Insira dois documentos nessa tabela;
4. Visualize os documentos de forma gráfica;
5. Visualize os documentos com comando TSQL.

Laboratório 3

A – Trabalhando com colunas computadas

1. Coloque o banco **PEDIDOS** em uso;
2. Crie a tabela **TB_FUNC_IDADE** com os seguintes campos:

```
Id_funcionario      inteiro, auto numerável e chave primária  
Nome do funcionário alfanumérico com 50 caracteres  
Data de Nascimento Campo data  
Idade              Campo calculado
```

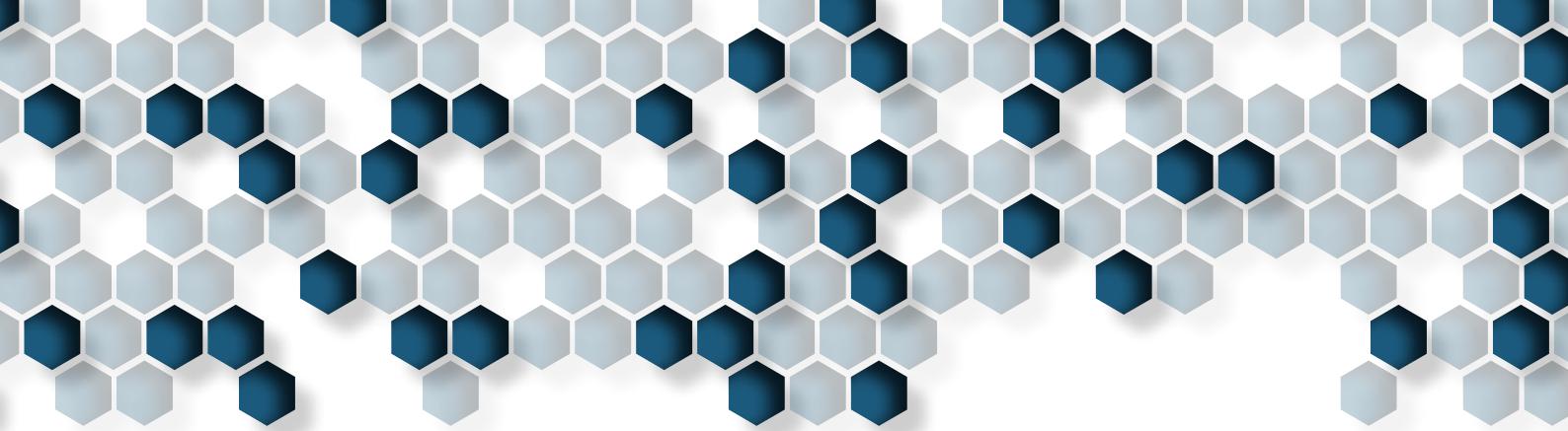
3. Insira os dados da tabela de empregados para a tabela **TB_FUNC_IDADE**;
4. Consulte as informações e verifique o campo calculado;
5. Adicione o campo **VLR_ITEM** na tabela **TB_ITENSPEDIDO**, com o cálculo adiante:

```
PR_UNITARIO * QUANTIDADE * (1 - DESCONTO /100)
```

6. Faça uma consulta na tabela e verifique o resultado.

SQL 2016 - Programação em T-SQL (online)

Gabriel M Grigorio
497.459.498-67



Customizando consultas

- 
- ◆ Plano de execução;
 - ◆ Dicas para construir consultas;
 - ◆ Índices;
 - ◆ Otimizando consultas.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 11 e 12.



1.1. Introdução

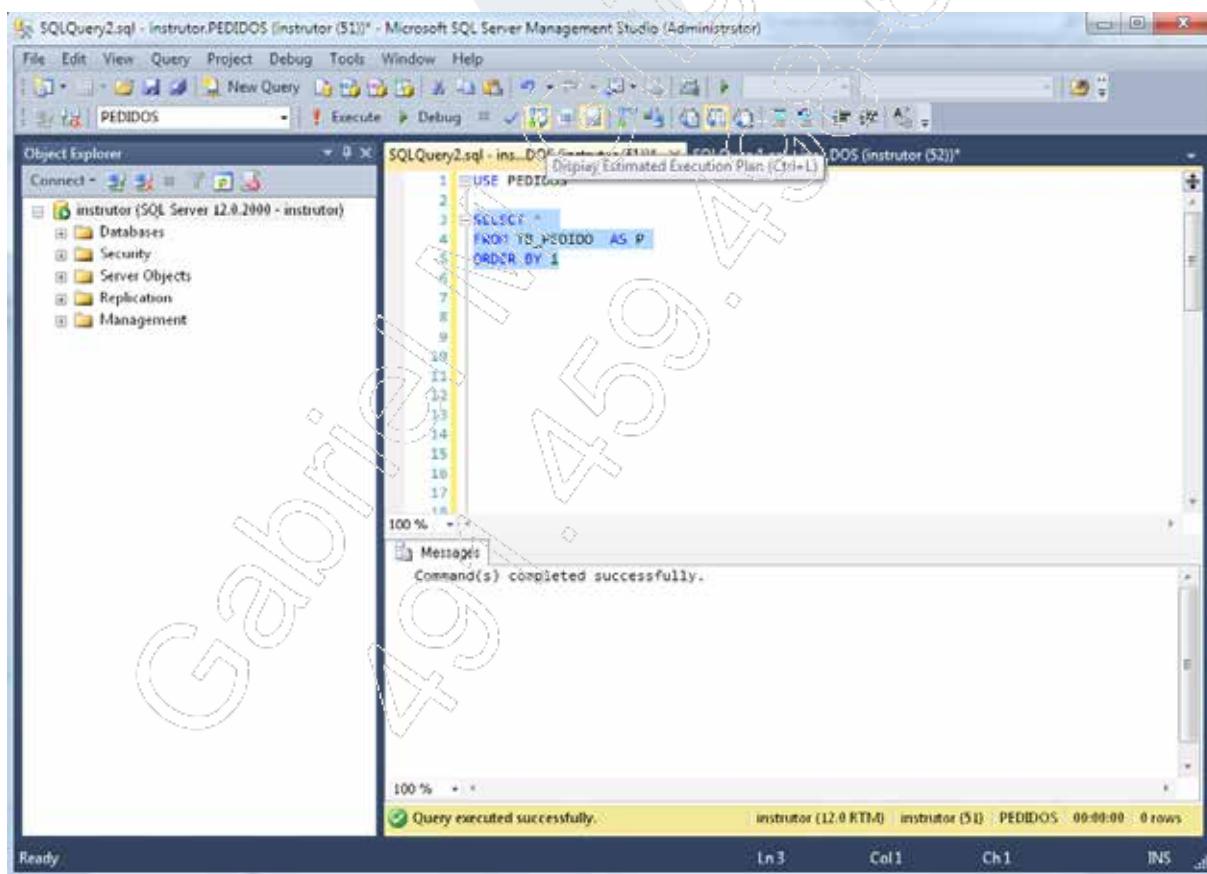
Quando executamos comandos no SQL Server, muitas vezes não nos preocupamos com o impacto que será gerado no servidor. É importante que nossas consultas garantam sempre o menor impacto no ambiente.

A customização do código deve se iniciar na fase de desenvolvimento do projeto, assim evitamos problemas no ambiente produtivo.

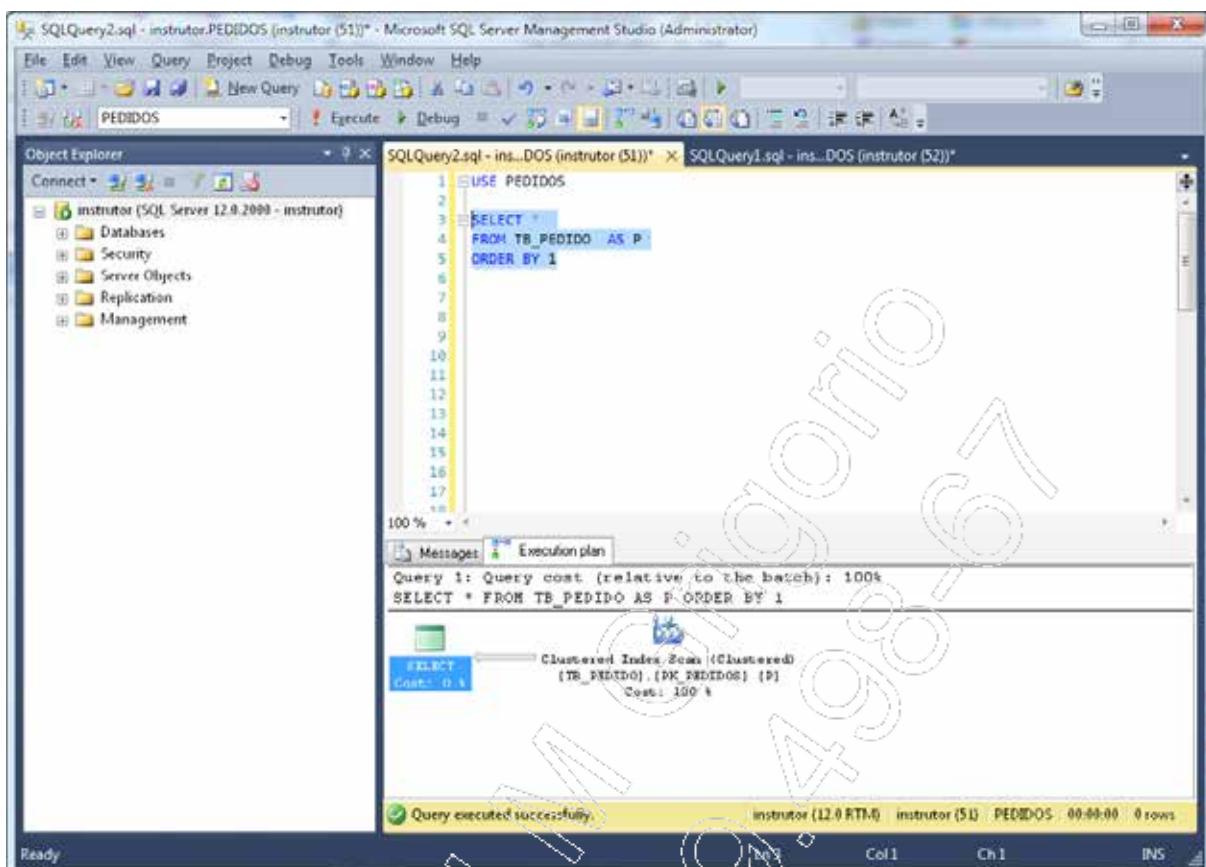
1.2. Plano de execução

É importante o entendimento de como o SQL otimiza uma consulta.

Para verificar o plano de execução estimado de uma consulta, utilize CTRL + L ou o botão **Display Estimated Execution Plan**:



Ao executar a consulta, o SQL mostrará como ela será executada e não os valores resultantes dela:



The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, a database named 'PEDIDOS' is selected. In the center pane, a query window displays the following T-SQL code:

```
1 USE PEDIDOS
2
3 SELECT *
4   FROM TB_PEDIDO AS P
5   ORDER BY 1
```

Below the code, the 'Messages' tab shows the execution results:

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM TB_PEDIDO AS P ORDER BY 1
```

An execution plan diagram is visible, showing a 'Clustered Index Scan (Clustered)' operation on the 'TB_PEDIDO' table, with a cost of 100. The status bar at the bottom indicates 'Query executed successfully.'

1.3. Dicas para construir consultas

A seguir, veremos algumas dicas que ajudarão na construção e otimização das consultas:

- **Na fase de desenvolvimento**
 - **Normalização:** A normalização é uma técnica para deixar o banco rápido e inteligente. Utilize as principais normas formais, para garantir que as tabelas foram desenvolvidas da maneira correta;
 - **Constraints:** A utilização de constraints é fundamental para a integridade das informações do banco, e são obrigatórias. A chave primária (**PRIMARY KEY**) da tabela identifica, de forma única e exclusiva, cada uma das linhas. Dessa maneira, todos os campos podem ser trocados (exceto a chave). Outra vantagem na criação da **PRIMARY KEY** é que o SQL automaticamente criará um índice clusterizado, ou seja, ordenará a tabela por este campo. Já a constraint **FOREIGN KEY** relacionará, de forma física, as tabelas, garantindo integridade relacional dos dados;

- **Poucas colunas:** Dependendo do assunto de negócio, a quantidade de informação pode gerar muitos campos. Como boa prática, recomenda-se a criação de tabelas segmentadas por assunto, por exemplo, para o assunto **Cadastro de funcionários**, crie tabelas separadas para: **Dados de cadastro, Endereços, Contatos e Telefones e Documentos**;
 - **Evite campos binários:** A utilização de campos binários simplifica o gerenciamento de arquivos, porém traz um grande inconveniente que é o crescimento exponencial do banco de dados. Esse crescimento ocorre por arquivos binários não serem estruturados;
 - **Utilize FILETABLE:** Para um gerenciamento melhor de arquivos binários, substitua campos binários por tabelas do tipo FILETABLE;
 - **Evite tabelas temporárias e cursores:** São ferramentas úteis, porém consomem muito recurso da máquina.
- **Nas consultas**
 - **Evite asterisco (*):** Ao efetuar um **SELECT ***, o SQL retornará todos os campos da(s) tabela(s). Como na grande maioria das vezes os campos utilizados são poucos, este tipo de recurso consumirá mais recursos do que informar os campos explicitamente;
 - **Filtre as consultas:** Utilize a cláusula **WHERE** para filtrar o resultado da consulta, evitando acesso a toda a tabela;
 - **Operando LIKE:** Evite este operando, pois realiza uma varredura no campo;
 - **Cuidado ao utilizar hints (dicas):** Hints são recursos úteis, mas podem prejudicar a performance não só da consulta como também do servidor;
 - **Índices:** Sempre verifique quais consultas precisam de índices. Uma dica é criar índices para consultas que são frequentemente utilizadas;
 - **Evite consultas AD HOC:** Consultas pontuais são mais lentas, pois o SQL verificará o código (PARSE) e analisará qual o melhor plano de execução. Como a consulta é pontual, o SQL não passará essas informações para o otimizador. Crie procedures, views e funções que possam ser analisadas e corrigidas diretamente no banco de dados.

1.4. Índices

Os índices tornam mais rápidos os procedimentos de ordenação e de busca na tabela. Duas estruturas diferentes de índices podem ser utilizadas:

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

A partir da versão 2012, são permitidos até 999 índices não clusterizados por tabela.

ÍNDICES NÃO “CLUSTERIZADOS”

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

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

1.4.1. Criando índices

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

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

Em que:

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

Exemplos:

- Verificando a tabela **TB_EMPREGADO** do banco **PEDIDOS**:

```
USE PEDIDOS  
EXEC SP_HELPINDEX TB_EMPREGADO
```

No resultado, somente é apresentado o índice clusterizado da chave primária:

index_name	index_description	index_keys
PK_Empregados	clustered, unique, primary key located on PRIMARY	CODFUN

- Criando um índice para o campo **NOME**:

```
CREATE INDEX IX_TB_EMPREGADO_NOME ON TB_EMPREGADO (NOME)
```

- Criando um índice para o campo **DATA_NASCIMENTO**:

```
CREATE INDEX IX_TB_EMPREGADO_DATA_NASCIMENTO ON TB_EMPREGADO (DATA_NASCIMENTO)
```

1.4.2. Índices compostos

Índices compostos são aqueles que possuem mais de um campo na definição do índice. Exemplo:

- Na tabela de fornecedores existe apenas o índice clusterizado da tabela **TB_FORNECEDOR**:

```
EXEC SP_HELPINDEX TB_FORNECEDOR
```

- Criação de um índice composto pelas colunas **ESTADO** e **CIDADE**:

```
CREATE INDEX IX_TB_FORNECEDOR_ESTADO_CIDADE ON TB_FORNECEDOR  
(ESTADO, CIDADE)
```

1.4.3. INCLUDE

INCLUDE são colunas incluídas no índice. Assim, dependendo da consulta, o SQL acessará somente o índice e não precisará acessar a tabela. Suas vantagens são a possibilidade de inclusão de campos não suportados como índices e o fato de não serem computados como números de colunas e tamanho dos índices.

Vejamos um exemplo de uma consulta que é realizada diversas vezes com os campos de código do cliente, nome e Estado:

```
SELECT CODCLI, NOME, ESTADO FROM TB_CLIENTE
```

Verifique o plano de execução da consulta:

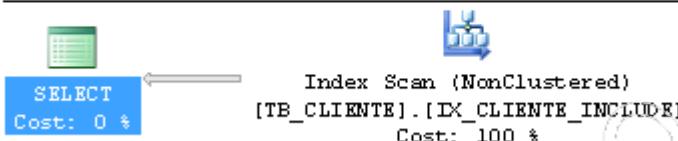


Mesmo a tabela possuindo um índice para o campo **NOME**, é necessário mais um campo que é o **ESTADO**. Assim, o SQL utiliza o próprio índice clusterizado.

Ao criarmos um índice com a coluna **ESTADO**, o SQL muda o comportamento, pois não é necessário acessar a tabela. O índice atende todas as necessidades da consulta.

```
CREATE INDEX IX_CLIENTE_INCLUDE ON TB_CLIENTE (NOME) INCLUDE  
(ESTADO)
```

Query 1: Query cost (relative to the batch): 100%
SELECT CODCLI, NOME, ESTADO FROM TB_CLIENTE



1.4.4. Excluindo índices

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

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

Em que:

- **<nome_tabela_ou_view>**: É o nome da tabela ou view com a qual o índice a ser excluído está associado;
- **<nome_indice>**: É o nome do índice a ser excluído.

Vejamos um exemplo em que o índice **IX_TB_CLIENTE_INCLUDE** é excluído da tabela **TB_CLIENTE**:

```
DROP INDEX TB_CLIENTE.IX_CLIENTE_INCLUDE
```

1.5. Otimizando consultas

Todas as instruções realizadas pelo SQL são processadas pelo Otimizador do SQL Server. Esse componente é responsável por determinar qual é o melhor caminho de acesso do dado nas tabelas. Assim, ao executar uma consulta, o SQL realizará as seguintes tarefas:

- Verificar se o código do comando está correto (PARSE);
- Verificar a sequência de leitura das tabelas;
- Utilizar os índices para melhoria de acesso ao dado;
- Utilizar os recursos de processamento.

1.5.1. Hints

É possível customizar (forçar) o SQL a sobrepor o Otimizador do SQL. Desta maneira, podemos definir como o SQL realizará a consulta.

Hints são úteis, mas podem causar efeitos indesejados como baixa performance ou comprometimento do processamento do servidor. Sempre dê preferência ao Otimizador do SQL e, em poucos casos, passe estas dicas.

Vejamos, a seguir, uma lista com os principais hints utilizados:

- **INDEX**

Ao utilizar este hint, você pode definir qual INDEX será utilizado. Observe, no exemplo adiante, que a consulta apresenta os clientes do estado de SP:

```
SELECT * FROM TB_CLIENTE WHERE ESTADO='SP'
```

Ao verificar o plano de execução, o SQL utiliza o índice da chave primária e não o índice por estado.

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM TB_CLIENTE WHERE ESTADO='SP'
-----
```

The execution plan diagram shows a query for selecting all columns from the TB_CLIENTE table where the state (ESTADO) is 'SP'. The plan consists of two main components: a 'Clustered Index Scan (Clustered)' operation on the primary key index [TB_CLIENTE].[PK_CLIENTES], which has a cost of 100%, and a 'SELECT' operation with a cost of 0%. An arrow points from the 'Clustered Index Scan' to the 'SELECT' operation, indicating that the scan feeds the select statement.

Isso pode ocorrer por várias razões: tamanho da tabela, quantidade de campos de retorno, estatísticas de acesso etc.

Para verificar se existe INDEX na tabela, utilize a procedure **SP_HELPINDEX**:

```
EXEC SP_HELPINDEX TB_CLIENTE
```

No resultado, existe um INDEX que atende à cláusula **WHERE**:

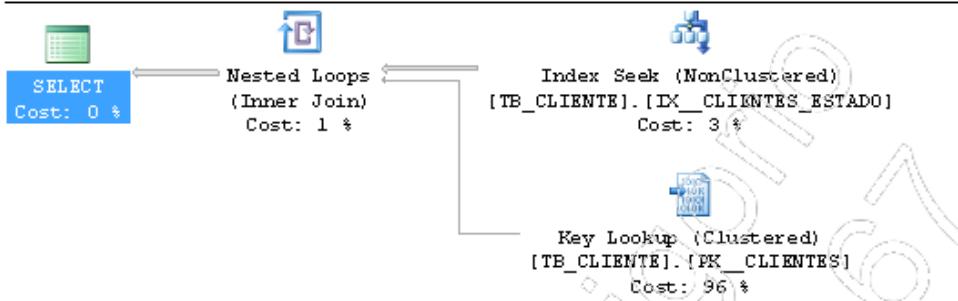
index_name	index_description	index_keys
IX_CLIENTES_CEP	nonclustered located on INDICES	CEP
IX_CLIENTES_CGC	nonclustered located on INDICES	CNPJ
IX_CLIENTES_ESTADO	nonclustered located on INDICES	ESTADO
IX_CLIENTES_FANTASIA	nonclustered located on INDICES	FANTASIA
IX_CLIENTES_NOME	nonclustered located on INDICES	NOME
PK_CLIENTES	clustered, unique, primary key located on PRIMARY	CODCLI

Utilizando o hint **INDEX**, a consulta ficará conforme o código adiante:

```
SELECT * FROM TB_CLIENTE WITH (INDEX = IX_CLIENTES_ESTADO) WHERE  
ESTADO='SP'
```

Vejamos o plano de execução:

Query 1: Query cost (relative to the batch): 100%
SELECT * FROM TB_CLIENTE WITH (INDEX = IX_CLIENTES_ESTADO) WHERE ESTADO='SP'



- **ROWLOCK**

Obriga a utilização de um bloqueio de linha, que é mantido pelo SQL Server até o final do comando. Porém, o bloqueio será mantido até o final da transação caso especifiquemos **HOLDLOCK**. Este tipo de bloqueio é o padrão utilizado pelo SQL Server.

! Bloqueio é o mecanismo do SQL para evitar que um recurso seja acessado por mais de uma transação.

```
UPDATE TB_CLIENTE WITH (ROWLOCK) SET ESTADO = ESTADO
```

- **PAGLOCK**

Esta opção obriga a utilização do bloqueio compartilhado no nível de página, sendo que tal bloqueio compartilhado é mantido até o final do comando. Assim como ocorre com **ROWLOCK**, o bloqueio será mantido até o final da transação, caso especifiquemos **HOLDLOCK**.

```
UPDATE TB_CLIENTE WITH (PAGLOCK) SET ESTADO = ESTADO
```

- **TABLOCK**

Por meio desta opção, o SQL Server é obrigado a utilizar o bloqueio compartilhado na tabela. Assim, os usuários poderão ler a tabela, mas não alterá-la.

```
UPDATE TB_CLIENTE WITH (TABLOCK) SET ESTADO = ESTADO
```

- **UPDLOCK**

Esta opção não obriga a utilização do bloqueio compartilhado, mas sim do bloqueio de atualização no nível de página durante a leitura dos dados da tabela. Quando utilizamos um bloqueio de atualização, outras transações concorrentes têm acesso de leitura dos dados, mas estão impedidas de fazer alterações neles.

```
UPDATE TB_CLIENTE WITH (UPDLOCK) SET ESTADO = ESTADO
```

- **TABLOCKX**

Por meio desta opção, o SQL Server é obrigado a utilizar o bloqueio exclusivo na tabela, sendo que tal bloqueio exclusivo é mantido até o final do comando. O bloqueio exclusivo impede a leitura e a alteração da tabela por parte de outros.

```
UPDATE TB_CLIENTE WITH (TABLOCKX) SET ESTADO = ESTADO
```

- **NOLOCK**

Esta opção não respeita a utilização de bloqueios. Quando aplicada, **NOLOCK** permite:

- A existência, na leitura, de transações não confirmadas ou um conjunto de páginas desfeitas (**ROLLBACK**);
- Leituras sujas (**Dirty-Reads**).

```
SELECT * FROM TB_CLIENTE WITH (NOLOCK)
```

! É um hint que aumenta consideravelmente a performance da consulta e eliminação de bloqueios compartilhados, porém pode gerar informações inconsistentes. Utilize este hint quando não existir a obrigatoriedade da precisão da informação.

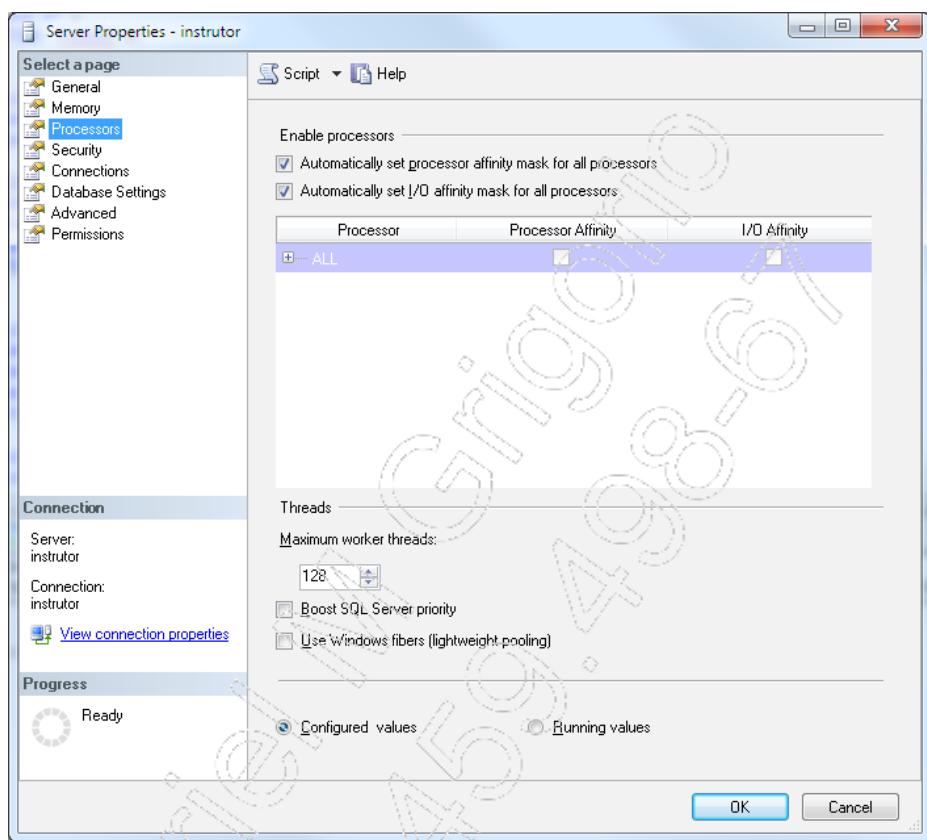
- **READ PAST**

Quando a opção **READ PAST** é utilizada, uma tabela com dados bloqueados pode ser lida, sendo que teremos a exibição apenas dos dados que não estão bloqueados.

```
SELECT * FROM TB_CLIENTE WITH (READPAST)
```

- MAXDOP

MAXDOP tem a finalidade de definir a quantidade de processadores utilizados na consulta. Ao aumentar a utilização de processadores, a consulta ganhará performance, porém pode gerar um impacto negativo com relação ao servidor que aguardará o encerramento da transação. A configuração do paralelismo é realizada nas propriedades do servidor, na aba **Processors**:



Vejamos uma consulta sem hint que utiliza o paralelismo padrão do servidor:

```
SELECT *
FROM TB_ITENSPEDIDO AS P
WHERE NUM_PEDIDO IN (SELECT NUM_PEDIDO FROM TB_PEDIDO WHERE
YEAR(DATA_EMISSAO) =2014)
ORDER BY 1,2
```

Vejamos uma consulta com hint que define que será utilizado somente um processador para a consulta:

```
SELECT *
FROM TB_ITENSPEDIDO AS P
WHERE NUM_PEDIDO IN (SELECT NUM_PEDIDO FROM TB_PEDIDO WHERE
YEAR(DATA_EMISSAO) =2014)
ORDER BY 1,2
OPTION (MAXDOP 1)
```

1.5.2. Customizando bloqueios na seção

Em uma seção, utilizamos opções de nível de isolamento específicas para fazer a configuração do bloqueio. Para configurar um nível de isolamento a ser aplicado na seção, utilizamos **SET TRANSACTION ISOLATION LEVEL**.

A configuração do nível de isolamento possibilita determinar qual será o comportamento padrão dos bloqueios dos comandos existentes em uma seção. Por meio das especificações de bloqueio, podemos sobrepor um bloqueio de seção de um determinado comando.

A sintaxe de **SET TRANSACTION ISOLATION LEVEL** é a seguinte:

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED  
| REPEATABLE READ | SERIALIZABLE | SNAPSHOT}
```

Podemos especificar um **isolamento de transação** de um comando. Nesse caso, utilizamos o comando **DBCC USEROPTIONS**, como mostrado a seguir:

```
SET TRANSACTION ISOLATION LEVEL nivel_isolamento  
DBCC USEROPTIONS
```

A seguir, veremos os níveis de isolamento utilizados junto de **SET TRANSACTION ISOLATION LEVEL**:

- **READ COMMITTED**

Este argumento, que é o padrão, faz com que o SQL Server utilize bloqueios compartilhados em operações de leitura. **READ COMMITTED** não aceita leituras sujas.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

A opção **READ COMMITTED** tem o comportamento controlado por **READ_COMMITTED_SNAPSHOT**. Esta é uma opção de banco de dados que pode ser ajustada como **ON** ou **OFF**, como podemos ver a seguir:

```
SET READ_COMMITTED_SNAPSHOT ON ou OFF
```

- **READ COMMITTED** com **READ_COMMITTED_SNAPSHOT OFF**

Faz com que o SQL Server utilize bloqueios compartilhados enquanto faz a leitura. Com este nível, não é possível fazer leituras sujas.

- **READ COMMITTED** com **READ_COMMITTED_SNAPSHOT ON**

Faz com que o SQL Server use um versionamento de linhas, e não bloqueio. Com este nível, outras transações podem fazer atualizações nos dados, que não ficam protegidos.

- **READ UNCOMMITTED**

Exerce a mesma função de **NOLOCK**, ou seja, o SQL Server é obrigado a não gerar bloqueios compartilhados. Diferente de **READ COMMITTED**, esta opção aceita leituras sujas.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

- **SERIALIZABLE**

Tido como o mais restritivo dos bloqueios, o **SERIALIZABLE** proíbe a alteração ou inserção de novas linhas que apresentam o mesmo critério da cláusula **WHERE** da transação. Desta forma, impede a ocorrência de leituras fantasma.

É aconselhável utilizar o nível de isolamento **SERIALIZABLE** apenas quando necessário.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

- **REPEATABLE READ**

Esta opção evita que os dados acessados em uma query sejam alterados por outros usuários, por meio da utilização de bloqueios nos dados (**SERIALIZABLE**). Ela também impede que a transação atual acesse dados alterados por outras transações que ainda não foram finalizadas (**READ COMMITTED**). Quando utilizamos **REPEATABLE READ**, as ocorrências de leituras não repetidas e leituras sujas não são permitidas.

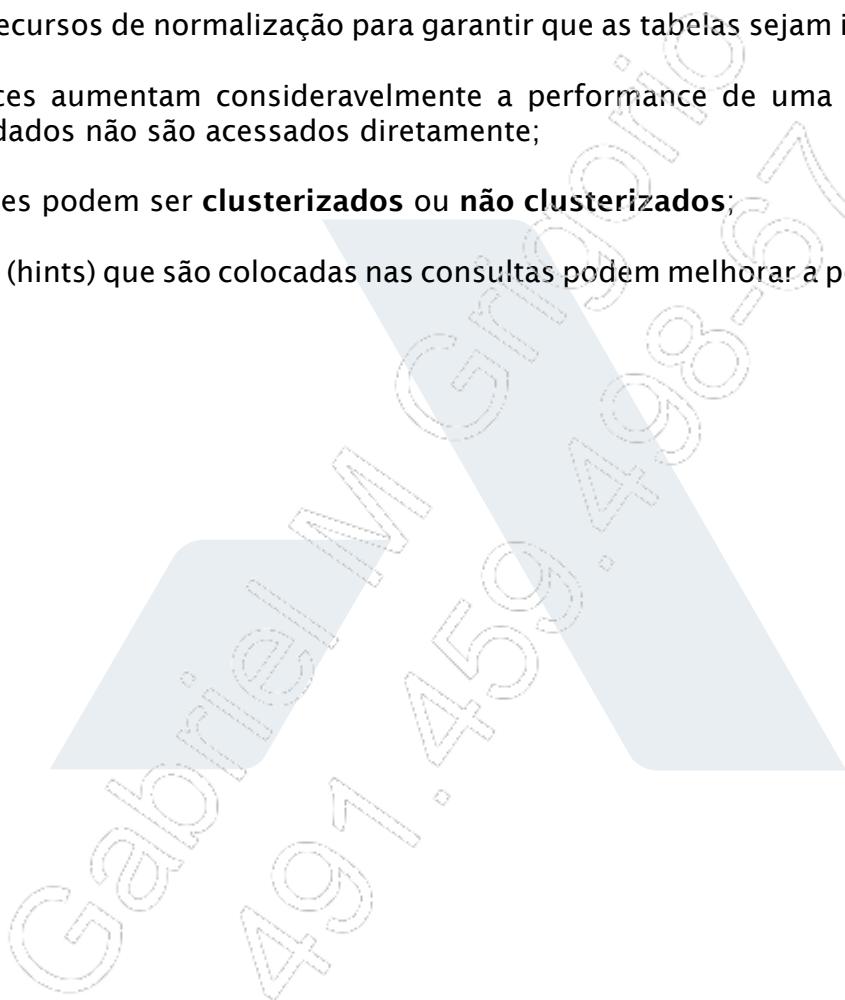
Apesar de tal condição, outro usuário tem a possibilidade de acrescentar novas linhas no conjunto. Assim, na próxima leitura da transação atual, as novas linhas serão consideradas. Lembramos que **REPEATABLE READ** só deve ser utilizada quando realmente necessária, já que apresenta um nível de concorrência menor do que o nível de isolamento padrão **READ COMMITTED**. Isso se deve ao fato de os bloqueio compartilhados serem mantidos até o final da transação, em vez de serem lançados ao final de cada comando.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

Pontos principais

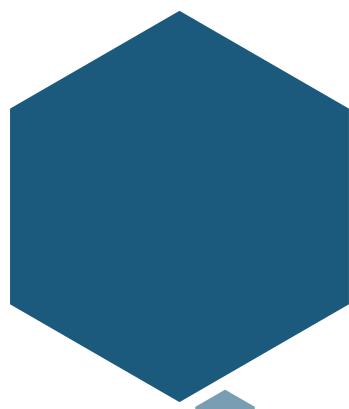
Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes da leitura.

- Ao iniciar um projeto que consome recursos de banco de dados, é importante desenvolver o código TSQL para que seja eficiente no ambiente produtivo;
- Utilize recursos de normalização para garantir que as tabelas sejam inteligentes;
- Os índices aumentam consideravelmente a performance de uma consulta, já que os dados não são acessados diretamente;
- Os índices podem ser **clusterizados** ou **não clusterizados**;
- As dicas (hints) que são colocadas nas consultas podem melhorar a performance.



SQL 2016 - Programação em T-SQL (online)

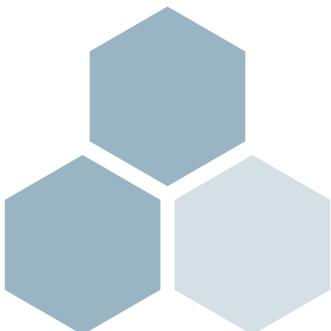
Gabriel M Grigorio
497.459.498-67



Customizando consultas

• Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 11 e 12.



1. Porque é importante utilizar a normalização?

- a) A normalização é importante, pois cria várias tabelas.
- b) O recurso de normalização permite a utilização de FOREIGN KEY.
- c) Sempre que possível, desnormalize as tabelas para evitar tabelas grandes.
- d) Permite a criação de um modelo mais inteligente.
- e) Para criar tabelas menores.

2. Quantos índices são possíveis criar no SQL?

- a) 1 Clusterizado e 999 não clusterizados.
- b) 999 clusterizados e 1 não clusterizado.
- c) 999 não clusterizados.
- d) 254 não clusterizados.
- e) Até 999 clusterizados e não clusterizados.

3. Qual a função do INCLUDE?

- a) Inclui colunas na tabela.
- b) Inclui colunas no índice clusterizado.
- c) Adiciona colunas no índice.
- d) Inclui colunas no índice não clusterizado.
- e) Inclui uma coluna calculada.

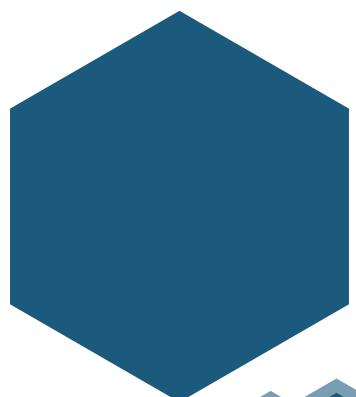
4. Qual das opções a seguir não é um hint?

- a) MAXDOP
- b) NOLOCK
- c) INDEX
- d) UPDLOCK
- e) LINELOCK

5. Qual é a função do comando adiante?

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

- a) Customiza a seção para nível de isolamento READ COMMITTED.
- b) Altera o banco de dados para READ COMMITTED.
- c) Efetua o mesmo que READPAST.
- d) Altera a seção para leitura de dados sujos.
- e) Evita leitura de dados não confirmado no banco de dados.

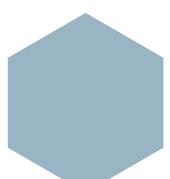


Customizando consultas



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 11 e 12.



Laboratório 1

A – Índices

1. Coloque o banco **PEDIDOS** em uso;
2. Verifique se a tabela **TB_CLIENTE** possui índices;
3. Crie os índices para a tabela **TB_CLIENTE**. Os campos são:
 - Nome;
 - Fantasia;
 - Estado e Cidade;
 - Nome, e inclua os campos Estado e Cidade.
4. Crie os índices para a tabela **TB_PEDIDO**. Os campos são:
 - **Data_emissao**;
 - **CODCLI**;
 - **CODVEN**.

B – Customizando consultas

1. Execute uma consulta que apresente os clientes. Utilize um hint que force a utilização de um índice criado no laboratório anterior;

2. Execute o comando:

```
BEGIN TRAN  
  
UPDATE TB_PRODUTO SET PRECO_VENDA *=PRECO_VENDA *1.2  
WHERE COD_TIPO=3;
```

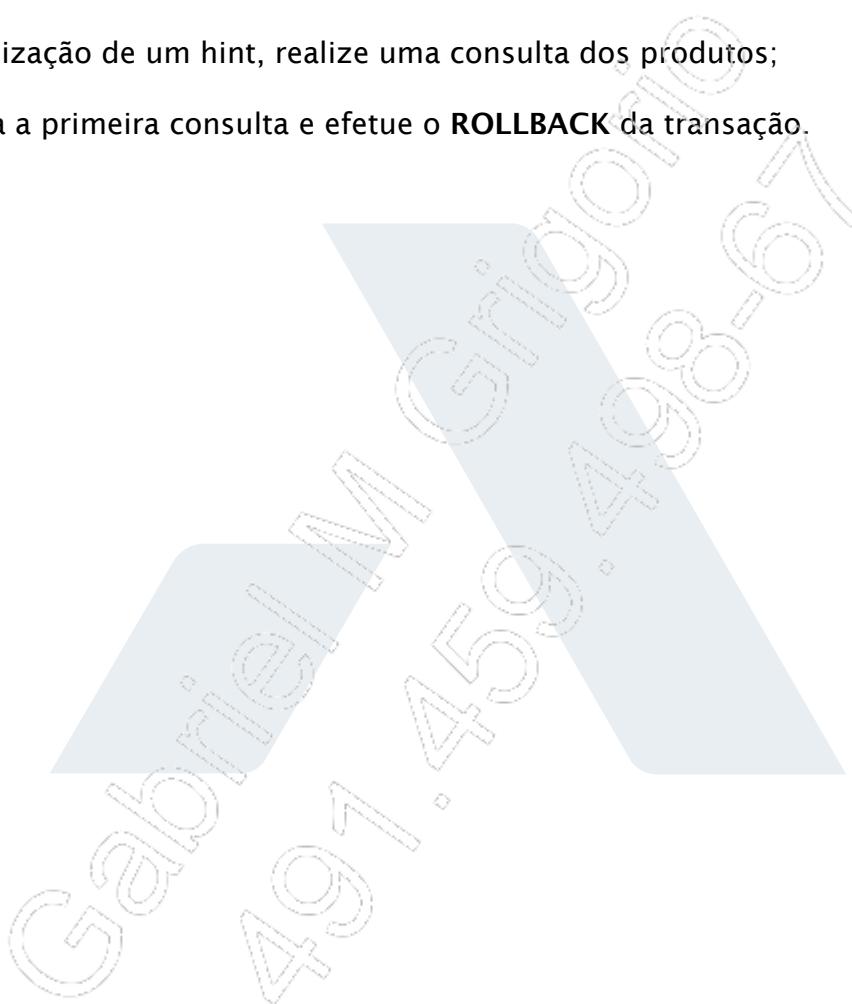
3. Abra uma nova consulta;

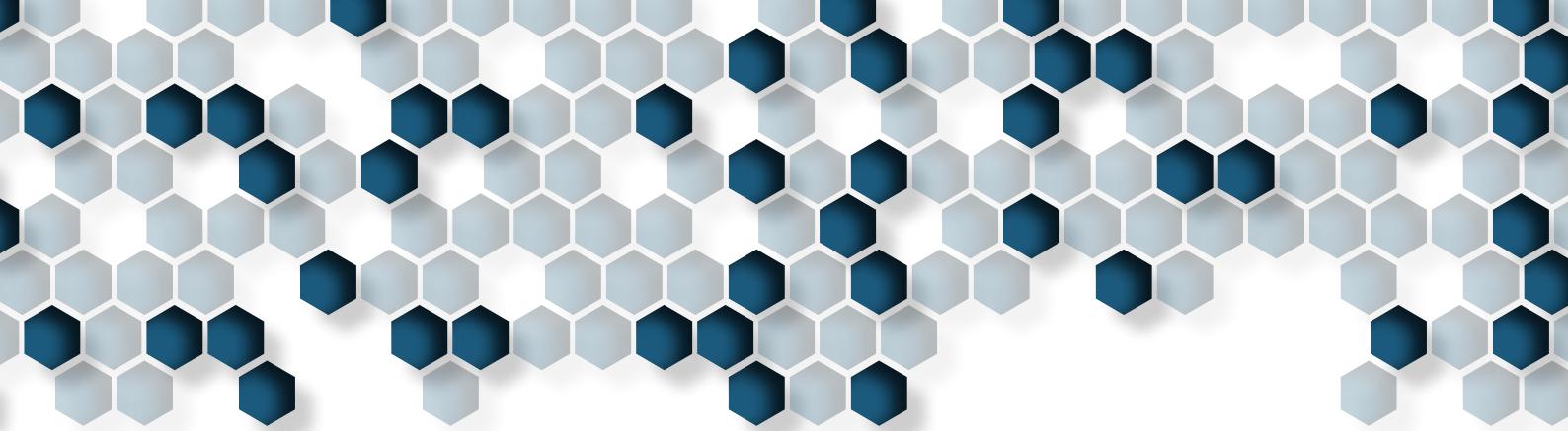
4. Execute o comando:

```
SET LOCK_TIMEOUT 5000
```

5. Faça uma consulta apresentando todos os produtos;

6. Execute a mesma consulta com um hint que permita a leitura de dados não confirmados;
7. Apresente os produtos que não estão bloqueados;
8. Abra uma nova consulta;
9. Customize a seção para leitura de dados não confirmados;
10. Sem a utilização de um hint, realize uma consulta dos produtos;
11. Volte para a primeira consulta e efetue o **ROLLBACK** da transação.





Acesso a recursos externos



- ◆ OPENROWSET;
 - ◆ BULK INSERT;
 - ◆ XML;
 - ◆ JSON.
- 

Esta Leitura Complementar refere-se ao conteúdo das Aulas 13 a 15.



1.1. Introdução

Há situações em que se faz necessária a transferência de dados entre regiões geograficamente diferentes. Para tanto, é preciso implementar uma solução que torne o ambiente mais gerenciável, visto que alguns ambientes, como o OLTP, requerem total consistência dos dados a todo o momento, ao passo que outros ambientes, como o de suporte a decisões, não exigem a frequente atualização dos dados.

Para que possamos acessar e trabalhar com os dados presentes em diversos servidores SQL Server e com dados heterogêneos que se encontram armazenados em outra origem de dados relacionais e não relacionais, devemos estabelecer um link com os servidores nos quais os dados estão presentes. No entanto, devemos ter em mente que o SQL Server apenas aceita este link caso haja um provedor OLE DB destinado à origem dos dados.

Caso os dados presentes em um servidor SQL Server remoto sejam acessados com frequência, este servidor deve ser definido como um servidor vinculado, bem como adicionado ao computador SQL Server local. A configuração do servidor remoto requer não apenas o estabelecimento de um link com a origem dos dados remotos, como também o estabelecimento da segurança entre os servidores local e remoto.

A escolha da ferramenta e da técnica para cada transferência de dados varia de situação para situação. Os critérios possíveis incluem: rápida implementação, transferências programadas regularmente e transformação de dados.

1.2. OPENROWSET

A função **OPENROWSET** inclui todas as informações de conexão necessárias para o acesso a um banco de dados local ou remoto a partir de uma fonte OLE DB. Vejamos algumas características de **OPENROWSET**:

- **OPENROWSET** pode ser referenciada como a tabela alvo de uma instrução **INSERT**, **UPDATE** ou **DELETE**;
- Diferentemente de uma consulta, que pode retornar múltiplos conjuntos de resultados, **OPENROWSET** retorna somente o primeiro conjunto de resultados;
- Suporta operações em massa através de um provedor BULK nativo que possibilita que os dados de um arquivo possam ser lidos e retornados como um rowset (conjunto de linhas).

Vale destacar que o acesso aos dados presentes em uma fonte de dados OLE DB requer o fornecimento das seguintes informações ao SQL Server:

- O nome do provedor OLE DB responsável por expor essa fonte;
- As informações necessárias ao provedor para que ele seja capaz de encontrar a fonte de dados (Connection String ou String de Conexão).

Essas informações podem ser fornecidas de maneiras distintas. Podemos fornecer ao SQL Server um ID de login que seja válido para a fonte de dados OLE DB. Podemos, ainda, fornecer o nome de um objeto que possa ser exposto como um rowset pela fonte de dados OLE DB, o qual é conhecido como tabela remota, bem como podemos fornecer uma query que possa ser enviada ao provedor OLE DB.

A sintaxe de **OPENROWSET** é a seguinte:

```
OPENROWSET
(   'provider_name' ,
    { 'datasource' ; 'user_id' ; 'password' | 'provider_string' } ,
    { [ catalog. ] [ schema. ] object | 'query' }
)
```

Em que:

- **provider_name**: Nome do OLE DB utilizado na conexão;
- **datasource**: Nome do banco de dados;
- **user_id**: Usuário para conexão;
- **password**: Senha para conexão;
- **provider_string**: String de conexão;
- **catalog.schema.object**: Nome do banco seguido do schema e do nome do objeto (tabela/view) que queremos acessar, lembrando que **catalog** e **schema** são opcionais;
- **query**: Instrução **SELECT**.

No exemplo a seguir, temos a utilização de **OPENROWSET**:

1. Habilite a visibilidade das opções avançadas:

```
EXEC sp_configure 'show advanced option', '1';
reconfigure
```

2. Habilite a utilização de **OPENROWSET**:

```
exec sp_configure 'Ad Hoc Distributed Queries',1
reconfigure
```

Feito isso, **OPENROWSET** estará habilitado para utilização no servidor;

3. Acesse o Excel e crie a tabela PESSOA no SQL Server:

```
CREATE DATABASE TESTE
GO
USE TESTE

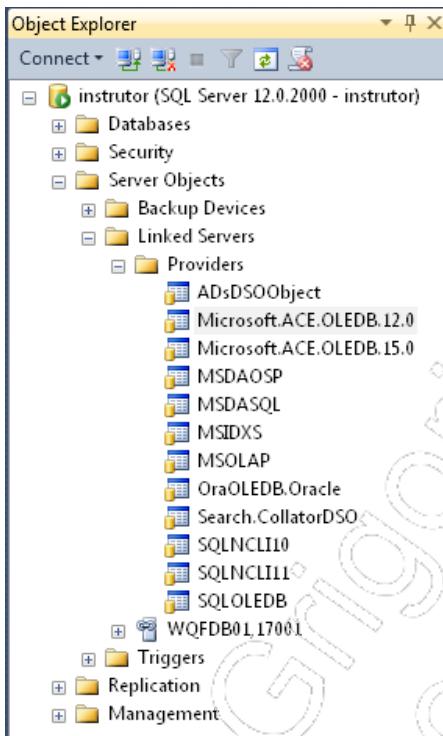
CREATE TABLE PESSOA
(
    COD INT,
    NOME VARCHAR(50)
)
```

The screenshot shows a Microsoft Excel window titled "PESSOA.xlsx - Microsoft Excel". The ribbon tabs are Início, Inserir, Layout, Fórmula, Dados, Revisão, Exibição, Desenho, and Arquivo. The "Fonte" tab is selected in the ribbon. The table "PESSOA" is displayed in the worksheet, with columns A and B. Row 1 contains the headers "COD" and "NOME". Rows 2 through 5 contain the data: 1 CARLOS MAGNO, 2 OLAVO C. OLIVEIRA, 3 MIKAIL BARROS, and 4 ARNALDO SILVA. The status bar at the bottom right shows "100%".

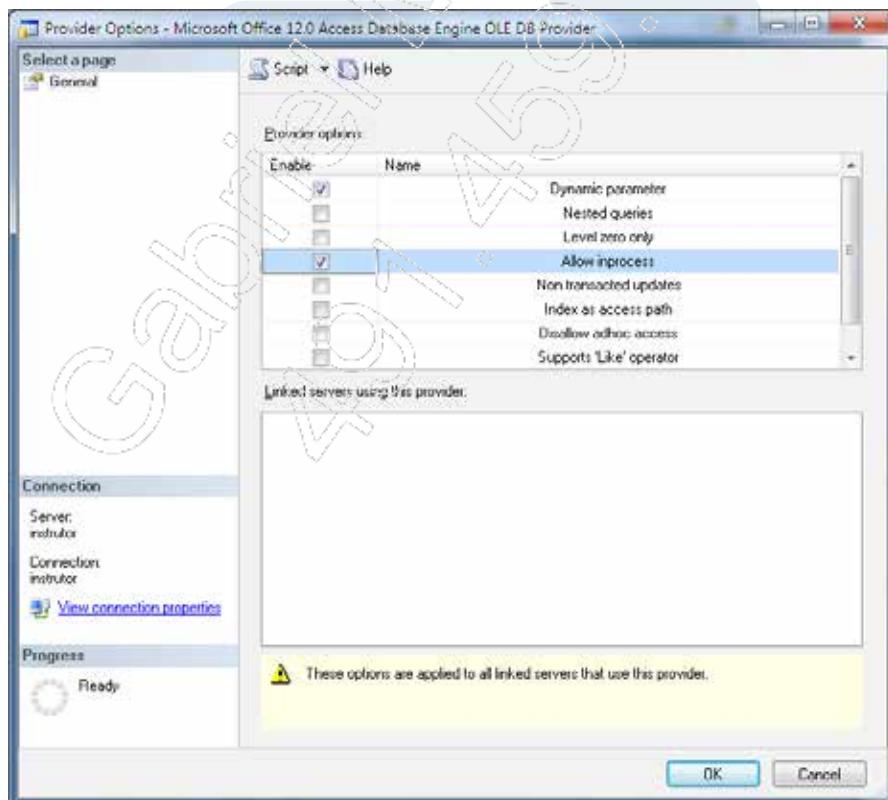
COD	NOME
1	CARLOS MAGNO
2	OLAVO C. OLIVEIRA
3	MIKAIL BARROS
4	ARNALDO SILVA

! Para arquivos do Excel ou Access de versão superior a 2007, é necessário configurar o provider.

4. Acesse o servidor e, em **Server Objects**, expanda **Linked Servers e Providers**:



5. Com o botão direito do mouse, selecione as propriedades;



6. Marque as opções **Dynamic parameter** e **Allow inprocess**;

7. Execute o seguinte trecho:

```
SELECT * FROM OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'Excel 12.0;Database=C:\DADOS\PESSOA.XLSX',
'SELECT COD, NOME FROM [NOMES$]')
```

8. Importe os dados da planilha para a tabela PESSOA do SQL Server:

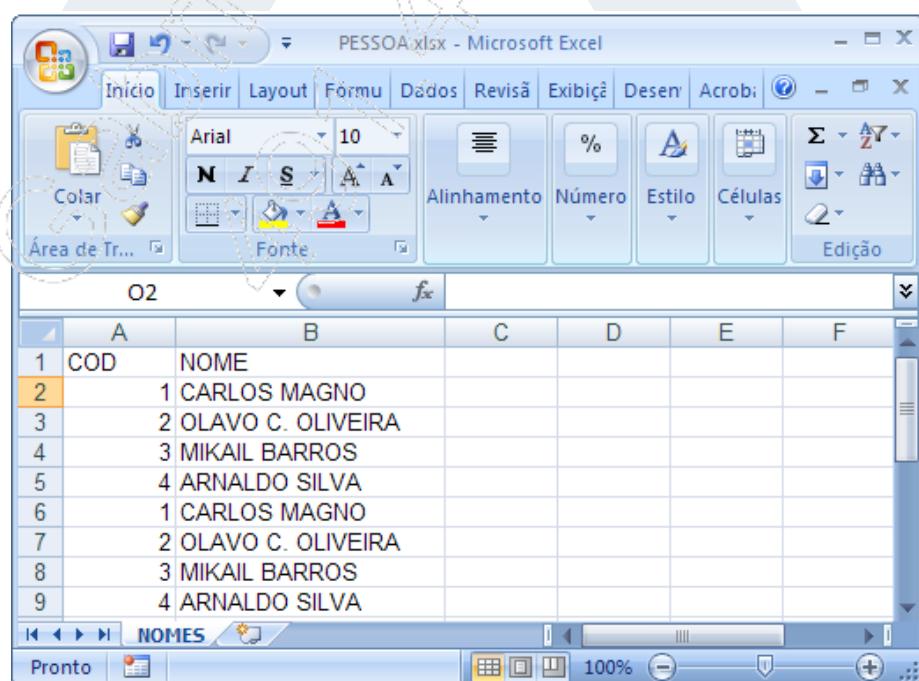
```
--IMPORTAR DADOS
INSERT INTO PESSOA
SELECT * FROM OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'Excel 12.0;Database=C:\DADOS\PESSOA.XLSX',
'SELECT COD, NOME FROM [NOMES$]')

SELECT * FROM PESSOA
```

9. Exporte dados para o Excel;

```
--EXPORTAR DADOS
INSERT INTO OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'Excel 12.0;Database=C:\DADOS\PESSOA.XLSX',
'SELECT COD, NOME FROM [NOMES$]')
SELECT * FROM PESSOA

SELECT * FROM OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'Excel 8.0;Database=C:\DADOS\PESSOA.XLS',
'SELECT COD, NOME FROM [NOMES$]');
SELECT * FROM PESSOA
```



A screenshot of Microsoft Excel showing a table of data. The table has two columns: 'COD' and 'NOME'. The data is as follows:

	A	B	C	D	E	F
1	COD	NOME				
2		1 CARLOS MAGNO				
3		2 OLAVO C. OLIVEIRA				
4		3 MIKAIL BARROS				
5		4 ARNALDO SILVA				
6		1 CARLOS MAGNO				
7		2 OLAVO C. OLIVEIRA				
8		3 MIKAIL BARROS				
9		4 ARNALDO SILVA				

10. Acesse o Access;

```
SELECT *
FROM OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'C:\Dados\Pedidos.accdb';
'admin''',PEDIDOS)
```

11. Crie uma **JOIN** entre a tabela do Access e a tabela do SQL Server:

```
SELECT P.* , C.NOME
FROM OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'C:\Dados\Pedidos.accdb';
'admin''',PEDIDOS) P
JOIN PEDIDOS.DBO.TB_CLIENTE C ON P.CODCLI = C.CODCLI
```

12. Importe os dados do Access.

```
CREATE TABLE TIPOPRODUTO
(COD_TIPO INT PRIMARY KEY, TIPO VARCHAR(30))

INSERT INTO TIPOPRODUTO
SELECT *
FROM OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'C:\Dados\Pedidos.accdb';
'admin''',TIPOPRODUTO)

SELECT * FROM TIPOPRODUTO
```

COD_TIPO	TIPO
1	0 NÃO CADASTRADO
2	1 ABRIDOR
3	2 PORTA LAPIS
4	3 REGUA
5	4 ACES.CHAVEIRO
6	5 CANETA
7	6 CHAVEIRO
8	7 BOTTON
9	8 MISTURADOR DE...
10	9 PORTA MOEDAS
11	10 CARTAO PVC
12	15 YO-YO
13	100 MAQUINAS
14	101 CARGAS P/ CANE...
15	102 ACESSORIOS P/C...
16	103 MATL DIVERSOS

1.3. BULK INSERT

O mecanismo de **BULK INSERT** permite a inserção de dados em massa vindos de arquivos de texto. A seguir, temos a sintaxe do comando **BULK INSERT**:

```
BULK INSERT
    [ database_name . [ schema_name ] . | schema_name . ] [ table_
name | view_name ]
    FROM 'data_file'
    [ WITH
    (
        [ [ , ] BATCHSIZE = batch_size ]
        [ [ , ] CHECK_CONSTRAINTS ]
        [ [ , ] CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' } ]
        [ [ , ] DATAFILETYPE =
            { 'char' | 'native'| 'widechar' | 'widennative' } ]
        [ [ , ] FIELDTERMINATOR = 'field_terminator' ]
        [ [ , ] FIRSTROW = first_row ]
        [ [ , ] FIRE_TRIGGERS ]
        [ [ , ] FORMATFILE = 'format_file_path' ]
        [ [ , ] KEEPIDENTITY ]
        [ [ , ] KEENNULLS ]
        [ [ , ] KILOBYTES_PER_BATCH = kilobytes_per_batch ]
        [ [ , ] LASTROW = last_row ]
        [ [ , ] MAXERRORS = max_errors ]
        [ [ , ] ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) ]
        [ [ , ] ROWS_PER_BATCH = rows_per_batch ]
        [ [ , ] ROWTERMINATOR = 'row_terminator' ]
        [ [ , ] TABLOCK ]
        [ [ , ] ERRORFILE = 'file_name' ]
    )]
```

Vejamos as opções a seguir:

- **Database_name**: Nome do banco de dados que sofrerá a carga de dados;
- **Schema_name**: Nome do schema que contém a tabela e/ou visão. Caso o mesmo schema do usuário esteja realizando a operação, esse dado não precisa ser informado;
- **Table_name**: Nome da tabela ou visão do banco de dados que sofrerá o processo de carga de dados;
- **Data_file**: Nome do arquivo de dados que será lido para realização da carga, incluindo o drive, o diretório e subdiretórios, se houverem;
- **BATCHSIZE = [batch_size]**: Pode-se especificar a quantidade de linhas que será tratada como uma transação. Em caso de falhas ou de sucesso, este parâmetro indica o tamanho da transação;

- **CHECK CONSTRAINTS:** Indica que deverão ser tratadas as constraints da tabela que recebem os dados. Constraints são verificadas no momento do carregamento caso essa opção seja utilizada. Por padrão, as constraints são ignoradas no processo de carga;
- **CODE_PAGE:** Indica o tipo de código de página de dados;
- **DATAFILE_TYPE [= {'CHAR' |'NATIVE' | 'WIDECHAR' | 'WIDENATIVE'}]:**
 - Caso o valor **CHAR** seja utilizado para a opção, um arquivo de dados no formato caractere será copiado;
 - Caso o valor **NATIVE** seja utilizado, os tipos de dados nativos do banco para copiar os arquivos serão empregados;
 - Caso o valor seja **WIDECHAR** para a opção, um arquivo de dados no formato UNICODE será copiado;
 - Caso o valor seja **WIDENATIVE** para a opção, será feita uma cópia seguindo os tipos de dados nativos, exceto **char**, **varchar** e **text**, que serão armazenados como caracteres padrão UNICODE.
- **ERRORFILE = 'file_name':** Especifica o arquivo de erro a ser gerado quando as transformações de dados não puderem ser realizadas. Caso o arquivo já exista depois de uma prévia execução, ocorrerá um erro de execução. Junto com a criação deste arquivo de erro, será gerado um arquivo com a extensão **.ERROR.txt** para referenciar cada linha e fornecer o diagnóstico dos erros, para serem corrigidos;
- **FIELDTERMINATOR [= 'field_terminator']:** O valor padrão para esta opção é **\t**, porém, pode-se definir outro caractere separador de campos;
- **FIRST_ROW [= first_row]:** Com valor padrão 1, indica o número da primeira linha a ser lida;
- **FIRE_TRIGGERS:** Caso os gatilhos não sejam informados, eles não serão disparados no momento da carga de dados;
- **KEEPIDENTITY:** Caso especifique esta opção, o arquivo de dados conterá os valores da coluna do tipo **Identity**. Em caso de omissão, novos valores serão atribuídos a colunas do tipo **Identity**;
- **KEEPNULLS:** As colunas da tabela e/ou visão serão carregadas como nulas caso não sejam informadas no arquivo;
- **KILOBYTES_PER_BATCH [= kilobyte_per_batch]:** Quantidade em KB, correspondente a cada carga de dados realizada;
- **LASTROW [=last_row]:** Indica o número da última linha a ser processada. O valor padrão é 0 e isso indica que todas as linhas do arquivo serão processadas;

- **MAXERRORS [=max_errors]**: Número máximo de erros admitidos pelo processo de carga. O valor padrão é 0, o que indica que nenhum erro será admitido;
- **ORDER ({column [ASC|DESC]},[....n])**: Especifica a ordem dos arquivos de dados em caso de existirem mais do que um. Caso os dados do arquivo de dados estejam na ordem do índice clustered na tabela de destino, o processo de **BULK INSERT** terá uma grande melhoria de performance. Se os dados estiverem em outra ordem, a cláusula **ORDER** será ignorada;
- **ROWTERMINATOR [=‘row_terminator’]**: Indica o caractere terminador de registro. Caso não seja informado um caractere, \n é o valor padrão;
- **ROW_PER_BATCH**: Define a quantidade de linhas que deverão ser lidas por cada transação;
- **TABLOCK**: Define o travamento do tipo **TABLOCK** (travamento de toda a tabela) como tipo de travamento da tabela (**LOCK**).

Processos de cargas tendem a ser mais eficientes quando o volume de leitura e manipulação é maior que um registro. Com isso, as transações se tornam maiores e o volume de I/O em disco (para leitura ou gravação de arquivo) e nos arquivos do banco de dados tende a ser menor.

A seguir, vejamos um exemplo:

- **Criação da tabela TESTE_BULK_INSERT**

```
CREATE TABLE TESTE_BULK_INSERT
( CODIGO      INT,
  NOME        VARCHAR(40),
  DATA_NASCIMENTO DATETIME )
```

- **Execução do comando BULK INSERT**

```
BULK INSERT TESTE_BULK_INSERT
  FROM 'C:\DADOS\BULK_INSERT.txt'
  WITH
  (
    FIELDTERMINATOR = ';' ,
    ROWTERMINATOR = '\n' ,
    codepage = 'acp'
  )
```

- Consulta da tabela

```
SELECT * FROM TESTE_BULK_INSERT
```

1.4. XML

XML (eXtensible Markup Language) é uma linguagem que permite a troca de informações de um modo mais simples e prático. O SQL suporta esta linguagem, utilizada em campos e variáveis.

1.4.1. FOR XML

Um resultado de uma consulta pode ser exportado para um formato XML. A cláusula para esta ação é o **FOR XML**. O **FOR XML** possui os seguintes tipos:

- **RAW**: O resultado será apresentado por linha;
- **AUTO**: A saída do XML será desenvolvida pelo SQL;
- **EXPLICIT**: Este tipo permite que seja desenvolvida a forma da saída;
- **PATH**: Permite o mesmo modo do **EXPLICIT**, porém mais simples.

A seguir, vejamos como é feita a exportação dos dados para XML:

- **XML RAW sem tag raiz (que não abre no browser) e um atributo para cada campo**

```
USE PEDIDOS
GO

SELECT ID_PRODUTO, COD_TIPO, DESCRICAO, PRECO_VENDA
FROM TB_PRODUTO
FOR XML RAW
```

O Resultado é apresentado em um link XML:



The screenshot shows the SSMS Results pane displaying XML output. The results table has two columns: 'Results' and 'Messages'. The 'Results' column contains the XML data:

	Results
1	<row ID_PRODUTO='1' COD_TIPO='1' DESCRICAO='ABRID...'>

O XML não possui uma tag principal (raiz ou ROOT).

```
<row ID_PRODUTO="1" COD_TIPO="1" DESCRICAO="ABRIDOR SACA-ROLHA TESTE ADO" PRECO_VENDA="0.0475" />
<row ID_PRODUTO="2" COD_TIPO="2" DESCRICAO="PORTA-LAPIS COM PEZINHO" PRECO_VENDA="1.5330" />
<row ID_PRODUTO="3" COD_TIPO="3" DESCRICAO="REGUA DE 20 CM" PRECO_VENDA="0.9610" />
<row ID_PRODUTO="4" COD_TIPO="4" DESCRICAO="PENTE PEQUENO" PRECO_VENDA="1.9219" />
<row ID_PRODUTO="5" COD_TIPO="4" DESCRICAO="PENTE JACARE" PRECO_VENDA="3.1117" />
<row ID_PRODUTO="6" COD_TIPO="5" DESCRICAO="SPECIAL PEN GOLD" PRECO_VENDA="4.2099" />
```

- XML RAW com tag raiz e um atributo para cada campo

```
SELECT ID_PRODUTO, COD_TIPO, DESCRICAO, PRECO_VENDA
FROM TB_PRODUTO
--      tag da linha , tag principal
FOR XML RAW('Produto'), ROOT('Produtos')
```

Resultado:

```
<Produtos>
<Produto ID_PRODUTO="1" COD_TIPO="1" DESCRICAO="ABRIDOR SACA-ROLHA TESTE ADO" PRECO_VENDA="0.0475" />
<Produto ID_PRODUTO="2" COD_TIPO="2" DESCRICAO="PORTA-LAPIS COM PEZINHO" PRECO_VENDA="1.5330" />
<Produto ID_PRODUTO="3" COD_TIPO="3" DESCRICAO="REGUA DE 20 CM" PRECO_VENDA="0.9610" />
<Produto ID_PRODUTO="4" COD_TIPO="4" DESCRICAO="PENTE PEQUENO" PRECO_VENDA="1.9219" />
<Produto ID_PRODUTO="5" COD_TIPO="4" DESCRICAO="PENTE JACARE" PRECO_VENDA="3.1117" />
<Produto ID_PRODUTO="6" COD_TIPO="5" DESCRICAO="SPECIAL PEN GOLD" PRECO_VENDA="4.2099" />
```

Execute o comando a seguir para atualizar a descrição do produto:

```
UPDATE TB_PRODUTO SET DESCRICAO = 'ABRIDOR SACA & ROLHA' WHERE ID_PRODUTO = 1
```

Execute novamente a consulta e verifique o resultado e como foi colocado o sinal de &.

```
<Produtos>
<Produto ID_PRODUTO="1" COD_TIPO="1" DESCRICAO="ABRIDOR SACA & ROLHA" PRECO_VENDA="0.0475" />
```

- XML RAW com tag raiz e um elemento para cada campo

```
SELECT ID_PRODUTO, COD_TIPO, DESCRICAO, PRECO_VENDA
FROM TB_PRODUTO
FOR XML RAW('Produto'), ROOT('Produtos'), ELEMENTS
```

O resultado é exibido a seguir. Observe os produtos ID 36 e 37. O campo COD_TIPO é nulo e a tag simplesmente some:

```
<Produto>
  <ID_PRODUTO>35</ID_PRODUTO>
  <COD_TIPO>5</COD_TIPO>
  <DESCRICAO>CANETA STAR II</DESCRICAO>
  <PRECO_VENDA>3.9354</PRECO_VENDA>
</Produto>
<Produto>
  <ID_PRODUTO>36</ID_PRODUTO>
  <DESCRICAO>KEY RING BIG LOCK</DESCRICAO>
</Produto>
<Produto>
  <ID_PRODUTO>37</ID_PRODUTO>
  <DESCRICAO>SPECIAL KEY RING</DESCRICAO>
</Produto>
<Produto>
  <ID_PRODUTO>38</ID_PRODUTO>
  <COD_TIPO>100</COD_TIPO>
  <DESCRICAO>MAQUINA VARETAR</DESCRICAO>
  <PRECO_VENDA>4.3014</PRECO_VENDA>
</Produto>
```

- XML RAW com tag raiz, um elemento para cada campo e mostrando as tags com valores nulos

```
SELECT ID_PRODUTO, COD_TIPO, DESCRICAO, PRECO_VENDA
  FROM TB_PRODUTO
 FOR XML RAW('Produto'), ROOT('Produtos'), ELEMENTS XSINIL
```

A seguir, temos o resultado. Observe os produtos com ID 36 e 37:

```
<Produto>
  <ID_PRODUTO>35</ID_PRODUTO>
  <COD_TIPO>5</COD_TIPO>
  <DESCRICAO>CANETA STAR II</DESCRICAO>
  <PRECO_VENDA>3.9354</PRECO_VENDA>
</Produto>
<Produto>
  <ID_PRODUTO>36</ID_PRODUTO>
  <COD_TIPO xsi:nil="true" />
  <DESCRICAO>KEY RING BIG LOCK</DESCRICAO>
  <PRECO_VENDA xsi:nil="true" />
</Produto>
<Produto>
  <ID_PRODUTO>37</ID_PRODUTO>
  <COD_TIPO xsi:nil="true" />
  <DESCRICAO>SPECIAL KEY RING</DESCRICAO>
  <PRECO_VENDA xsi:nil="true" />
</Produto>
<Produto>
  <ID_PRODUTO>38</ID_PRODUTO>
  <COD_TIPO>100</COD_TIPO>
  <DESCRICAO>MAQUINA VARETAR</DESCRICAO>
  <PRECO_VENDA>4.3014</PRECO_VENDA>
</Produto>
```

- XML AUTO com tag raiz e um elemento para cada campo

```
-- o nome da coluna dá nome ao elemento de cada campo
SELECT Empregado.CODFUN AS Código, Empregado.NOME, Empregado.DATA_
ADMISSAO, Empregado.SALARIO
-- o apelido da tabela dá nome à tag de linha
FROM TB_EMPREGADO Empregado
FOR XML AUTO, ROOT('Empregados'), ELEMENTS XSINIL
```

Resultado:

```
<Empregados xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<Empregado>
  <Código>1</Código>
  <NOME>OLAVO TRINDADE</NOME>
  <DATA_ADMISSAO>1986-10-05T00:00:00</DATA_ADMISSAO>
  <SALARIO>3000.00</SALARIO>
</Empregado>
<Empregado>
  <Código>2</Código>
  <NOME>JOSE REIS</NOME>
  <DATA_ADMISSAO>1987-05-02T00:00:00</DATA_ADMISSAO>
  <SALARIO>600.00</SALARIO>
</Empregado>
```

- XML AUTO com tag raiz, um elemento para cada campo e com lista de subitens (mestre x detalhe)

```
SELECT Empregado.CODFUN, Empregado.NOME, Empregado.DATA_ADMISSAO,
Empregado.SALARIO, Dependente.CODDEP, Dependente.NOME, Dependente.
DATA_NASCIMENTO
FROM TB_EMPREGADO Empregado JOIN TB_DEPENDENTE Dependente ON
Empregado.CODFUN = Dependente.CODFUN
FOR XML AUTO, ROOT('Empregados'), ELEMENTS XSINIL
```

Resultado:

```
<Empregados xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Empregado>
    <CODFUN>1</CODFUN>
    <NOME>OLAVO TRINDADE</NOME>
    <DATA ADMISSAO>1986-10-05T00:00:00</DATA ADMISSAO>
    <SALARIO>3000.00</SALARIO>
    <Dependente>
      <CODDEP>1</CODDEP>
      <NOME>PEDRO</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
  </Empregado>
  <Empregado>
    <CODFUN>2</CODFUN>
    <NOME>JOSE REIS</NOME>
    <DATA ADMISSAO>1987-05-02T00:00:00</DATA ADMISSAO>
    <SALARIO>600.00</SALARIO>
    <Dependente>
      <CODDEP>1</CODDEP>
      <NOME>CARLOS</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
    <Dependente>
      <CODDEP>2</CODDEP>
      <NOME>ALBERTO</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
    <Dependente>
      <CODDEP>3</CODDEP>
      <NOME>MARTINS</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
    <Dependente>
      <CODDEP>4</CODDEP>
      <NOME>MANOEL</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
    <Dependente>
      <CODDEP>5</CODDEP>
      <NOME>PAULO</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
    <Dependente>
      <CODDEP>6</CODDEP>
      <NOME>MARIA</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
```

- XML AUTO com tag raiz, um elemento para cada campo e com lista de subitens (mestre x detalhe de dois níveis)

```
SELECT
    Cliente.CODCLI AS IdCliente, Cliente.NOME AS Cliente,
    Pedidos.NUM_PEDIDO AS IdPedido, Pedidos.VLR_TOTAL AS VlrPedido,
    Pedidos.DATA_EMISSAO AS Emissao
FROM TB_CLIENTE Cliente JOIN TB_PEDIDO Pedidos ON Cliente.CODCLI =
Pedidos.CODCLI
WHERE Pedidos.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
FOR XML AUTO, ROOT('Clientes')
```

Resultado:

```
<Clientes>
<Cliente IdCliente="15" Cliente="A. LUZ DE ALMEIDA">
    <Pedidos IdPedido="3157" VlrPedido="503.71" Emissao="2014-01-20T00:00:00" />
</Cliente>
<Cliente IdCliente="27" Cliente="ABB LIMA FILHO LTDA.AAAA">
    <Pedidos IdPedido="3118" VlrPedido="920.19" Emissao="2014-01-16T00:00:00" />
</Cliente>
<Cliente IdCliente="461" Cliente="ADEMILSON FRANCISCO COSTA">
    <Pedidos IdPedido="3052" VlrPedido="2089.08" Emissao="2014-01-09T00:00:00" />
</Cliente>
<Cliente IdCliente="26" Cliente="ADEMILSON RANGEL MACHADO">
    <Pedidos IdPedido="2989" VlrPedido="3493.94" Emissao="2014-01-03T00:00:00" />
</Cliente>
<Cliente IdCliente="371" Cliente="ADEMIR SANTO SARRAFINI">
    <Pedidos IdPedido="3244" VlrPedido="30.20" Emissao="2014-01-29T00:00:00" />
</Cliente>
<Cliente IdCliente="32" Cliente="ADILSON CAVALCANTE DE OLIVEIRA-ME">
    <Pedidos IdPedido="3045" VlrPedido="2058.61" Emissao="2014-01-09T00:00:00" />
    <Pedidos IdPedido="3126" VlrPedido="1762.68" Emissao="2014-01-16T00:00:00" />
    <Pedidos IdPedido="3217" VlrPedido="1959.96" Emissao="2014-01-27T00:00:00" />
</Cliente>
```

Outra opção é a utilização de elementos no lugar de atributos:

```
SELECT
    Cliente.CODCLI AS IdCliente, Cliente.NOME AS Cliente,
    Pedidos.NUM_PEDIDO AS IdPedido,
    Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS Emissao
FROM TB_CLIENTE Cliente JOIN TB_PEDIDO Pedidos ON Cliente.CODCLI =
Pedidos.CODCLI
WHERE Pedidos.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
FOR XML AUTO, ROOT('Clientes'), ELEMENTS
```

Resultado:

```
<Clientes>
  <Cliente>
    <IdCliente>15</IdCliente>
    <Cliente>A. LUZ DE ALMEIDA</Cliente>
    <Pedidos>
      <IdPedido>3157</IdPedido>
      <VlrPedido>503.71</VlrPedido>
      <Emissao>2014-01-20T00:00:00</Emissao>
    </Pedidos>
  </Cliente>
  <Cliente>
    <IdCliente>27</IdCliente>
    <Cliente>ABB LIMA FILHO LTDA.AAAA</Cliente>
    <Pedidos>
      <IdPedido>3118</IdPedido>
      <VlrPedido>920.19</VlrPedido>
      <Emissao>2014-01-16T00:00:00</Emissao>
    </Pedidos>
  </Cliente>
  <Cliente>
    <IdCliente>461</IdCliente>
    <Cliente>ADEMILSON FRANCISCO COSTA</Cliente>
    <Pedidos>
      <IdPedido>3052</IdPedido>
      <VlrPedido>2089.08</VlrPedido>
      <Emissao>2014-01-09T00:00:00</Emissao>
    </Pedidos>
  </Cliente>
```

- XML AUTO com tag raiz, um elemento para cada campo e com lista de subitens (mestre x detalhe de três níveis)

```
SELECT
  Cliente.CODCLI AS IdCliente, Cliente.NOME AS Cliente,
  Pedidos.NUM_PEDIDO AS IdPedido,
  Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS Emissao,
  Itens.NUM_ITEM AS IdItem, Itens.ID_PRODUTO AS IdProduto,
  Itens.QUANTIDADE AS Quantidade, Itens.PR_UNITARIO AS PrUnitario
FROM TB_CLIENTE Cliente
JOIN TB_PEDIDO Pedidos ON Cliente.CODCLI = Pedidos.CODCLI
JOIN TB_ITENSPEDIDO Itens ON Pedidos.NUM_PEDIDO = Itens.NUM_PEDIDO
WHERE Pedidos.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
-- Importante para ter um resultado correto
ORDER BY Cliente.NOME, Pedidos.NUM_PEDIDO
FOR XML AUTO, ROOT('Clientes')
```

Resultado:

```
<Clientes>
<Cliente IdCliente="15" Cliente="A. LUZ DE ALMEIDA">
    <Pedidos IdPedido="3157" VlrPedido="503.71" Emissao="2014-01-20T00:00:00">
        <Itens IdItem="1" IdProduto="31" Quantidade="67" PrUnitario="0.8100" />
        <Itens IdItem="2" IdProduto="38" Quantidade="122" PrUnitario="1.8800" />
        <Itens IdItem="3" IdProduto="57" Quantidade="205" PrUnitario="0.8800" />
        <Itens IdItem="4" IdProduto="15" Quantidade="32" PrUnitario="1.2400" />
    </Pedidos>
</Cliente>
<Cliente IdCliente="27" Cliente="ABB LIMA FILHO LTDA.AAAA">
    <Pedidos IdPedido="3118" VlrPedido="920.19" Emissao="2014-01-16T00:00:00">
        <Itens IdItem="1" IdProduto="56" Quantidade="406" PrUnitario="1.0900" />
        <Itens IdItem="2" IdProduto="3" Quantidade="113" PrUnitario="0.4200" />
        <Itens IdItem="3" IdProduto="34" Quantidade="162" PrUnitario="0.2900" />
        <Itens IdItem="4" IdProduto="53" Quantidade="176" PrUnitario="0.6200" />
        <Itens IdItem="5" IdProduto="39" Quantidade="88" PrUnitario="0.4000" />
        <Itens IdItem="6" IdProduto="65" Quantidade="86" PrUnitario="0.2000" />
        <Itens IdItem="7" IdProduto="9" Quantidade="79" PrUnitario="1.2800" />
        <Itens IdItem="8" IdProduto="65" Quantidade="625" PrUnitario="0.2000" />
    </Pedidos>
</Cliente>
```

- XML EXPLICIT (XML é montado explicitamente no comando SELECT)

```
SELECT 1 AS Tag, NULL AS Parent,
--      conteúdo      tag      id atributo --><Empregado
Codigo='1">
    CODFUN      [Empregado!1!Codigo],
--      dentro da tag Empregado criar elemento "Nome"
    NOME AS [Empregado!1!Funcionario!ELEMENT],
--      dentro da tag Empregado criar elemento "Salario"
    SALARIO [Empregado!1!Renda!ELEMENT],
--      dentro da tag Empregado criar elemento "DataAdm"
    DATA_ADMISSAO [Empregado!1!DataAdm!ELEMENT]
FROM TB_EMPREGADO
ORDER BY CODFUN
FOR XML EXPLICIT, ROOT('Empregados')
```

Resultado:

```
<Empregados>
  <Empregado Código="1">
    <Funcionario>OLAVO TRINDADE</Funcionario>
    <Renda>3000.00</Renda>
    <DataAdm>1986-10-05T00:00:00</DataAdm>
  </Empregado>
  <Empregado Código="2">
    <Funcionario>JOSE REIS</Funcionario>
    <Renda>600.00</Renda>
    <DataAdm>1987-05-02T00:00:00</DataAdm>
  </Empregado>
  <Empregado Código="3">
    <Funcionario>MARCELO SOARES</Funcionario>
    <Renda>2400.00</Renda>
    <DataAdm>1986-10-05T00:00:00</DataAdm>
  </Empregado>
  <Empregado Código="4">
    <Funcionario>PAULO CESAR JUNIOR</Funcionario>
    <Renda>600.00</Renda>
    <DataAdm>1987-05-06T00:00:00</DataAdm>
  </Empregado>
```

- XML EXPLICIT

```
SELECT 1 AS Tag, NULL AS Parent,
-- conteúdo      tag      id elemento
--   <Código>1</Código>
  CODFUN      [Empregado!1!Código!ELEMENT],
-- dentro da tag Empregado criar elemento "Nome"
  NOME AS [Empregado!1!Funcionario!ELEMENT],
-- dentro da tag Empregado criar elemento "Salario"
  SALARIO [Empregado!1!Renda!ELEMENT],
-- dentro da tag Empregado criar elemento "DataAdm"
  DATA_ADMISSAO [Empregado!1!DataAdm!ELEMENT]
FROM TB_EMPREGADO
ORDER BY CODFUN
FOR XML EXPLICIT, ROOT('Empregados')
```

Resultado:

```
<Empregados>
<Empregado>
    <Codigo>1</Codigo>
    <Funcionario>OLAVO TRINDADE</Funcionario>
    <Renda>3000.00</Renda>
    <DataAdm>1986-10-05T00:00:00</DataAdm>
</Empregado>
<Empregado>
    <Codigo>2</Codigo>
    <Funcionario>JOSE REIS</Funcionario>
    <Renda>600.00</Renda>
    <DataAdm>1987-05-02T00:00:00</DataAdm>
</Empregado>
<Empregado>
    <Codigo>3</Codigo>
    <Funcionario>MARCELO SOARES</Funcionario>
    <Renda>2400.00</Renda>
    <DataAdm>1986-10-05T00:00:00</DataAdm>
</Empregado>
```

- **XML EXPLICIT (idem ao anterior, mas com tag oculta)**

```
SELECT 1 AS Tag, NULL AS Parent,
--      atributo ficará oculto
      CODFUN [Empregado!1!Codigo!HIDE],
--      dentro da tag Empregado criar tag "Nome"
      NOME AS [Empregado!1!Funcionario!ELEMENT],
--      dentro da tag Empregado criar tag "Salario"
      SALARIO [Empregado!1!Renda!ELEMENT],
--      dentro da tag Empregado criar tag "DataAdm"
      DATA_ADMISSAO [Empregado!1!DataAdm!ELEMENT]
FROM TB_EMPREGADO
ORDER BY CODFUN
FOR XML EXPLICIT, ROOT('Empregados')
```

Resultado:

```
<Empregados>
  <Empregado>
    <Funcionario>OLAVO TRINDADE</Funcionario>
    <Renda>3000.00</Renda>
    <DataAdm>1986-10-05T00:00:00</DataAdm>
  </Empregado>
  <Empregado>
    <Funcionario>JOSE REIS</Funcionario>
    <Renda>600.00</Renda>
    <DataAdm>1987-05-02T00:00:00</DataAdm>
  </Empregado>
  <Empregado>
    <Funcionario>MARCELO SOARES</Funcionario>
    <Renda>2400.00</Renda>
    <DataAdm>1986-10-05T00:00:00</DataAdm>
  </Empregado>
```

- XML EXPLICIT (forma mais detalhista de gerar o XML)

```
SELECT 1 AS TAG, NULL AS PARENT,
  -- gera a tag principal no primeiro nível
  '' [Empregados!1],
  -- define o restante da estrutura do XML
  NULL AS [Empregado!2!CODFUN],
  NULL AS [Empregado!2!Nome!ELEMENT],
  NULL [Empregado!2!Salario!ELEMENT],
  NULL [Empregado!2!DataAdm!ELEMENT]
UNION ALL
  -- fornece os dados definidos na estrutura anterior
  -- Tag de nível 2, o parent desta Tag é a Tag do nível anterior
  SELECT 2 AS Tag, 1 AS Parent, NULL,
    CODFUN,
    NOME,
    SALARIO,
    DATA_ADMISSAO
  FROM TB_EMPREGADO
  FOR XML EXPLICIT , ROOT('Empregados')
```

Resultado:

```
<Empregados>
  <Empregados>
    <Empregado CODFUN="1">
      <Nome>OLAVO TRINDADE</Nome>
      <Salario>3000.00</Salario>
      <DataAdm>1986-10-05T00:00:00</DataAdm>
    </Empregado>
    <Empregado CODFUN="2">
      <Nome>JOSE REIS</Nome>
      <Salario>600.00</Salario>
      <DataAdm>1987-05-02T00:00:00</DataAdm>
    </Empregado>
    <Empregado CODFUN="3">
      <Nome>MARCELO SOARES</Nome>
      <Salario>2400.00</Salario>
      <DataAdm>1986-10-05T00:00:00</DataAdm>
    </Empregado>
```

- **XML EXPLICIT (Mestre x Detalhe)**

```
SELECT 1 AS TAG, NULL AS PARENT,
  -- gera a tag principal no primeiro nível
  '' [Empregados!1],
  -- define o restante da estrutura
  NULL AS [Empregado!2!CODFUN],
  NULL AS [Empregado!2!Nome!ELEMENT],
  NULL [Empregado!2!Salario!ELEMENT],
  NULL [Empregado!2!DataAdm!ELEMENT],
  -- subnível vinculado a cada empregado
  NULL [Dependente!3!CodFun!HIDE],
  NULL [Dependente!3!CodDep],
  NULL [Dependente!3!Nome!ELEMENT],
  NULL [Dependente!3!DataNasc!ELEMENT]
UNION ALL
  -- fornece os dados definidos na estrutura anterior
  -- Tag de nível 2, o parent desta Tag é a Tag do nível anterior (1)
SELECT 2 AS Tag, 1 AS Parent, NULL,
  CODFUN,
  NOME,
  SALARIO,
  DATA_ADMISSAO, NULL, NULL, NULL, NULL
FROM TB_EMPREGADO
UNION ALL
  -- Tag de nível 3, o parent desta Tag é a Tag do nível anterior (2)
SELECT 3 AS Tag, 2 AS Parent, NULL,
  E.CODFUN,
  E.NOME,
  E.SALARIO,
```

```
E.DATA ADMISSAO,  
D.CODFUN, D.CODDEP, D.NOME, D.DATA NASCIMENTO  
FROM TB_EMPREGADO E  
JOIN TB_DEPENDENTE D ON E.CODFUN = D.CODFUN  
ORDER BY [Empregado!2!CODFUN],[Dependente!3!CodFun!HIDE]  
FOR XML EXPLICIT;
```

Resultado:

```
<Empregados>  
  <Empregado CODFUN="1">  
    <Nome>OLAVO TRINDADE</Nome>  
    <Salario>3000.00</Salario>  
    <DataAdm>1986-10-05T00:00:00</DataAdm>  
    <Dependente CodDep="1">  
      <Nome>PEDRO</Nome>  
    </Dependente>  
  </Empregado>  
  <Empregado CODFUN="2">  
    <Nome>JOSE REIS</Nome>  
    <Salario>600.00</Salario>  
    <DataAdm>1987-05-02T00:00:00</DataAdm>  
    <Dependente CodDep="1">  
      <Nome>CARLOS</Nome>  
    </Dependente>  
    <Dependente CodDep="2">  
      <Nome>ALBERTO</Nome>  
    </Dependente>  
    <Dependente CodDep="3">  
      <Nome>MARTINS</Nome>  
    </Dependente>  
    <Dependente CodDep="4">  
      <Nome>MANOEL</Nome>  
    </Dependente>  
    <Dependente CodDep="5">  
      <Nome>PAULO</Nome>  
    </Dependente>  
    <Dependente CodDep="6">  
      <Nome>MARIA</Nome>  
    </Dependente>  
  </Empregado>  
  <Empregado CODFUN="3">  
    <Nome>MARCELO SOARES</Nome>  
    <Salario>2400.00</Salario>  
    <DataAdm>1986-10-05T00:00:00</DataAdm>  
    <Dependente CodDep="1">  
      <Nome>LUIZA</Nome>  
    </Dependente>  
  </Empregado>
```

- **XML PATH (montado todo no próprio comando SELECT)**

Neste caso, não há substituição automática dos caracteres < (menor), > (maior), " (aspas) e ' (apóstrofo). Precisaremos criar uma função para fazer esta substituição.

```
CREATE FUNCTION FN_XML_CHAR( @S VARCHAR(1000) )
RETURNS VARCHAR(1000)
AS BEGIN
DECLARE @CONT INT = 1;
DECLARE @RET VARCHAR(1000) = '';
DECLARE @C CHAR(1);
WHILE @CONT <= LEN(@S)
BEGIN
SET @C = SUBSTRING(@S,@CONT,1);
SET @RET += CASE
WHEN @C = '<' THEN '&lt;'
WHEN @C = '>' THEN '&gt;'
WHEN @C = '&' THEN '&amp;'
WHEN @C = '"' THEN '&quot;'
WHEN @C = '''' THEN '&#39;'
ELSE @C
END
SET @CONT += 1;
END
RETURN @RET;
END
GO
```

- **XML PATH**

```
SELECT
CAST('<Codigo>' + CAST(C.CODCLI AS VARCHAR(5)) + '</Codigo>' AS XML) AS
"node()", 
CAST('<Nome>' + DBO.FN_XML_CHAR( C.NOME ) + '</Nome>' AS XML) AS
"node()"
FROM TB_CLIENTE C
FOR XML PATH('Cliente'), ROOT('Clientes')
```

Resultado:

```
<Clientes>
  <Cliente>
    <Codigo>438</Codigo>
    <Nome>(NINO) ANTONIO ROSA FILHO</Nome>
  </Cliente>
  <Cliente>
    <Codigo>372</Codigo>
    <Nome>3R (ARISTEU,ADALTON)</Nome>
  </Cliente>
  <Cliente>
    <Codigo>15</Codigo>
    <Nome>A. LUZ DE ALMEIDA</Nome>
  </Cliente>
  <Cliente>
    <Codigo>27</Codigo>
    <Nome>ABB LIMA FILHO LTDA.AAAA</Nome>
  </Cliente>
  <Cliente>
    <Codigo>28</Codigo>
    <Nome>ABILIO MARTINS PINTO JR-ME</Nome>
  </Cliente>
  <Cliente>
    <Codigo>617</Codigo>
    <Nome>ADALBERTO M. CABRAL</Nome>
  </Cliente>
```

- **XML PATH (Mestre detalhe com atributo no detalhe)**

```
SELECT
  CAST('<Codigo>' + CAST(C.CODCLI AS VARCHAR(5)) + '</Codigo>' AS XML)
AS "node()", 
  CAST('<Nome>' + DBO.FN_XML_CHAR( C.NOME ) + '</Nome>' AS XML) AS
"node()", 
  (
    SELECT
      NUM_PEDIDO AS "@IdPedido", VLR_TOTAL AS "@VlrTotal",
      DATA_EMISSAO AS "@DataEmissao"
    FROM TB_PEDIDO
    WHERE CODCLI = C.CODCLI AND
      DATA_EMISSAO BETWEEN '2007.1.1' AND '2007.1.31'
    ORDER BY NUM_PEDIDO
    FOR XML PATH('Pedidos'), TYPE)
  FROM TB_CLIENTE C
  ORDER BY C.NOME
  FOR XML PATH('Cliente'), ROOT('Clientes')
```

Resultado:

```
<Clientes>
  <Cliente>
    <Codigo>438</Codigo>
    <Nome>(NINO) ANTONIO ROSA FILHO</Nome>
  </Cliente>
  <Cliente>
    <Codigo>372</Codigo>
    <Nome>3R (ARISTEU,ADALTON)</Nome>
  </Cliente>
  <Cliente>
    <Codigo>15</Codigo>
    <Nome>A. LUZ DE ALMEIDA</Nome>
  </Cliente>
  <Cliente>
    <Codigo>27</Codigo>
    <Nome>ABB LIMA FILHO LTDA.AAAA</Nome>
  </Cliente>
  <Cliente>
    <Codigo>28</Codigo>
    <Nome>ABILIO MARTINS PINTO JR-ME</Nome>
  </Cliente>
  <Cliente>
    <Codigo>617</Codigo>
    <Nome>ADALBERTO M. CABRAL</Nome>
  </Cliente>
  <Cliente>
    <Codigo>461</Codigo>
    <Nome>ADEMILSON FRANCISCO COSTA</Nome>
  </Cliente>
```

- **XML PATH (Mestre detalhe com elementos no detalhe)**

```
SELECT
  CAST('<Codigo>' + CAST(C.CODCLI AS VARCHAR(5)) + '</Codigo>' AS XML)
AS "node()", 
  CAST('<Nome>' + DBO.FN_XML_CHAR( C.NOME ) + '</Nome>' AS XML) AS
"node()", 
  ( SELECT
      CAST('<NumPedido>' + CAST( NUM_PEDIDO AS VARCHAR (5)) +
          '</NumPedido>' AS XML)
    AS "node()", 
      CAST('<VlrTotal>' + CAST( VLR_TOTAL AS VARCHAR (15)) +
          '</VlrTotal>' AS XML)
    AS "node()", 
      CAST('<DataEmissao>' + CONVERT(VARCHAR(10),DATA_EMISSAO,
112) +
          '</DataEmissao>' AS
XML) AS "node()"
```

```
FROM TB_PEDIDO
WHERE CODCLI = C.CODCLI AND DATA_EMISSAO BETWEEN '2007.1.1'
AND '2007.1.31'
    ORDER BY NUM_PEDIDO
    FOR XML PATH('Pedido'), TYPE)
FROM TB_CLIENTE C
ORDER BY C.NOME
FOR XML PATH('Cliente'), ROOT('Clientes')
```

Resultado:

```
<Clientes>
<Cliente>
<Codigo>438</Codigo>
<Nome>(NINO) ANTONIO ROSA FILHO</Nome>
</Cliente>
<Cliente>
<Codigo>372</Codigo>
<Nome>3R (ARISTEU,ADALTON)</Nome>
</Cliente>
<Cliente>
<Codigo>15</Codigo>
<Nome>A. LUZ DE ALMEIDA</Nome>
</Cliente>
<Cliente>
<Codigo>27</Codigo>
<Nome>ABB LIMA FILHO LTDA.AAAA</Nome>
</Cliente>
<Cliente>
<Codigo>28</Codigo>
<Nome>ABILIO MARTINS PINTO JR-ME</Nome>
</Cliente>
<Cliente>
<Codigo>617</Codigo>
<Nome>ADALBERTO M. CABRAL</Nome>
</Cliente>
```

1.4.2. Métodos XML

Vejamos, nos subtópicos a seguir, os métodos XML.

1.4.2.1. Query

Este método permite a consulta em uma estrutura XML. Veja os exemplos adiante:

```
-- Declarando uma variável XML  
DECLARE @XML XML  
  
-- Carrega as informações da consulta para a variável XML,  
utilizando o FOR XML:  
SET @XML =  
(  
    SELECT CODFUN, NOME, DATA_ADMISSAO  
    FROM TB_EMPREGADO AS EMPREGADO  
    FOR XML AUTO, ELEMENTS  
)
```

Consultando o resultado do XML:

```
SELECT @XML.query('EMPREGADO')
```

Resultado:

```
|<EMPREGADO>  
|<CODFUN>1</CODFUN>  
|<NOME>OLAVO TRINDADE</NOME>  
|<DATA_ADMISSAO>1986-10-05T00:00:00</DATA_ADMISSAO>  
|</EMPREGADO>  
|<EMPREGADO>  
|<CODFUN>2</CODFUN>  
|<NOME>JOSE REIS</NOME>  
|<DATA_ADMISSAO>1987-05-02T00:00:00</DATA_ADMISSAO>  
|</EMPREGADO>  
|<EMPREGADO>  
|<CODFUN>3</CODFUN>  
|<NOME>MARCELO SOARES</NOME>  
|<DATA_ADMISSAO>1986-10-05T00:00:00</DATA_ADMISSAO>  
|</EMPREGADO>
```

Para execução das consultas, é necessário realizar a declaração da variável e da atribuição da consulta para @XML. Como estas instruções são repetidas, as próximas consultas não serão apresentadas.

Consultando um campo específico:

```
...  
SELECT @XML.query('EMPREGADO/NOME')
```

Resultado:

```
<NOME>OLAVO TRINDADE</NOME>  
<NOME>JOSE REIS</NOME>  
<NOME>MARCELO SOARES</NOME>  
<NOME>PAULO CESAR JUNIOR</NOME>  
<NOME>JOAO LIMA MACHADO DA SILVA</NOME>  
<NOME>CARLOS ALBERTO SILVA</NOME>  
<NOME>ELIANE PEREIRA</NOME>  
<NOME>RUDGE RAMOS SANTANA DA PENHA</NOME>
```

Consultando um registro específico:

```
...  
--Retorna o primeiro registro  
SELECT @XML.query('EMPREGADO[1]/NOME')  
--Retorna o décimo registro  
SELECT @XML.query('EMPREGADO[10]/NOME')
```

Pesquisando um valor de um determinando campo. No exemplo, a pesquisa retornará todos os campos do registro:

```
SELECT @XML.query('EMPREGADO[NOME=''MARCELO SOARES'']')
```

O mesmo exemplo anterior, porém retornando apenas o campo da data de admissão:

```
SELECT @XML.query('EMPREGADO[NOME=''MARCELO SOARES'']/DATA_ADMISSAO')
```

1.4.2.2. Value

Retorna o valor do caminho do XML. Vejamos os exemplos a seguir:

A consulta adiante retorna o campo **NOME** do primeiro registro:

```
DECLARE @XML XML
SET @XML =
    (SELECT CODFUN, NOME, DATA_ADMISSAO FROM TB_EMPREGADO
AS EMPREGADO
FOR XML AUTO, ELEMENTS)

SELECT @XML.value('(EMPREGADO/NOME)[1]', 'varchar(100)')
```

A seguir, a consulta que retorna o campo **ID** do décimo-quinto registro:

```
...
SELECT @XML.value('(EMPREGADO/CODFUN)[15]', 'INT')
```

1.4.2.3. Exists

Exists verifica a existência de um caminho específico. O retorno é 0, quando não existir, e 1, quando existir. Vejamos os exemplos a seguir:

A consulta adiante verifica se existe o campo **CODFUN**:

```
DECLARE @XML XML
SET @XML =
    (SELECT CODFUN, NOME, DATA_ADMISSAO FROM TB_EMPREGADO
AS EMPREGADO
FOR XML AUTO, ELEMENTS)

SELECT @XML.exist('EMPREGADO/CODFUN' )
```

Consulta que verifica se existe o registro 35 e o retorno será 1:

```
...
SELECT @XML.exist('(EMPREGADO/CODFUN)[35]' )
```

Ao alterar o valor para 3500, o retorno será 0, pois não encontrou o registro correspondente:

```
...
SELECT @XML.exist('(EMPREGADO/CODFUN)[3500]' )
```

Junto com o comando **CASE**:

```
...  
SELECT CASE @XML.exist('(EMPREGADO/CODFUN)[3500]')  
WHEN 0 THEN 'Não EXISTE'  
WHEN 1 THEN 'EXISTE' END
```

1.4.2.4. Nodes

O método **Nodes** permite que a expressão XML seja integrada à cláusula **FROM**.

Exemplo:

Retornar o campo **NOME** do XML:

```
DECLARE @XML XML  
SET @XML =  
    (SELECT CODFUN, NOME, DATA ADMISSAO  
     FROM TB_EMPREGADO AS EMPREGADO FOR XML AUTO, ELEMENTS )  
  
SELECT C.query('.') as Nome  
FROM @xml.nodes('EMPREGADO/NOME') AS X(C)
```

1.4.3. Gravando um arquivo XML

O comando **BCP** permite a importação e exportação de arquivos para o SQL Server. O **BCP** é executado no **prompt de comando** ou através da procedure **XP_CMDSHELL**, que executa comandos do sistema operacional.

Para habilitar a execução da procedure **XP_CMDSHELL**, é necessário executar os passos a seguir:

```
--Habilitar opções avançadas  
sp_configure 'show advanced option',1  
go  
reconfigure  
go  
  
--Habilitar a execução da procedure XP_CMDSHELL  
sp_configure 'xp_cmdshell',1  
go  
reconfigure
```

Vejamos, a seguir, os parâmetros do comando **BCP**:

- **Queryout**: Nome do arquivo de saída;
- **S**: Nome do servidor;
- **T**: Acesso através de conta do Windows;
- **w**: Utiliza UNICODE;
- **r**: Terminador de linha;
- **t**: Terminador de campo {TAB}.

```
DECLARE @CMD VARCHAR(4000)

SET @CMD =
'BCP "SELECT * FROM PEDIDOS.DBO.TB_TIPOPRODUTO AS TIPO FOR XML AUTO,
ROOT(''RESULTADO''), ELEMENTS '' +
' QUERYOUT "C:\DADOS\ARQUIVOXML.XML" -SINSTRUTOR -t -w -t -T'

EXEC MASTER..XP_CMDSHELL @CMD
```

Após a execução do comando, o arquivo será gerado no local indicado.

1.4.4. Abrindo um arquivo XML

A consulta de um arquivo XML pode ser realizada através do **OPENROWSET**. Veja o exemplo adiante:

Crie a tabela **TB_TIPO_XML** para carregar as informações do arquivo XML gerado no exemplo anterior:

```
CREATE TABLE TB_TIPO_XML
(
    COD_TIPO INT,
    TIPO      VARCHAR(30)
)
GO
```

Utilizando o **OPENROWSET**, realize a consulta e inserção na tabela:

```
INSERT INTO TB_TIPO_XML
SELECT
    X.TIPO.query('COD_TIPO').value('.','INT'),
    X.TIPO.query('TIPO').value('.','VARCHAR(30)')
FROM
    (
```

```
SELECT CAST(X AS XML)
FROM OPENROWSET(
    BULK 'C:\DADOS\ARQUIVOXML.XML',
    SINGLE_BLOB) AS T(X)
) AS T(X)
CROSS APPLY X.nodes('RESULTADO/TIPO') AS X(TIPO);
```

Consulte a tabela para verificar se o arquivo foi carregado com sucesso:

```
SELECT * FROM TB_TIPO_XML
```

	COD TIPO	TIPO
1	0	NÃO CADASTRADO
2	1	ABRIDOR
3	2	PORTA LAPIS
4	3	REGUA
5	4	ACES.CHEVEIRO
6	5	CANETA
7	6	CHAVEIRO
8	7	BOTTÓN
9	8	MISTURADOR DE DRINKS
10	9	PORTA MOEDAS
11	10	CARTAO PVC
12	15	YO-YO
13	100	MAQUINAS
14	101	CARGAS P/ CANETA

1.5. JSON

JSON (JavaScript Object Notation) É uma notação para a troca de informações baseada em Javascript. O suporte para JSON foi adicionado na versão 2016 do SQL Server.

1.5.1. FOR JSON

Vejamos, adiante, a execução de uma consulta simples com saída JSON:

```
SELECT ID_PRODUTO, COD_TIPO, DESCRICAO, PRECO_VENDA
FROM TB_PRODUTO
FOR JSON AUTO
```

	Results	Messages
1	JSON_F52E2B61-18A1-11d1-B105-00805F49916B	
2	[{"ID_PRODUTO":1,"COD_TIPO":1,"DESCRICAO":"ABRIDOR SACA & ROLHA","PRECO_VENDA":0.1418}, {"ID_PRODUTO":2,"COD_TIPO":2,"DESCRICAO":"PORTA L..."]	
3	CANETA STILL","PRECO_VENDA":0.5474}, {"ID_PRODUTO":26,"COD_TIPO":5,"DESCRICAO":"CANETA VERSATIL C/CORDAO","PRECO_VENDA":3.9415}, {"ID_PRO...	
	O_VENDA":5.8073}, {"ID_PRODUTO":51,"COD_TIPO":103,"DESCRICAO":"LIXA DE UNHA","PRECO_VENDA":10.1795}, {"ID_PRODUTO":52,"COD_TIPO":103,"DESCRI...	

A seguir, uma consulta com saída para JSON e ROOT de nome **Empregados**:

```
SELECT Empregado.CODFUN AS Código, Empregado.NOME, Empregado.DATA_
ADMISSAO, Empregado.SALARIO
FROM TB_EMPREGADO Empregado
FOR JSON AUTO, ROOT('Empregados')
```

The screenshot shows the SQL Server Management Studio interface with the 'Results' tab selected. The output is a JSON array named 'Empregados' containing five objects, each representing an employee with fields: Código, NOME, DATA_ADMISSAO, and SALARIO.

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	{"Empregados": [{"Código": 1, "NOME": "OLAVO TRINDADE", "DATA_ADMISSAO": "1986-10-05T00:00:00", "SALARIO": 3000.00}, {"Código": 2, "NOME": "JOAQUIM JUNIOR FILHO", "DATA_ADMISSAO": "1984-02-20T00:00:00", "SALARIO": 500.00}, {"Código": 24, "NOME": "ITAMAR FIGUEIREDO", "DATA_ADMISSAO": "2001-09-11T00:00:00", "SALARIO": 800.00}, {"Código": 50, "NOME": "AUGUSTO SILVEIRA DA SILVA", "DATA_ADMISSAO": "2016-06-10T10:46:50.633", "SALARIO": 1000.00}, {"Código": 1076, "NOME": "TESTE INCLUSÃO", "DATA_ADMISSAO": "2016-06-10T10:46:50.633", "SALARIO": 1000.00}]}]

Vejamos um exemplo com vários campos e ROOT **Clients**:

```
SELECT
    Cliente.CODCLI AS IdCliente, Cliente.NOME AS Cliente,
    Pedidos.NUM_PEDIDO AS IdPedido,
    Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS Emissao,
    Itens.NUM_ITEM AS IdItem, Itens.ID_PRODUTO AS IdProduto,
    Itens.QUANTIDADE AS Quantidade, Itens.PR_UNITARIO AS PrUnitario
FROM TB_CLIENTE Cliente
JOIN TB_PEDIDO Pedidos ON Cliente.CODCLI = Pedidos.CODCLI
JOIN TB_ITENSPEDIDO Itens ON Pedidos.NUM_PEDIDO = Itens.NUM_PEDIDO
WHERE Pedidos.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
-- Importante para ter um resultado correto
ORDER BY Cliente.NOME, Pedidos.NUM_PEDIDO
FOR JSON AUTO, ROOT('Clients')
```

Neste exemplo, será utilizado o **JSON PATH** com ROOT:

```
SELECT
    Cliente.CODCLI AS IdCliente, Cliente.NOME AS Cliente,
    Pedidos.NUM_PEDIDO AS IdPedido,
    Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS Emissao
FROM TB_CLIENTE Cliente JOIN TB_PEDIDO Pedidos ON Cliente.CODCLI =
Pedidos.CODCLI
WHERE Pedidos.DATA_EMISSAO BETWEEN '2014.1.1' AND '2014.1.31'
FOR JSON PATH, ROOT('Clients')
```

1.5.2. OPENJSON

OPENJSON é uma função que possibilita a leitura de um conjunto de dados JSON.

Vejamos, adiante, a execução de uma consulta simples com OPENJSON:

```
SELECT * FROM OPENJSON('["São Paulo", "Rio de Janeiro", "Minas Gerais", "Paraná", "Santa Catarina"]')
```

	key	value	type
1	0	São Paulo	1
2	1	Rio de Janeiro	1
3	2	Minas Gerais	1
4	3	Paraná	1
5	4	Santa Catarina	1

Neste exemplo, será criada uma variável JSON e carregado um valor para ser lido com a função JSON:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{ "CODCLI": 1, "NOME": "IMPACTA Treinamento"}'

SELECT * FROM OPENJSON(@json) AS CLIENTE;
```

	key	value	type
	CODCLI	1	2
	NOME	IMPACTA Treinamento	1

Informando os campos KEY e Value:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{ "CODCLI": 1, "NOME": "IMPACTA Treinamento"}'

SELECT [KEY], Value FROM OPENJSON(@json) AS CLIENTE;
```

	KEY	Value
1	CODCLI	1
2	NOME	IMPACTA Treinamento

Apresentando somente o campo **Nome**:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{ "CODCLI": 1, "NOME": "IMPACTA Treinamento"}'

SELECT [KEY], Value FROM OPENJSON(@json) AS CLIENTE
WHERE [KEY] = 'NOME'
```

	Results	Messages
	KEY	Value
1	NOME	IMPACTA Treinamento

1.5.3. JSON_VALUE

JSON_VALUE permite extrair um valor diretamente do caminho especificado. No exemplo adiante, serão apresentados os campos **NOME** e **CODCLI**:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{ "CODCLI": 1, "NOME": "IMPACTA Treinamento"}'

-- Campo Nome
SELECT JSON_VALUE(@json, '$.NOME') AS NOME

-- Campo CODCLI
SELECT JSON_VALUE(@json, '$.CODCLI')
```

	Results	Messages
1	NOME	IMPACTA Treinamento
1	(No column name)	1

No exemplo a seguir, são utilizados colchetes para selecionar qual linha será apresentada:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'
{"CLIENTES": [
 {"CODCLI":1,"NOME":"IMPACTA Treinamento"},
 {"CODCLI":2,"NOME":"FACULDADE IMPACTA"}
 ]}'

SELECT JSON_VALUE(@json, '$.CLIENTES[0].NOME') AS NOME

SELECT JSON_VALUE(@json, '$.CLIENTES[1].NOME') AS NOME
```

No exemplo adiante, será utilizada uma matriz e restaurado o segundo valor:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{ "CODCLI": 1, "NOME": "IMPACTA Treinamento", "FONE": ["(11)3342-1234", "(11)3342-1235"] }'

SELECT JSON_VALUE(@json, '$.FONE[1]') AS FONE
```

Results	
	Messages
1	(11)3342-1235

1.5.4. JSON_QUERY

JSON_QUERY retorna uma parte ou matriz de um JSON.

A consulta adiante retorna as informações da matriz FONE:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{ "CODCLI": 1, "NOME": "IMPACTA Treinamento", "FONE": ["(11)3342-1234", "(11)3342-1235"] }'

SELECT JSON_QUERY(@json, '$.FONE') AS FONE
```

FONE
[{"FONE": "(11)3342-1234"}, {"FONE": "(11)3342-1235"}]

No exemplo a seguir, são apresentadas todas as informações do JSON:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{"CLIENTES": [
    {"CODCLI":1,"NOME":"IMPACTA Treinamento"},
    {"CODCLI":2,"NOME":"FACULDADE IMPACTA"}
]}'
SELECT JSON_QUERY(@json, '$') a
```

a
{"CLIENTES": [{"CODCLI":1,"NOME":"IMPACTA Treina..."]}

1.5.5. ISJSON

ISJSON valida a entrada do JSON. Retorna **1** quando é válido e **0** quando inválido.

Neste exemplo, será validada uma variável do tipo JSON:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{"CLIENTES": [
    {"CODCLI":1,"NOME":"IMPACTA Treinamento"},
    {"CODCLI":2,"NOME":"FACULDADE IMPACTA"}
]}'
SELECT CASE ISJSON(@json)
        WHEN 1 THEN 'Variável JSON válida'
        ELSE 'Variável JSON inválida'
    END as Validadcao
```

Como os parâmetros estão OK, o retorno será **1** e acrescentando-se o **CASE**:

Validacao
Variável JSON válida

No exemplo adiante, será retirado : (dois pontos) depois da palavra **CLIENTES**:

```
DECLARE @json NVARCHAR(4000)

SET @json = N'{"CLIENTES": [
    {"CODCLI":1,"NOME":"IMPACTA Treinamento"}, 
    {"CODCLI":2,"NOME":"FACULDADE IMPACTA"}]
}'

SELECT CASE ISJSON(@json)
    WHEN 1 THEN 'Variável JSON válida'
    ELSE 'Variável JSON inválida'
END as Validacao
```

O resultado será um tipo JSON inválido:

Results	
	Messages
1	Variável JSON inválida

1.5.6. Exportação para arquivo JSON

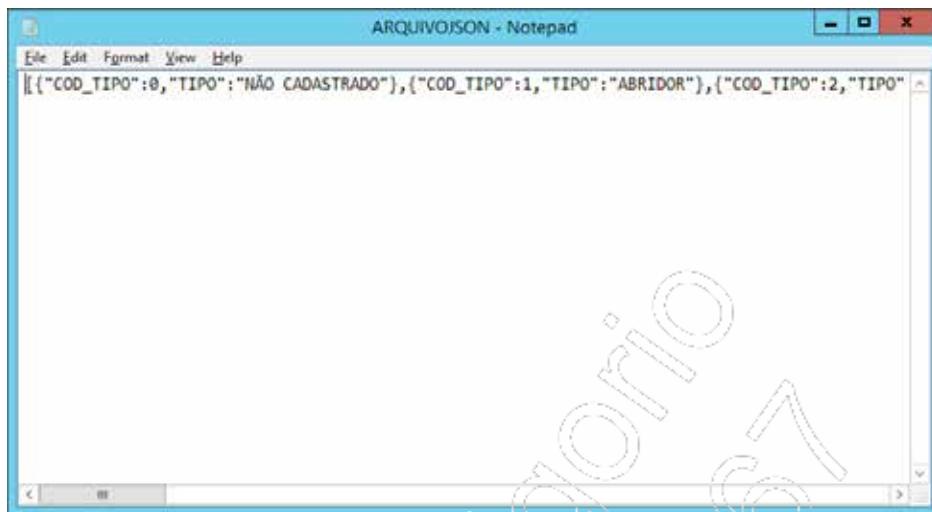
Um método para exportar um arquivo JSON é a utilização do BCP. No exemplo a seguir, será exportado o resultado da consulta no banco **Pedidos**, tabela **TB_TIPOPRODUTO**:

```
DECLARE @CMD VARCHAR(4000)

SET @CMD =
'BCP "SELECT * FROM PEDIDOS.DBO.TB_TIPOPRODUTO AS TIPO FOR JSON AUTO" ' +
' QUERYOUT "C:\DADOS\ARQUIVOJSON.XML" -SINSTRUTOR -t -w -t -T'

EXEC MASTER..XP_CMDSHELL @CMD
```

Results	
	Messages
1	output
2	NULL
3	Starting copy...
4	NULL
5	1 rows copied.
6	Network packet size (bytes): 4096
7	Clock Time (ms.) Total : 15 Average : (6...
8	NULL



1.5.7. Importação de arquivo JSON

Para importarmos um arquivo JSON, utilizaremos o **BULK INSERT**.

Crie a tabela **TB_TIPO_JSON** para carregar as informações do arquivo JSON gerado no exemplo anterior:

```
CREATE TABLE TB_TIPO_JSON
(
    COD_TIPO INT,
    TIPO      VARCHAR(30)
)
GO
```

Para realizar a consulta no arquivo:

```
SELECT RESULTADO.*
FROM OPENROWSET (BULK 'C:\DADOS\ARQUIVOJSON.JSON', SINGLE_NCLOB) as j
CROSS APPLY OPENJSON(BulkColumn)
WITH( COD_TIPO INT, TIPO nvarchar(30)) AS RESULTADO
```

Utilizando o recurso de **INSERT** com um **SELECT**:

```
INSERT INTO TB_TIPO_JSON
SELECT RESULTADO.*
FROM OPENROWSET (BULK 'C:\DADOS\ARQUIVOJSON.JSON', SINGLE_NCLOB) as
j
CROSS APPLY OPENJSON(BulkColumn)
WITH( COD_TIPO INT, TIPO nvarchar(30)) AS RESULTADO
```

Messages
(17 row(s) affected)

Para validar, realize uma consulta na tabela:

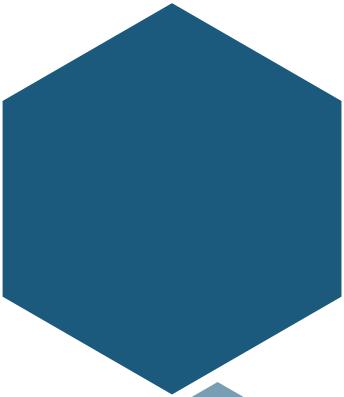
```
SELECT * FROM TB_TIPO_JSON
```

COD_TIPO	TIPO
0	NÃO CADASTRADO
1	ABRIDOR
2	PÓRTA LÁPIS
3	REGUA
4	ACES.CHAVEIRO
5	CANETA
6	CHAVEIRO
7	BOTTON
8	MISTURADOR DE DRINKS
9	PORTA MOEDAS
10	CARTAO PVC
15	YO-YO

Pontos principais

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

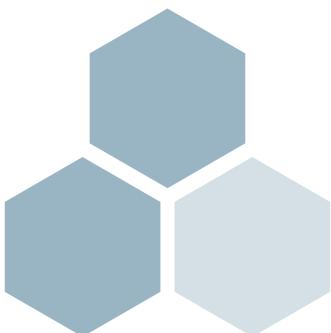
- Todas as informações de conexão necessárias para o acesso a um banco de dados local ou remoto, a partir de uma fonte OLE DB, estão incluídas em **OPENROWSET**;
- O SQL Server suporta transações com arquivos e dados XML;
- Para exportar uma consulta, utilize FOR XML;
- Os métodos que o SQL utiliza para acessar tipos XML são: **Query**, **Value**, **Exists** e **Nodes**;
- O SQL 2016 permite acesso para tipo de dados JSON;
- As funções JSON são: **OPENJSON**, **JSON_VALUE**, **JSON_QUERY** e **ISJSON**.



Acesso a recursos externos

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 13 a 15.



1. Sobre acesso a recursos externos, qual alternativa está incorreta?

- a) É necessário habilitar a opção Ad Hoc Distributed Queries.
- b) Não podemos acessar recursos externos.
- c) O comando OPENROWSET realiza consultas em outras bases.
- d) No comando OPENROWSET, são configuradas as informações de acesso.
- e) Podemos acessar várias outras fontes de dados como: Excel, Access, outras bases SQL etc.

2. Qual afirmação está errada com relação à exportação de consultas para XML?

- a) Podemos exportar para XML utilizando o FOR XML.
- b) É um meio simples de exportação para XML.
- c) É configurável.
- d) Não é um meio confiável.
- e) Existem várias opções de exportação do comando FOR XML.

3. Qual a função do BULK INSERT?

- a) Função de inserção de dados.
- b) Inserção de arquivos de texto.
- c) Inserção de planilha do Excel.
- d) Comando de carga XML.
- e) Comando de carga JSON.

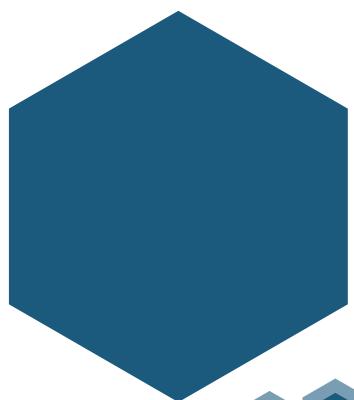
4. O que o comando adiante realiza?

```
SELECT ID_PRODUTO, COD_TIPO, DESCRICAO, PRECO_VENDA  
FROM TB_PRODUTO  
FOR XML RAW
```

- a) Gera uma consulta no formato RAW.
- b) Após a execução, o SQL cria um arquivo no formato XML.
- c) Gera um erro de sintaxe, pois deveria ter AUTO no final do comando.
- d) Gera uma consulta da tabela TB_PRODUTO com saída no formato XML.
- e) Este comando está desatualizado e deve ser utilizado o FOR JSON.

5. Qual função está errada com relação ao JSON?

- a) CLOSEJSON
- b) OPENJSON
- c) JSON_VALUE
- d) JSON_QUERY
- e) ISJSON



Acesso a recursos externos



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 13 a 15.



Laboratório 1

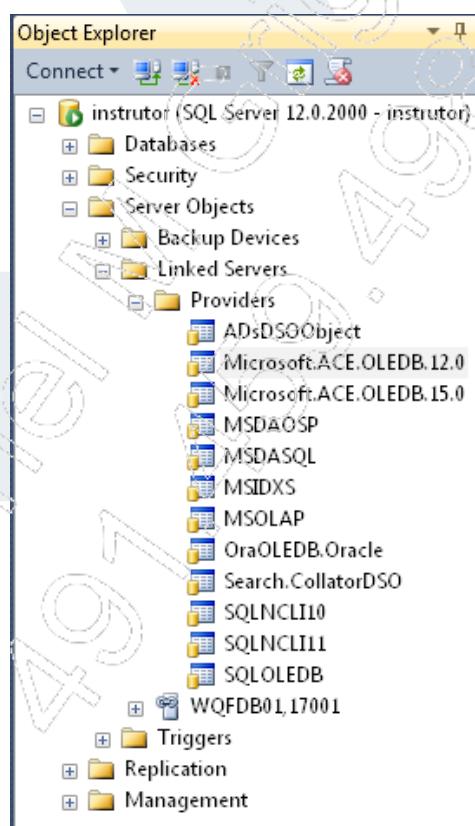
A – Configurando o SQL Server

1. Execute os comandos adiante para habilitar opções avançadas e consultas distribuídas:

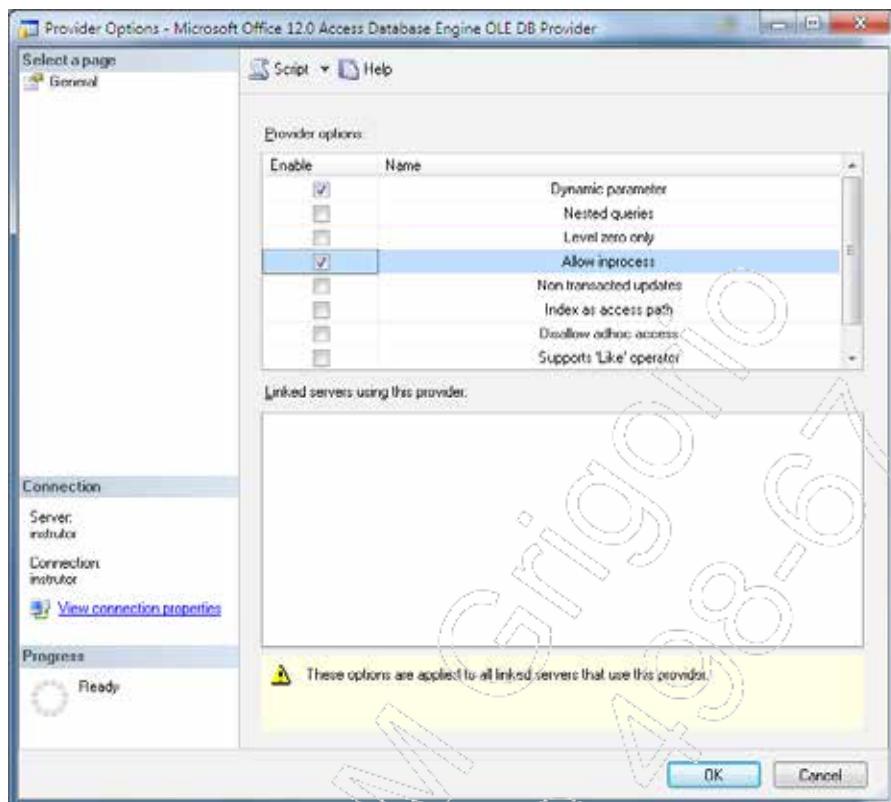
```
EXEC sp_configure 'show advanced option', '1';
Reconfigure

exec sp_configure 'Ad Hoc Distributed Queries',1
reconfigure
```

2. Acesse o servidor e, em **Server Objects**, expanda **Linked Servers** e **Providers**:



3. Com o botão direito do mouse, selecione as propriedades:



4. Marque as opções **Dynamic parameter** e **Allow inprocess**.

B – Consultas distribuídas

1. Faça uma consulta na tabela **Produtos** do banco Access (**Pedidos.accdb**), que está na pasta **Cap05**;
2. Utilizando o **OPENROWSET**, realize uma consulta na tabela **CLIENTES**;
3. Ainda utilizando o **OPENROWSET**, realize uma consulta na tabela **CLIENTES** do banco **Pedidos.accdb** e relacione com a tabela **PEDIDOS** do banco **PEDIDOS**. Apresente as seguintes informações: **Num_pedido**, **Nome**, **VLR_TOTAL**, e **DATA_EMISAO** dos pedidos de janeiro de 2014.

C – Trabalhando com BULK INSERT

1. Crie a tabela **TESTE_BULK_INSERT**;

```
CREATE TABLE TESTE_BULK_INSERT
( CODIGO      INT,
  NOME        VARCHAR(40),
  DATA_NASCIMENTO DATETIME )
```

2. Através do comando **BULK INSERT**, faça a carga na tabela **TESTE_BULK_INSERT** com o arquivo **BULK_INSERT.txt**;
3. Faça uma consulta na tabela **TESTE_BULK_INSERT** e verifique se as informações foram carregadas.

Laboratório 2

A – Trabalhando com XML

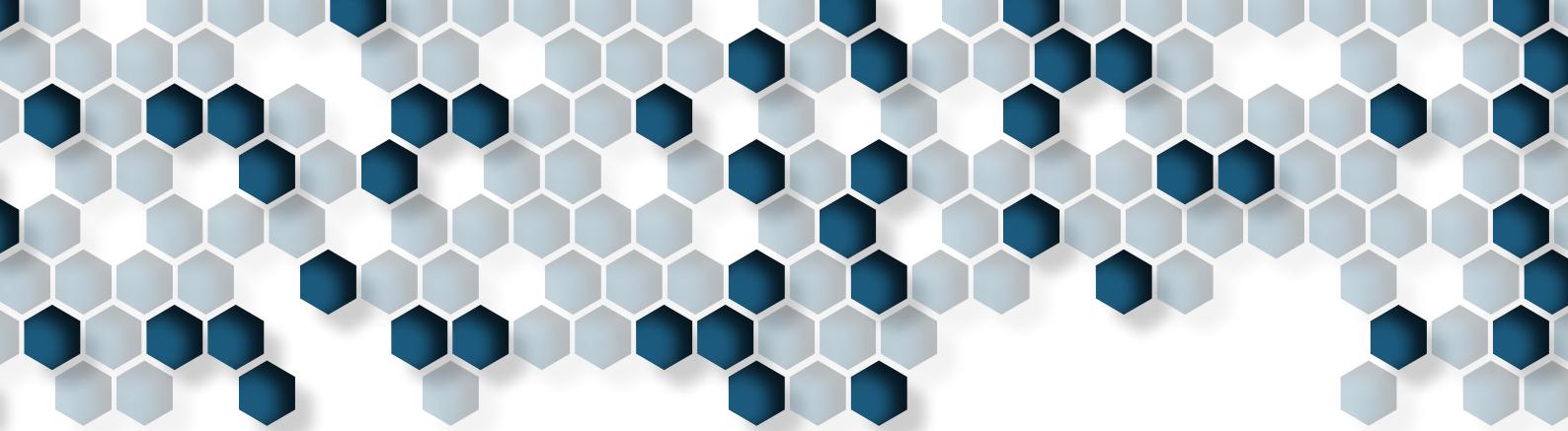
1. Coloque o banco de dados **PEDIDOS** em uso;
2. Realize uma consulta, apresentando as seguintes informações: número de pedido, nome do cliente, nome do vendedor, data de emissão e valor total dos pedidos de janeiro de 2014, ordenado pelo número de pedido;
3. Execute a consulta anterior, exportando para XML conforme o modelo a seguir:

```
<row NUM_PEDIDO="2964" CLIENTE="VIACAO LIMEIRENSE LTDA" VENDEDOR="CELSO MARTINS" DATA_EMISSAO="2007-01-01T00:00:00" VLR_TOTAL="1735.43" />
<row NUM_PEDIDO="2965" CLIENTE="PERSONAL INDUSTRIA COMERCIO E EXPORTACAO LTDA." VENDEDOR="MARCELO" DATA_EMISSAO="2007-01-01T00:00:00" VLR_TOTAL="1482.60" />
```

Laboratório 3

A – Trabalhando com JSON

1. Faça uma consulta apresentando o código e nome dos clientes;
2. Utilizando a consulta anterior, execute uma saída com JSON;
3. Gere um arquivo no padrão JSON para a consulta do item 1.



Views

- ◆ Utilizando views;
- ◆ CREATE VIEW;
- ◆ ALTER VIEW;
- ◆ DROP VIEW;
- ◆ Visualização de informações sobre views;
- ◆ Views atualizáveis;
- ◆ Retorno de dados tabulares.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 16 a 18.



1.1. Introdução

Uma **view** é uma tabela virtual formada por linhas e colunas de dados, os quais são provenientes de tabelas referenciadas em uma query que define a view. Suas linhas e colunas são geradas de forma dinâmica no instante em que se referencia a view.

A query que determina uma view pode ser proveniente de uma ou mais tabelas ou, ainda, de outras views. Para determinar uma view que utiliza dados provenientes de fontes distintas, podemos utilizar as queries distribuídas.

É permitida a realização de quaisquer queries e até mesmo a alteração de dados por meio de views.

1.2. Utilizando views

Ao criarmos uma view, podemos selecionar o conteúdo que desejamos exibir de uma tabela, visto que views podem funcionar como filtros que auxiliam no agrupamento de dados das tabelas, protegendo certas colunas e simplificando o código de uma programação.

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

As views podem ser de três tipos. A escolha entre um deles depende da finalidade para a qual as criaremos. A seguir, temos a descrição dos três tipos de views:

- **Views standard:** Nesse tipo de view são reunidos, em uma tabela virtual, dados provenientes de uma ou mais views ou tabelas base;
- **Views indexadas:** Esse tipo de view é obtido por meio da criação de um índice clusterizado sobre a view;
- **Views particionadas:** Esse tipo de view permite que os dados em uma grande tabela sejam divididos em tabelas menores. Os dados são particionados entre as tabelas de acordo com os valores aceitos nas colunas.

1.2.1. Vantagens oferecidas pelas views

A utilização de views apresenta as seguintes vantagens:

- **Reutilização:** As views são objetos de caráter permanente, portanto, elas podem ser lidas por vários usuários de forma simultânea;
- **Redução do custo de execução:** Os resultados já computados que ficam armazenados em uma view indexada são empregados pelo otimizador de query e, assim, é reduzido o custo de execução;
- **Segurança:** As views permitem ocultar determinadas colunas de uma tabela;
- **Compatibilidade:** Views são capazes de criar uma interface compatível com versões anteriores, simulando uma tabela que teve seu esquema modificado;
- **Cópia de dados:** As views podem ser muito úteis para a melhoria de desempenho e para a partição de dados nos casos em que copiamos dados para o SQL Server ou a partir dele;
- **Simplificação do código:** As views permitem criar um código de programação muito mais limpo, na medida em que podem conter um **SELECT** complexo. Dessa forma, podemos criar essas views para os programadores a fim de poupar-lhos do trabalho de escrever várias consultas **SELECT** complexas.

1.2.2. Restrições

Antes de criarmos views, devemos levar em consideração as restrições que serão citadas a seguir:

- A instrução **SELECT** de uma view não pode:
 - Gerar duas colunas com o mesmo nome;
 - Utilizar a cláusula **ORDER BY**, a não ser que seja incluída a cláusula **TOP (n)**;
 - Utilizar a palavra-chave **INTO**;
 - Fazer referência a uma tabela temporária ou variável;
 - Utilizar variáveis.

- Apenas poderemos usar **SELECT *** em uma definição de view se a cláusula **SCHEMABINDING** não for especificada;
- As views podem possuir, no máximo, 1024 colunas;
- As views podem ser aninhadas em até 32 níveis;
- A instrução **CREATE VIEW** deve ser a única instrução em um batch.

1.2.3. 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;
- **texttype**: 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
```

1.2.4. Views de catálogo

As **views de catálogo** representam a interface mais geral para os metadados de catálogos, sendo que até mesmo os metadados disponíveis para usuários são exibidos por meio dessas views. As views de catálogo permitem não apenas obter mas também alterar e apresentar informações personalizadas.

O SQL Server Database Engine utiliza as informações que são retornadas por essas views, dentre as quais não estão incluídos dados referentes ao backup, à replicação, ao Database Maintenance Plan ou dados referentes ao catálogo SQL Server Agent. Há views de catálogo que utilizam linhas herdadas de outras views de catálogo.

1.3. CREATE VIEW

A sintaxe utilizada para a criação de uma view é a seguinte:

```
CREATE VIEW nome_view [ (lista_de_colunas) ]
[ WITH [ENCRYPTION][,SCHEMABINDING] ]
AS instrucao_select
[ WITH CHECK OPTION ]
```

Em que:

- **nome_view**: Nome da view;
- **lista_de_colunas**: Nomes das colunas da view;
- **WITH ENCRYPTION**: Protege o código fonte da view, impedindo que ele seja aberto a partir do **Object Explorer**;

- **WITH SCHEMABINDING:** Cria uma view ligada às estruturas das tabelas às quais ela faz referência. As tabelas que participam da view não poderão ter suas estruturas alteradas enquanto a view não for alterada de forma compatível;
- **instrucao_select:** Comando **SELECT** que será gravado na view;
- **WITH CHECK OPTION:** Impede a inclusão e a alteração de dados através da view que sejam incompatíveis com a cláusula **WHERE** da instrução **SELECT**.

Por meio de uma view, também é possível incluir dados em uma tabela. Para isso, é preciso que ocorra uma das seguintes situações: ou as colunas da tabela base que não são exibidas na view devem aceitar valores nulos, ou devem ser autoincrementais, ou devem ter um valor padrão (default) definido para elas.

Caso a view contenha um **SELECT** que realiza a leitura dos dados presentes em diferentes tabelas, para inserir dados nessas tabelas por meio da view, é preciso inseri-los nas colunas que atinjam uma tabela por vez. Quando realizamos a inclusão, a alteração ou a exclusão dos dados de uma tabela base da view, a tarefa realizada reflete na view de forma automática.

O exemplo a seguir demonstra a utilização de **CREATE VIEW**:

```
USE PEDIDOS
-- Criando a VIEW
CREATE VIEW VIE_EMP1 AS
SELECT CODFUN, NOME, DATA_ADMISSAO,
       COD_DEPTO, COD_CARGO, SALARIO
FROM TB_EMPREGADO
-- Testando a VIEW
SELECT * FROM VIE_EMP1
--
SELECT CODFUN, NOME FROM VIE_EMP1
```

1.3.1. Utilizando WITH ENCRYPTION

A cláusula **WITH ENCRYPTION** permite realizar a criptografia das entradas da tabela **syscomments** que contenham o texto do comando **CREATE VIEW**. Ao utilizarmos esta cláusula, a view não pode ser publicada como parte da replicação do SQL Server. Quando as views, os triggers ou as stored procedures são criptografados, o usuário deixa de ter acesso ao código que as gerou. Depois de criptografada, uma view não pode mais ser lida pelo usuário porque alguns caracteres ilegíveis são inseridos no lugar de sua codificação.

Vejamos, a seguir, como utilizar a cláusula **WITH ENCRYPTION**:

```
-- Utilizando ENCRYPTION
CREATE VIEW VIE_EMP2_A WITH ENCRYPTION
AS
SELECT TOP 100 CODFUN, NOME, DATA ADMISSAO,
       COD_DEPTO, COD_CARGO, SALARIO
FROM TB_EMPREGADO
ORDER BY NOME
-- Procurar esta view no Object Explorer e
-- dar um clique com o botão direito sobre ela
```

! Podemos observar no **Object Explorer** que não é possível ter acesso ao código fonte da view.

1.3.2. Utilizando **WITH SCHEMABINDING**

Quando especificamos **SCHEMABINDING**, a view fica ligada ao esquema (estrutura) da(s) tabela(s) à(s) qual(is) faz referência, então, não podemos fazer modificações na(s) tabela(s) se isto implicar em alterações na definição da view. Para que a(s) tabela(s) possa(m) ser alterada(s), é necessário que as dependências existentes entre a view e a(s) tabela(s) sejam retiradas, portanto, primeiro a definição da view deve ser modificada ou removida.

Ao tentarmos modificar ou remover tabelas ou views que possam causar alterações na definição da view criada com **SCHEMABINDING**, o Database Engine gera um erro.

Também, no instante em que afetarem a definição da view, as instruções **ALTER TABLE** executadas em tabelas que participam de views com **SCHEMABINDING** falharão.

! Não será possível especificar **SCHEMABINDING** nos casos em que a view tiver colunas de tipos de dados criados pelos usuários. Para criação de índices é obrigatória a utilização desta cláusula.

O uso de **SCHEMABINDING** requer que a instrução **SELECT** contenha o nome do SCHEMA dos objetos: tabelas, views ou funções. É importante lembrar que os objetos referenciados devem estar, obrigatoriamente, no mesmo banco de dados.

O exemplo a seguir demonstra a utilização de **SCHEMABINDING**:

```
CREATE VIEW VIE_EMP3
WITH ENCRYPTION, SCHEMABINDING
AS
SELECT CODFUN, NOME, DATA_ADMISSAO,
       COD_DEPTO, COD_CARGO, SALARIO, NUM_DEPEND
FROM DBO.TB_EMPREGADO
GO
-- Testando a VIEW
SELECT * FROM VIE_EMP3
-- Não é possível apagar a coluna da tabela
GO
ALTER TABLE TB_EMPREGADO DROP COLUMN NUM_DEPEND
GO
-- Criando índices para a view
CREATE UNIQUE CLUSTERED INDEX IX_VIE_EMP3_CODFUN
ON VIE_EMP3(CODFUN)
GO
--
CREATE INDEX IX_VIE_EMP3_NOME
ON VIE_EMP3(NOME)
```

1.3.3. Utilizando WITH CHECK OPTION

Esta cláusula faz com que os critérios definidos na cláusula **WHERE** sejam seguidos no momento em que são executados os comandos que realizam a alteração de dados com relação às views. Dessa forma, a cláusula **WITH CHECK OPTION** assegura que os dados continuem visíveis por meio da view, mesmo após uma linha ter sido alterada.

! Caso a cláusula **TOP seja definida em qualquer lugar na instrução **SELECT**, a cláusula **WITH ENCRYPTION** não poderá ser especificada.**

Por meio da view, podemos impedir a inclusão de dados que não estejam de acordo com a cláusula **WHERE**. Para isso, basta acrescentar a cláusula **WITH CHECK OPTION** no final do código de criação da view, como mostrado no exemplo a seguir:

```
-- VIEW com condição de filtro
CREATE VIEW VIE_EMP4 WITH ENCRYPTION
AS
SELECT CODFUN, NOME, DATA_ADMISSAO,
       COD_DEPTO, COD_CARGO, SALARIO
FROM TB_EMPREGADO
WHERE COD_DEPTO = 2
GO
-- Testando
SELECT * FROM VIE_EMP4
```

A view apresentada mostrará apenas funcionários que trabalham no departamento de código 2, como podemos visualizar adiante:

	CODFUN	NOME	DATA_ADMISSAO	COD_DEPTO	COD_CARGO	SALARIO
1	2	JOSE REIS	1987-05-02 00:00:00.000	2	14	600.00
2	9	RUDGE RAMOS SANTANA DA PENHA	1985-12-23 00:00:00.000	2	4	800.00
3	19	SEBASTIÃO SILVA	1988-04-06 00:00:00.000	2	1	8300.00
4	20	EURICO BRANDÃO	1988-01-09 00:00:00.000	2	4	800.00
5	25	MARIA DA PENHA	1983-07-15 00:00:00.000	2	11	4500.00
6	28	MARIANO DE OLIVEIRA	1993-04-03 00:00:00.000	2	9	3330.00
7	38	LUIS FERNANDO LEMOS	2005-01-01 00:00:00.000	2	14	600.00
8	40	JOAQUIM ALBERTO	2003-04-05 00:00:00.000	2	5	500.00

No SQL Server, podemos alterar uma tabela através de uma view, mediante algumas condições vistas a seguir:

```
INSERT INTO VIE_EMP4 ( NOME, DATA_ADMISSAO, COD_DEPTO,
                      COD_CARGO, SALARIO)
VALUES ('TESTE INCLUSÃO', GETDATE(), 1, 1, 1000)
```

No entanto, devemos observar que o comando **INSERT** anterior insere um funcionário no departamento código 1 e, portanto, a view nunca exibirá este funcionário, porque ela mostra somente funcionários do departamento 2.

O ideal, então, é que esta view não permita a inclusão de registros que não satisfazam a condição de filtro. Isso é feito com a instrução **WITH CHECK OPTION**:

```
ALTER VIEW VIE_EMP4 WITH ENCRYPTION
AS
SELECT CODFUN, NOME, DATA_ADMISSAO,
       COD_DEPTO, COD_CARGO, SALARIO
FROM TB_EMPREGADO
WHERE COD_DEPTO = 2
WITH CHECK OPTION
```

1.4. ALTER VIEW

Mesmo depois de criadas, as views permitem que façamos alterações em seus nomes ou em suas definições. A sintaxe utilizada para alterar uma view é a seguinte:

```
ALTER VIEW nome_view [ (lista_de_colunas ) ]
[ WITH [ENCRYPTION][,SCHEMABINDING] ]
AS instrucao_select
[ WITH CHECK OPTION ]
```

Em que:

- **nome_view**: Nome da view;
- **lista_de_colunas**: Nomes das colunas da view;
- **WITH ENCRYPTION**: Protege o código fonte da view, impedindo que ele seja aberto a partir do **Object Explorer**;
- **WITH SCHEMABINDING**: Cria uma view ligada às estruturas das tabelas às quais ela faz referência. As tabelas que participam da view não poderão ter suas estruturas alteradas enquanto a view não for alterada de forma compatível;
- **instrucao_select**: Comando **SELECT** que será gravado na view;
- **WITH CHECK OPTION**: Impede a inclusão e a alteração de dados através da view que sejam incompatíveis com a cláusula **WHERE** da instrução **SELECT**.

1.5. DROP VIEW

A sintaxe utilizada para remover uma view de um banco de dados é a seguinte:

```
DROP VIEW nome_view [ ,n ] [ ; ]
```

Em que:

- **nome_view**: É o nome da view a ser excluída.

A exclusão de uma view implica a exclusão de todas as permissões que tenham sido dadas sobre ela. Dessa forma, devemos utilizar o comando **DROP VIEW** apenas nas situações em que desejamos de fato retirar esse objeto do sistema. Caso contrário, podemos utilizar o comando **ALTER VIEW** para alterar o código de criação de uma view.

1.6. Visualizando informações sobre views

As informações sobre as views podem ser vistas por meio do SQL Management Studio, tanto pelo **Object Explorer** (onde encontramos a lista de views do banco de dados e temos acesso a colunas, triggers, índices e estatísticas definidas na view) como pela caixa de diálogo **View Properties** (onde encontramos as propriedades individuais das views).

Também, podemos consultar as informações sobre as views por meio de:

- **sys.view**: Exibe a lista de views do banco de dados;
- **sp_helptext**: Mostra as definições de views não criptografadas;
- **sys.sql_dependencies**: Exibe todos os objetos que possuem alguma dependência de outro objeto, como é o caso das views.

Em muitos casos, pode ser útil consultar informações sobre views para sabermos o modo como seus dados foram derivados das tabelas originais ou quais são os dados definidos pela view.

Nas situações em que pretendemos mudar o nome de um objeto, por exemplo, e esse objeto é referenciado por uma view, será necessário modificá-la para que seu texto utilize o novo nome do objeto. Dessa forma, é importante primeiro consultar as dependências desse objeto para sabermos se as views serão atingidas por tal mudança.

Vale lembrar, ainda, que:

- Views não criptografadas fornecerão mais informações sobre sua definição;
- A forma como as consultas em views são feitas é igual à das tabelas ordinárias.

1.7. Views atualizáveis

As views podem ser utilizadas com a finalidade de alterar dados nas tabelas. Porém, as alterações são feitas com algumas restrições. Vejamos quais são:

- As colunas a serem alteradas com instruções **UPDATE**, **INSERT** ou **DELETE**, por exemplo, devem pertencer a uma mesma tabela base;
- As colunas alteradas em uma view devem referenciar diretamente os dados originais nas colunas da tabela;
- Para poderem ser modificadas, as colunas não podem ser computadas ou derivadas pelo uso de funções agregadas, como **AVG**, **COUNT**, **SUM**, **MIN**, **MAX**, **GROUPING**, **STDEV**, **STDEVP**, **VAR** e **VARP**;
- Para poderem ser modificadas, colunas que tiverem sido computadas por uma expressão ou pelos operadores **UNION**, **UNION ALL**, **CROSSJOIN**, **EXCEPT** e **INTERSECT** devem ser especificadas com um trigger **INSTEAD OF**;
- As colunas a serem alteradas não serão afetadas pelo uso das cláusulas **GROUP BY**, **HAVING** ou **DISTINCT**;

- É necessário, por meio da instrução **INSERT**, especificar valores para todas as colunas da tabela original que não permitam valores nulos e que não tenham definições **DEFAULT**;
- Ao incluirmos **WITH CHECK OPTION** na definição da view, torna-se obrigatória a adesão aos critérios do **SELECT** por parte de todas as instruções de modificação de dados executados na view;
- **TOP** não pode ser utilizado com **WITH CHECK OPTION** no **SELECT** a ser gravado na view.

1.8. Retornando dados tabulares

Podemos utilizar o termo **dado tabular** para definir qualquer recurso que retorne linhas e colunas. Pois bem, se quisermos deixar gravado no nosso banco de dados um procedimento que nos retorne um dado tabular, dispomos de três opções: **view**, **procedure** e **função tabular**.

No exemplo a seguir, temos o uso de uma view para retornar dados tabulares:

```
--- Devolvendo dado tabular com VIEW
-----[REDACTED]-----
CREATE VIEW VIE_MAIOR_PEDIDO AS
    SELECT TOP 12 MONTH( DATA_EMISSAO ) AS MES,
           YEAR( DATA_EMISSAO ) AS ANO,
           MAX( VLR_TOTAL ) AS MAIOR_PEDIDO
    FROM TB_PEDIDO
    -- NÃO ACEITA PARÂMETRO. Trabalha somente com constantes
    WHERE YEAR(DATA_EMISSAO) = 2014
    GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
    ORDER BY MES

--- Consulta com SELECT
SELECT * FROM VIE_MAIOR_PEDIDO
-- Ordenando consulta com ORDER BY
SELECT * FROM VIE_MAIOR_PEDIDO
ORDER BY MAIOR_PEDIDO DESC
-- Filtrando consulta com WHERE
SELECT * FROM VIE_MAIOR_PEDIDO
WHERE MES > 6
-- Exemplo da instrução JOIN em uma VIEW
SELECT V.*, P.NUM_PEDIDO, C.NOME AS CLIENTE
FROM VIE_MAIOR_PEDIDO V JOIN TB_PEDIDO P
    ON V.MES = MONTH( P.DATA_EMISSAO ) AND
       V.ANO = YEAR( P.DATA_EMISSAO ) AND
       V.MAIOR_PEDIDO = P.VLR_TOTAL
JOIN TB_CLIENTE C ON P.CODCLI = C.CODCLI
```

Uma view não pode receber parâmetros e não utiliza variáveis e comandos de programação, portanto sempre retornará todos os registros da consulta. Para filtrar o ano de 2014, será necessário incluir a cláusula **WHERE**.

```
-- Consulta com SELECT
SELECT * FROM VIE_MAIOR_PEDIDO WHERE ANO=2014;
```

Agora vejamos este outro exemplo, que utiliza uma stored procedure para retornar dados tabulares:

```
-----  
--- Devolvendo dado tabular com STORED PROCEDURE  
-----  
  
CREATE PROCEDURE STP_MAIOR_PEDIDO @ANO INT  
AS BEGIN  
    SELECT MONTH( DATA_EMISSAO ) AS MES,  
          YEAR( DATA_EMISSAO ) AS ANO,  
          MAX( VLR_TOTAL ) AS MAIOR_PEDIDO  
    FROM TB_PEDIDO  
    -- Aceita parâmetro. Trabalha com dados variáveis  
    WHERE YEAR(DATA_EMISSAO) = @ANO  
    GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)  
    ORDER BY MES  
END  
-----  
-- Executa com EXEC  
-- Não aceita ORDER BY  
-- Não aceita WHERE  
-- Não aceita JOIN etc...  
EXEC STP_MAIOR_PEDIDO 2012  
EXEC STP_MAIOR_PEDIDO 2013  
EXEC STP_MAIOR_PEDIDO 2014  
GO
```

Com o uso de uma stored procedure, resolvemos o problema do parâmetro, pois a mesma procedure poderá retornar os dados de qualquer ano. No entanto, uma stored procedure é executada com **EXEC nomeProcedure listaDeParametros**. Não é possível utilizar **WHERE**, **ORDER BY**, **JOIN**, entre outros.

Já no exemplo seguinte, que utiliza função tabular, reunimos as vantagens de **VIEW** (executada com **SELECT**) e de **PROCEDURE** (recebe parâmetro) para retornar dados tabulares:

```
-- Devolvendo dado tabular com FUNÇÃO TABULAR

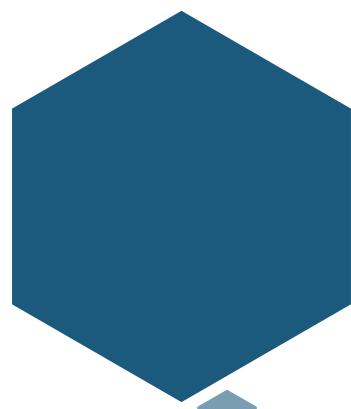
CREATE FUNCTION FN_MAIOR_PEDIDO( @DT1 DATETIME,
                                    @DT2 DATETIME)
RETURNS TABLE
AS
RETURN ( SELECT MONTH( DATA_EMISSAO ) AS MES,
              YEAR( DATA_EMISSAO ) AS ANO,
              MAX( VLR_TOTAL ) AS MAIOR_VALOR
        FROM TB_PEDIDO
        -- Aceita parâmetros. Trabalha com variáveis
        WHERE DATA_EMISSAO BETWEEN @DT1 AND @DT2
        GROUP BY MONTH( DATA_EMISSAO ),
                  YEAR( DATA_EMISSAO ) )
GO

-- Executa com SELECT
SELECT * FROM DBO.FN_MAIOR_PEDIDO( '2014.1.1','2014.12.31')
ORDER BY ANO, MES
-- Aceita filtro
SELECT * FROM DBO.FN_MAIOR_PEDIDO( '2014.1.1','2014.12.31')
WHERE MES > 6
ORDER BY ANO, MES
-- Aceita JOIN
SELECT F.MES, F.ANO, F.MAIOR_VALOR, P.NUM_PEDIDO
FROM FN_MAIOR_PEDIDO( '2014.1.1','2014.12.31') F
JOIN TB_PEDIDO P
ON F.MES = MONTH( P.DATA_EMISSAO ) AND
F.ANO = YEAR( P.DATA_EMISSAO ) AND
F.MAIOR_VALOR = P.VLR_TOTAL
ORDER BY F.ANO, F.MES
```

Pontos principais

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

- Uma **view** é uma tabela virtual formada por linhas e colunas de dados provenientes de tabelas referenciadas em uma query;
- Para criar uma view, utilizamos o comando **CREATE VIEW**;
- Após criadas, as views podem sofrer alterações em seus nomes ou em suas definições, por meio do comando **ALTER VIEW**;
- Para excluir uma view, utilizamos o comando **DROP VIEW**. A exclusão de uma view implica na exclusão de todas as permissões que tenham sido dadas sobre ela;
- As informações sobre as views podem ser vistas por meio do SQL Management Studio, através do **Object Explorer** ou da caixa de diálogo **View Properties**;
- Para ter uma view indexada, é necessário criar um índice clusterizado único para ela. Assim, o conjunto de resultados é armazenado no banco de dados e, por consequência, o desempenho apresenta melhora;
- Podemos utilizar views com a finalidade de alterar dados nas tabelas. As alterações, porém, têm algumas restrições.



Views

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 16 a 18.



Editora
IMPACTA



1. Com relação às características da utilização de uma view, qual alternativa está incorreta?

- a) Segurança.
- b) Simplificação do código.
- c) Reutilização.
- d) Compatibilidade.
- e) Mais lenta que consultas normais.

2. Qual cláusula não é compatível com a criação de uma view?

- a) WITH CHECK OPTION
- b) WITH NOLOCK
- c) WITH SCHEMABINDING
- d) WITH ENCRYPTION
- e) WITH SCHEMABINDING , ENCRYPTION

3. O que ocorre ao utilizarmos a cláusula WITH ENCRYPTION?

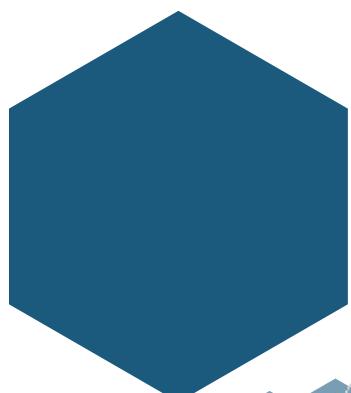
- a) Todos os registros da view são encriptados.
- b) Somente as colunas com valores não nulos são encriptadas.
- c) O código é encriptado, mas visível para o usuário OWNER.
- d) O código é encriptado.
- e) O código é encriptado, mas visível pelo Object Explorer.

4. O que não é correto afirmar sobre a cláusula WITH SCHEMABINDING?

- a) É uma opção que limita as alterações da tabela e não deve ser utilizada.
- b) É necessário informar o schema ao informar a tabela ou view.
- c) Vincula os objetos de referência à view, não permitindo alterações no schema enquanto a view existir.
- d) É obrigatória para a criação de índices para a view.
- e) Utilize para garantir confiabilidade nos objetos VIEWS.

5. Para que uma view seja atualizada, qual afirmação está incorreta?

- a) Não é possível atualizar registros a partir de uma view.
- b) As colunas devem pertencer a uma mesma tabela.
- c) As colunas não podem ser computadas.
- d) Colunas que não são especificadas na view devem permitir valores nulos ou possuir valores DEFAULT.
- e) Colunas não serão alteradas se forem utilizadas as cláusulas GROUP BY, HAVING ou DISTINCT.

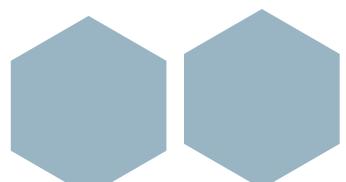


Views

Mãos à obra!

Gabriel M. Grigorio
497.459.498-67

Este laboratório refere-se ao conteúdo das Aulas 16 a 18.



Laboratório 1

A – Criando views e consultando as tabelas virtuais criadas pela view

1. Coloque em uso o banco de dados **PEDIDOS**;
2. Crie uma view (**VIE_TOT_VENDIDO**) para mostrar o total vendido (soma de **TB_PEDIDO.VLR_TOTAL**) em cada mês do ano. Devem ser mostrados o mês, o ano e o total vendido;
3. Faça uma consulta **VIE_TOTAL_VENDIDO** no ano de 2014. Os dados devem ser ordenados por mês;
4. Crie uma view (**VIE_MAIOR_PEDIDO**) para mostrar o valor do maior pedido (**MAX** de **TB_PEDIDO.VLR_TOTAL**) vendido em cada mês do ano. Devem ser mostrados o mês, o ano e o maior pedido;
5. Faça uma consulta **VIE_MAIOR_PEDIDO** no ano de 2014. Os dados devem ser ordenados por mês;
6. Faça um **JOIN**, utilizando **VIE_MAIOR_PEDIDO** e **PEDIDOS**, que mostre, também, o número do pedido (**TB_PEDIDO.NUM_PEDIDO**) de maior valor em cada mês. Deve-se filtrar o ano de 2014 e ordenar por mês;
7. Realize o mesmo procedimento descrito no passo anterior, desta vez mostrando também o nome do cliente que comprou esse pedido;
8. Crie uma view (**VIE_ITENS_PEDIDO**) que mostre todos os campos da tabela **TB_ITENS_PEDIDO**, mais a data de emissão do pedido (**DATA_EMISSAO**), a descrição do produto (**DESCRICÃO**), o nome do cliente que comprou (**NOME**) e o nome do vendedor que vendeu (**NOME**);
9. Execute **VIE_ITENS_PEDIDO**, filtrando apenas pedidos de janeiro de 2014;
10. Crie a tabela **tb_CLIENTE_VIEW** com os seguintes campos:

ID	Inteiro, autonumerável e PRIMARY KEY
Nome	Alfanumérico de 50
Estado	Alfanumérico de 2

11. Crie uma view de nome **vW_Clientes_VIEW** para consulta e atualização da tabela **tb_CLIENTE_VIEW**;
12. Faça a inserção de dois registros através da view **vW_Clientes_VIEW**;
13. Realize a consulta através da view **vW_Clientes_VIEW**.



Introdução à programação

- Variáveis;
- Operadores;
- Controle de fluxo;
- WHILE;
- Outros comandos.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 19 a 21.



1.1. Introdução

Esta leitura aborda conceitos importantes não apenas para a programação com a linguagem SQL, mas para a programação de modo geral.

Estudaremos a criação de variáveis, o uso de operadores aritméticos, relacionais e lógicos e o uso de elementos de controle de fluxo.

1.2. Variáveis

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

A seguir, temos um exemplo de declaração de variáveis:

```
DECLARE @A INT = 10;
DECLARE @B INT = 20;
PRINT @A + @B
```

Essa declaração também pode ser feita da seguinte forma:

```
DECLARE @A INT = 10, @B INT = 20;
PRINT @A + @B
```

Podemos notar a existência do caractere @ antes da variável. Devemos utilizá-lo no momento da declaração de variáveis. Ainda, é possível perceber que cada uma das variáveis possui um tipo de dados atribuído.

Após a declaração da variável, é possível que um comando ajuste a variável para um valor no batch, valor este que poderá ser obtido a partir da variável por outro comando no batch.

1.2.1. Atribuindo valores às variáveis

É possível atribuir valores para cada uma das variáveis. Para tal, utilizamos o comando **DECLARE** ou **SET**, como mostra o exemplo a seguir:

```
-- Atribuição com SET
DECLARE @A INT = 10, @B INT = 20, @C INT;
SET @C = @A + @B;
PRINT @C;
```

Podemos, também, fazer a mesma declaração da seguinte forma:

```
DECLARE @A INT, @B INT, @C INT;
SET @A = 10;
SET @B = 20;
SET @C = @A + @B;
PRINT @C;
```

Também é possível realizar a operação a partir da instrução **SELECT**:

```
DECLARE @A INT
SELECT @A = 3
SELECT @A AS Valor
```

1.3. Operadores

Um **operador** é definido como um símbolo que especifica uma ação realizada em uma ou várias expressões. Há diversos tipos de operadores que podemos utilizar no SQL Server 2016, os quais são divididos em categorias. A seguir, veremos os operadores aritméticos, relacionais e lógicos.

1.3.1. Operadores aritméticos

Os **operadores aritméticos** realizam operações matemáticas em duas expressões de quaisquer tipos de dados numéricos. Vejamos:

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Retorna o resto de uma divisão

1.3.2. Operadores relacionais

Os operadores dessa categoria são utilizados para comparar duas expressões. A tabela a seguir descreve os operadores relacionais:

Operador	Descrição
=	Igual a
>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual
<>	Diferente de
!=	Diferente de
!<	Não menor que
!>	Não maior que

1.3.3. Operadores lógicos

Operadores dessa categoria são utilizados para verificar se uma condição é ou não verdadeira, e retornam um tipo de dado booleano com o valor **TRUE** ou **FALSE**. A tabela a seguir descreve os operadores lógicos:

Operador	Descrição
ALL	Retorna TRUE se todo um conjunto de comparações for TRUE .
AND	Retorna TRUE se duas expressões forem TRUE .
ANY	Retorna TRUE se qualquer comparação de um conjunto de comparações for TRUE .
BETWEEN	Retorna TRUE se o operando estiver dentro de uma determinada faixa de valores.
EXISTS	Retorna TRUE se uma subquery possuir quaisquer linhas.
IN	Retorna TRUE se um operando for igual a um dentro de uma lista de expressões.
LIKE	Retorna TRUE se um operando atender a uma condição.
NOT	Reverte o valor de qualquer outro operador booleano.
OR	Retorna TRUE se uma das expressões booleanas for TRUE .
SOME	Retorna TRUE se alguma comparação de um conjunto de comparações for TRUE .

1.3.4. Precedência

A precedência de operadores determina a sequência em que as operações são realizadas em uma expressão que envolve diversos tipos de operadores. Essa sequência é determinante para o resultado retornado. A tabela a seguir demonstra a ordem de precedência adotada pelos operadores utilizados no SQL Server 2016:

Ordem de execução	Operadores
1	\sim (Bitwise NOT)
2	$*$, $/$, $\%$
3	$+$ (Positivo), $-$ (Negativo), $+$ (Adição), $(+)$ (Concatenação), $-$ (Subtração), $\&$ (Bitwise AND), \wedge (Bitwise exclusivo OR), \mid (Bitwise OR)
4	$=$, $>$, $<$, $>=$, $<=$, \neq , $!=$, $\!>$, $\!<$
5	NOT
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	$=$ (operador de atribuição)

É possível que dois operadores com a mesma ordem de precedência sejam utilizados em uma mesma expressão. Nesse caso, a avaliação dos operadores ocorre da esquerda para a direita, de acordo com a posição deles dentro da expressão.

Podemos utilizar parênteses para ignorar a ordem de precedência dos operadores, ou seja, o que estiver entre parênteses é primeiramente avaliado. Só depois dessa operação é que o valor obtido poderá ser utilizado pelos operadores posicionados fora dos parênteses. Também, parênteses podem ser aninhados, ou seja, podemos ter parênteses entre parênteses. Nessa situação, a expressão dentro dos parênteses aninhados é avaliada antes das outras.

1.4. Controle de fluxo

O SQL Server 2016 trabalha com elementos que compreendem a chamada linguagem de **controle de fluxo**, cuja função é gerenciar o fluxo de execução de comandos Transact-SQL, blocos de comando, stored procedures e funções definidas pelo usuário.

Se não utilizarmos a linguagem de controle de fluxo nas instruções, os comandos Transact-SQL separados são executados sequencialmente na ordem em que aparecem no código.

A utilização de palavras como **BEGIN**, **END**, **IF**, **ELSE**, **WHILE** e **CASE** permite direcionar o uso da linguagem Transact-SQL para obter uma ação específica. Elas têm a capacidade de ligar e relacionar instruções umas às outras e torná-las dependentes entre si.

1.4.1. BEGIN/END

Os elementos **BEGIN** e **END** são utilizados por instruções de controle de fluxo para delimitar um bloco de instruções, ou seja, iniciar e finalizar, respectivamente, um bloco de comandos, de maneira que este possa ser posteriormente executado. Podemos aninhar blocos definidos por tais elementos.

Caso seja executado um bloco de comandos logo após a realização de um teste de condição, os elementos em questão serão utilizados após um comando **IF** ou **WHILE**.

1.4.2. IF/ELSE

Os elementos **IF** e **ELSE** do SQL Server são utilizados para testar condições quando um comando Transact-SQL é executado.

IF e **ELSE** funcionam similarmente aos comandos de mesmo nome utilizados em outras linguagens de programação para testar condições de execução de comandos.

A sintaxe para a utilização de **IF** e **ELSE** é a seguinte:

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

Observando a sintaxe, podemos perceber que ela possui o argumento **expressao_booleana**. Esta expressão pode retornar **TRUE** ou **FALSE**. Se houver um comando **SELECT** na expressão booleana, será preciso colocá-lo entre parênteses.

Já **comando_sql | bloco_comando** representa um comando Transact-SQL ou agrupamento de comandos definido pela utilização de um bloco de comando.

A criação de um bloco de comando é feita por meio dos elementos de controle de fluxo **BEGIN** e **END**.

Os exemplos a seguir demonstram a utilização de **IF** e **ELSE**:

- **Exemplo 1**

No trecho a seguir, como @A não é maior que @B, as instruções contidas no **IF** não serão executadas e somente o último **PRINT** funcionará:

```
DECLARE @A INT = 10, @B INT = 15;
IF @A > @B
BEGIN
PRINT @A;
PRINT 'É MAIOR QUE';
PRINT @B;
END
PRINT 'CONTINUAÇÃO DO CÓDIGO'
```

- **Exemplo 2**

No trecho a seguir, como @A é maior que @B, as instruções contidas no **IF** serão executadas e também o último **PRINT** funcionará, porque está fora do **IF**:

```
DECLARE @A INT = 15, @B INT = 10;
IF @A > @B
BEGIN
PRINT @A;
PRINT 'É MAIOR QUE';
PRINT @B;
END
PRINT 'CONTINUAÇÃO DO CÓDIGO'
```

- **Exemplo 3**

No trecho seguinte, temos um bloco tanto para o caso verdadeiro quanto para o caso falso:

```
DECLARE @A INT = 15, @B INT = 10;
IF @A > @B
BEGIN
PRINT @A;
PRINT 'É MAIOR QUE';
PRINT @B;
END
ELSE
BEGIN
PRINT @A;
PRINT 'NÃO É MAIOR QUE';
PRINT @B;
END
PRINT 'CONTINUAÇÃO DO CÓDIGO'
```

1.5. WHILE

O comando **WHILE** faz com que um comando ou bloco de comandos SQL seja executado repetidamente, isto é, cria-se um loop do comando ou bloco de comandos, que será executado enquanto a condição especificada for verdadeira.

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

A sintaxe de **WHILE** é a seguinte:

```
WHILE expressao_booleana
    { comando_sql | bloco_comando }
    [ BREAK ]
    { comando_sql | bloco_comando }
    [ CONTINUE ]
    { comando_sql | bloco_comando }
```

Em que:

- **expressao_booleana**

Trata-se de uma expressão que retorna **TRUE** ou **FALSE**. Se essa expressão possuir um comando **SELECT**, este deverá estar entre parênteses.

- **{ comando_sql | bloco_comando }**

Trata-se de qualquer instrução Transact-SQL ou grupo de instruções definido em um bloco de comandos.

- **BREAK**

Interrompe um loop. Quaisquer instruções que vierem após a palavra **END** são executadas.

- **CONTINUE**

Reinicia um loop iniciado por **WHILE** ignorando as instruções que vierem após a palavra **CONTINUE**.

1.5.1. BREAK

O comando **WHILE** pode ser utilizado junto de **BREAK**, que é responsável por interromper um loop. **BREAK**, que normalmente é iniciado por um teste **IF**, também pode ser utilizado para finalizar a execução de um loop em um comando **IF/ELSE**.

1.5.2. CONTINUE

É possível reiniciar a execução de um loop executado por **WHILE**. Para isso, basta utilizar **WHILE** com **CONTINUE**.

Assim como **BREAK**, o comando **CONTINUE** normalmente é iniciado por um teste **IF**. Se houver comandos após **CONTINUE**, eles serão ignorados.

1.5.3. Exemplos

Os exemplos exibidos a seguir apresentam algumas formas de utilização do comando **WHILE**:

```
-- 1. WHILE
-- 1.1. Números inteiros de 0 até 100
DECLARE @CONT INT = 0;
WHILE @CONT <= 100
BEGIN
PRINT @CONT;
SET @CONT += 1; -- OU SET @CONT = @CONT + 1
END
PRINT 'FIM'

-- 1.2. Números inteiros de 100 até 0
DECLARE @CONT INT = 100;
WHILE @CONT >= 0
BEGIN
PRINT @CONT;
SET @CONT -= 1; -- OU SET @CONT = @CONT - 1
END
PRINT 'FIM'

-- 1.3. Tabuadas do 1 ao 10 (Loops encadeados)
DECLARE @T INT = 1, @N INT;
WHILE @T <= 10
BEGIN
PRINT 'TABUADA DO ' + CAST(@T AS VARCHAR(2));
PRINT '';
SET @N = 1;
WHILE @N <= 10
BEGIN
PRINT CAST(@T AS VARCHAR(2)) + ' x ' +
CAST(@N AS VARCHAR(2)) + ' = ' +
CAST(@T*@N AS VARCHAR(3));
SET @N += 1;
END -- WHILE @N
END -- WHILE @T
```

```
SET @T += 1;
PRINT '';
END -- WHILE @T

-- 1.4. Palpites para a mega-sena
DECLARE @DEZENA INT, @CONT INT = 1;
WHILE @CONT <= 6
BEGIN
    SET @DEZENA = 1 + 60 * RAND();
    PRINT @DEZENA;
    SET @CONT += 1;
END
PRINT 'BOA SORTE';

-- 1.5. Palpites para a mega-sena
-- (versão 2 para não repetir a mesma dezena)
DECLARE @DEZENA INT, @CONT INT = 1;
IF OBJECT_ID('TBL_MEGASENA') IS NOT NULL
    DROP TABLE TBL_MEGASENA;

CREATE TABLE TBL_MEGASENA( NUM_DEZENA INT );

WHILE @CONT <= 6
BEGIN
    SET @DEZENA = 1 + 60 * RAND();
    IF EXISTS( SELECT * FROM TBL_MEGASENA
                WHERE NUM_DEZENA = @DEZENA )
        CONTINUE;
    INSERT INTO TBL_MEGASENA VALUES (@DEZENA);
    SET @CONT += 1;
END
SELECT * FROM TBL_MEGASENA ORDER BY NUM_DEZENA;
```

1.6. Outros comandos

Vejamos, nos subtópicos seguintes, outros comandos de controle de fluxo.

1.6.1. GOTO

Este comando pula a execução de um batch (um ou mais comandos enviados simultaneamente de uma aplicação para o SQL Server a fim de serem executados) para uma label. Os comandos existentes entre o comando **GOTO** e a label não são executados.

A seguir, temos um exemplo da utilização de **GOTO**:

```
-- GOTO  
A:  
PRINT 'AGORA ESTOU NO PONTO "A"'  
GOTO C  
B:  
PRINT 'AGORA ESTOU NO PONTO "B"'  
GOTO D  
C:  
PRINT 'AGORA ESTOU NO PONTO "C"'  
GOTO B  
D:  
PRINT 'AGORA ESTOU NO PONTO "D"'  
PRINT 'FIM. QUE BAGUNÇA!'
```

! Como o comando **GOTO** torna o código muito confuso, não é recomendável utilizá-lo.

1.6.2. RETURN

Este comando sai de uma query ou procedure de maneira incondicional, podendo ser utilizado em qualquer ponto para sair de um bloco de comandos, de um batch ou de uma procedure. Sua sintaxe é simples:

```
RETURN [expressão]
```

Na sintaxe anterior, **expressão** é o valor inteiro retornado. O que vier após **RETURN** não será executado.

A seguir, temos um exemplo da utilização de **RETURN**:

```
-- RETURN  
PRINT 'AGORA ESTOU NO PONTO "A"'  
PRINT 'AGORA ESTOU NO PONTO "B"'  
RETURN  
PRINT 'AGORA ESTOU NO PONTO "C"'  
PRINT 'AGORA ESTOU NO PONTO "D"'
```

1.6.3. WAITFOR

Este comando de controle de fluxo bloqueia a execução de uma stored procedure, de um batch ou de uma transação até que:

- Um determinado intervalo de tempo ou tempo especificado seja alcançado;
- Um comando especificado modifique ou retorne pelo menos uma linha.

O exemplo a seguir demonstra a utilização de WAITFOR:

```
-- WAITFOR  
WAITFOR DELAY '00:00:05'  
PRINT 'ESPEREI 5 SEGUNDOS'
```

1.6.4. EXISTS

A cláusula **EXISTS** realiza a validação de um subconjunto de dados. A utilização em programação permite validar a existência ou não de valores de uma consulta.

Neste exemplo, vamos validar se o código do cliente de número 43 existe:

```
IF EXISTS (SELECT * FROM PEDIDOS.DBO.TB_CLIENTE WHERE CODCLI=43)  
BEGIN  
    PRINT 'Código existe'  
END  
ELSE  
BEGIN  
    PRINT 'Código não existe'  
END
```

1.6.5. Atribuição de valor de uma consulta

Para carregar um valor de uma consulta, utilize **SET** e a consulta entre parênteses:

```
DECLARE @SOMA DECIMAL(10,2)  
  
SET @SOMA = (SELECT SUM(VLR_TOTAL) FROM PEDIDOS.DBO.tb_PEDIDO WHERE  
VLR_TOTAL IS NOT NULL)  
  
PRINT @SOMA
```

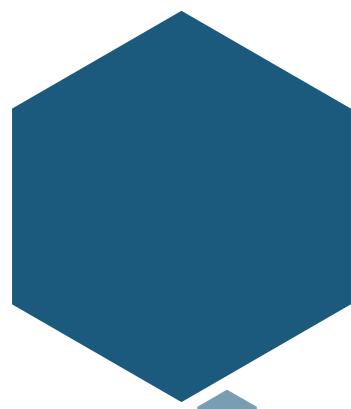
Pontos principais

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

- Uma **variável local** do Transact-SQL é um objeto nos scripts e batches que mantém um valor de dado. Com o comando **DECLARE**, podemos criar variáveis locais, sendo isto feito no corpo de uma procedure ou batch;
- Um **operador** é um símbolo que especifica uma ação realizada em uma ou várias expressões. Entre as diversas categorias de operadores que podemos utilizar no SQL Server 2016, podemos destacar os operadores aritméticos, relacionais e lógicos;
- O **controle de fluxo** tem como função gerenciar o fluxo de execução de comandos Transact-SQL, blocos de comando, stored procedures e funções definidas pelo usuário;
- Os elementos **IF** e **ELSE** do SQL Server são utilizados para testar condições quando um comando Transact-SQL é executado;
- O comando **WHILE** faz com que um comando ou bloco de comandos SQL seja executado repetidamente, enquanto a condição especificada for verdadeira;
- **EXISTS** permite validar uma subconsulta retornando verdadeiro ou falso.

SQL 2016 - Programação em T-SQL (online)

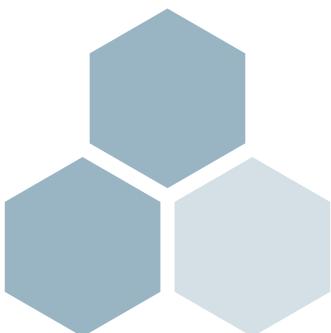
Gabriel M Grigorio
497.459.498-67



Introdução à programação

• Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 19 a 21.



Editora
IMPACTA



1. Verifique o código a seguir:

```
DECLARE @A INT = 35;  
DECLARE @B INT = 20;  
PRINT @A - @B
```

Qual a função da variável neste código?

- a) Armazenar valores.
- b) Realizar a subtração de valores.
- c) Mostrar a soma de @A - @B.
- d) Mostrar a subtração de @A - @B.
- e) Realizar a operação de subtração e apresentação dos valores.

2. Verifique o código a seguir:

```
DECLARE @A INT, @B INT, @C INT;  
SET @A = 40;  
SET @B = 20;  
SET @C = @C + @A + @B;  
PRINT @C;
```

Qual resultado será apresentado?

- a) A soma de 40 + 60.
- b) A soma de 40 + 60, mais o valor da variável @C.
- c) Não retorna valor, pois a variável @C é nula.
- d) O valor de 60.
- e) A sintaxe está errada.

3. Verifique o código a seguir:

```
DECLARE @A INT, @B INT, @C INT = 0;  
SET @A = 10;  
SET @B = 5;  
SET @C = ((@A + @B) / 3) * 2  
SET @C *= 1.2  
PRINT @C;
```

Qual resultado será apresentado?

- a) 9
- b) 10
- c) 11
- d) 12
- e) A sintaxe está errada.

4. Verifique o código a seguir:

```
DECLARE @CONT INT = 0;  
WHILE @CONT <= 100  
    BEGIN  
        BREAK  
        PRINT @CONT;  
        SET @CONT += 1;  
    END  
PRINT 'FIM'
```

Qual resultado será apresentado?

- a) Apresenta os números inteiros de 0 a 100.
- b) Apresenta os números inteiros de 0 a 99.
- c) Apresenta os números inteiros de 0 a 100 e, no final, a palavra "FIM".
- d) A palavra "FIM".
- e) A sintaxe está errada.

5. Verifique o código a seguir:

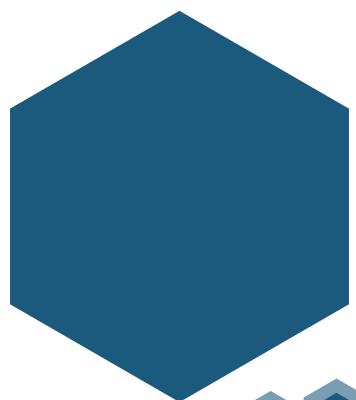
```
DECLARE @CONT INT = 0;  
WHILE @CONT <= 100  
    BEGIN  
        PRINT @CONT;  
        CONTINUE  
        SET @CONT += 1;  
    END  
PRINT 'FIM'
```

Qual resultado será apresentado?

- a) Apresenta os números inteiros de 0 a 100.
- b) Gera um loop infinito.
- c) Apresenta os números inteiros de 0 a 100 e, no final, a palavra "FIM".
- d) A palavra "FIM".
- e) A sintaxe está errada.

SQL 2016 - Programação em T-SQL (online)

Gabriel M Grigorio
497.459.498-67

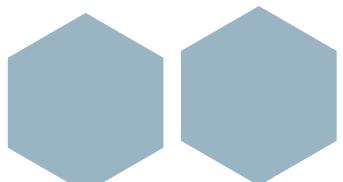


Introdução à programação



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 19 a 21.



Laboratório 1

A – Programando em SQL

1. Complete o código a seguir de modo a mostrar o maior dos três números sorteados:

```
DECLARE @A INT, @B INT, @C INT;
DECLARE @MAIOR INT;
SET @A = 50 * RAND();
SET @B = 50 * RAND();
SET @C = 50 * RAND();
-- Aqui devem ser colocados os IFs ---

-----
PRINT @A;
PRINT @B;
PRINT @C;
PRINT 'MAIOR = ' + CAST(@MAIOR AS VARCHAR(2));
```

2. Complete o código a seguir (que sorteará quatro números no intervalo de 0 a 10, os quais representarão as quatro notas de um aluno) de acordo com o que o comentário pede:

```
DECLARE @N1 NUMERIC(4,2), @N2 NUMERIC(4,2), @N3 NUMERIC(4,2), @N4
NUMERIC(4,2);
DECLARE @MEDIA NUMERIC(4,2);
SET @N1 = 10*RAND();
SET @N2 = 10*RAND();
SET @N3 = 10*RAND();
SET @N4 = 10*RAND();
-- Imprimir as 4 notas que foram sorteadas

-- Calcular e imprimir a média das 4 notas

-- Imprimir REPROVADO se média menor que 5, caso contrário APROVADO

-- Dependendo da média, imprimir uma das classificações adiante
--      Média até 2.....PÉSSIMO
--      Acima de 2 até 4.....RUIM
--      Acima de 4 até 6.....REGULAR
--      Acima de 6 até 8.....BOM
--      Acima de 8.....ÓTIMO
```

Introdução à programação

Aulas 19 a 21

3. Escreva um código que gere e imprima os números pares de 0 até 100;
4. Escreva um código que gere e imprima os números ímpares entre 0 e 100;
5. Complete o código de modo a calcular a soma de todos os números inteiros de 0 até @N;

```
DECLARE @N INT, @CONT INT = 1, @SOMA INT = 0;  
SET @N = CAST( 20 * RAND() AS INT );  
-- Complete o código -----
```

```
--  
PRINT 'A SOMA DE 1 ATÉ ' + CAST(@N AS VARCHAR(2)) +  
' É ' + CAST(@SOMA AS VARCHAR(4));
```

6. Complete o código de modo a calcular o fatorial de @N. Por exemplo, o fatorial de 5 é $1 * 2 * 3 * 4 * 5 = 120$;

```
DECLARE @N INT, @CONT INT = 1, @FAT INT = 1;  
SET @N = CAST( 10 * RAND() AS INT );  
-- Complete o código -----
```

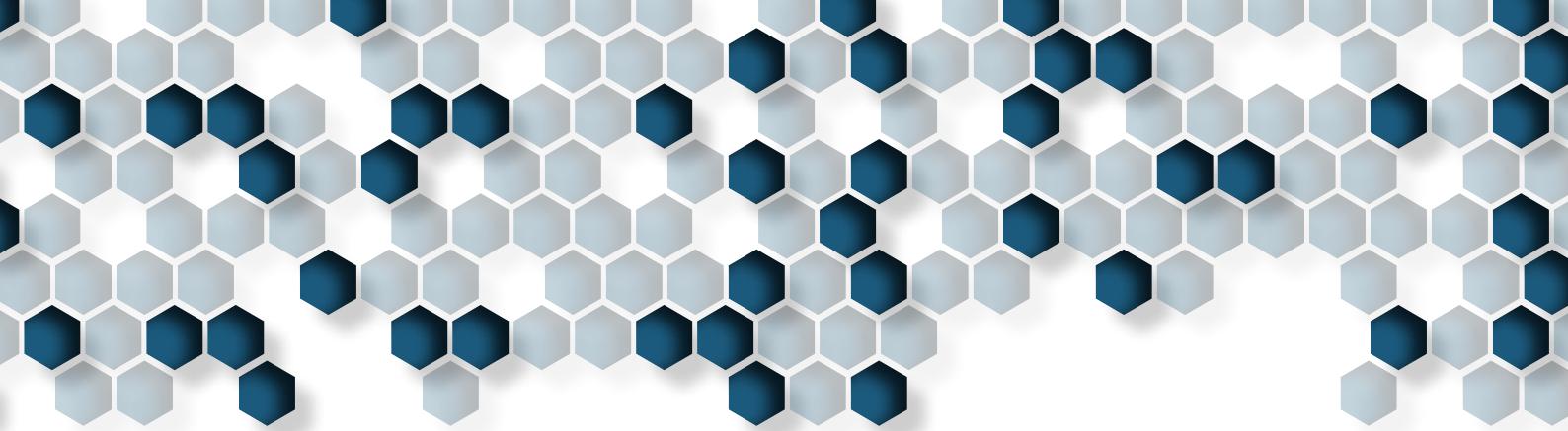
```
--  
PRINT 'O FATORIAL DE ' + CAST(@N AS VARCHAR(2)) +  
' É ' + CAST(@FAT AS VARCHAR(10));
```

7. Insira os comandos de acordo com os comentários adiante, de modo que o código gere todos os números primos de 1 até 1000.

Números primos são números inteiros divisíveis apenas por 1 e por ele próprio.

```
-- 1. Declarar as variáveis @N, @I (inteiros) e  
--     @SN_PRIMO do tipo CHAR(1)  
  
-- 2. Imprimir os números 1, 2 e 3 que já sabemos serem primos  
  
-- 3. Iniciar a variável @N com 4  
  
-- 4. Enquanto @N for menor ou igual a 1000  
--     4.1. Iniciar a variável @I com 2  
--     4.2. Iniciar a variável @SN_PRIMO com 'S'
```

```
-- 4.3. Enquanto @I for menor ou igual a @N / 2  
    -- 4.3.1. Se o resto da divisão de @N por @I  
    -- for zero (é divisível)  
        -- 4.3.1.1. Colocar 'N' na variável @SN_PRIMO  
        -- sinalizando assim que @N não é um número primo  
        -- 4.3.1.2. Abandonar este Loop (4.3)  
    -- 4.3.2. Somar 1 na variável @I  
    -- Final do loop 4.3.  
-- 4.4. Se @SN_PRIMO for 'S', imprimir @N porque ele é primo  
-- 4.5. Somar 1 na variável @N  
-- Final do loop (4)
```

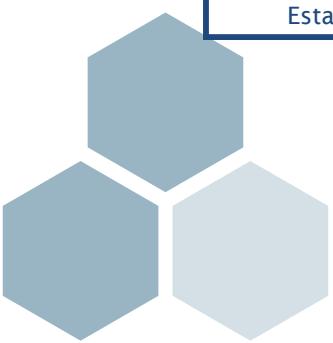


Funções



- Funções e stored procedures;
 - Funções escalares;
 - Funções tabulares;
 - Funções nativas (built-in);
 - Funções de classificação;
 - Funções definidas pelo usuário;
 - Campos computados com funções.
- 

Esta Leitura Complementar refere-se ao conteúdo da Aula 22.



1.1. Introdução

Uma **função (function)** é, basicamente, um objeto que contém um bloco de comandos Transact-SQL responsável por executar um procedimento e retornar um valor ou uma série de valores, os quais podem ser retornados para uma outra função, para um aplicativo, para uma stored procedure ou diretamente para o usuário.

Nesta leitura, abordaremos diversos aspectos relativos às funções do SQL e sua utilização.

1.2. Funções e stored procedures

As funções possuem semelhanças com as stored procedures, já que ambas consistem em uma maneira simplificada de realizar consultas complexas. Ambas aceitam parâmetros que permitem realizar consultas sem conhecimento profundo das estruturas utilizadas para obter metadados. Além disso, ambas podem ter sua lógica compartilhada com um ou mais aplicativos e são executadas no servidor de dados. Contudo, as funções não podem realizar operações que alteram dados no sistema.

Ao utilizar funções, devemos considerar que o retorno obtido de uma função pode ser um valor único escalar ou dados de uma tabela, e que uma função não aceita como parâmetros de entrada dados do tipo **cursor**, **table** ou **timestamp**.

1.3. Funções escalares

As **funções escalares** retornam um valor único, cujo tipo é definido por uma cláusula **RETURNS**. Retornam qualquer valor exceto os tipos: **text**, **ntext**, **image**, **cursor** e **timestamp**. Os parâmetros de entrada podem ser aceitos ou recusados por meio dessas funções, que utilizam sempre uma expressão válida.

Uma função escalar pode ser simples, ou seja, o valor escalar retornado é resultado de uma instrução única. Uma função escalar pode, também, possuir múltiplas instruções que retornarão um valor único. Neste caso, o corpo da função é definido em um bloco **BEGIN...END**, onde estará contida a série de instruções.

A seguir, apresentamos a sintaxe de uma função escalar:

```
CREATE FUNCTION <nome_funcao>
( [@nome_parametro [AS] <tipo_parametro [, ...] ] )
RETURNS <tipo_retorno> [ WITH [ENCRYPTION] [,SCHEMABINDING] ]
[AS]
BEGIN
    <corpo_da_funcao>
    RETURN <expressao_escalar>
END
```

Em que:

- **nome_funcao:** Representa o nome da função definida pelo usuário. Deve incluir o nome do schema ao qual pertence. Após o nome da função, é necessário inserir parênteses, mesmo que nenhum parâmetro seja especificado;
- **@nome_parametro:** Representa um parâmetro. Podem ser declarados um ou mais parâmetros, até o limite de 2100 parâmetros declarados. Se não houver um padrão para o parâmetro, o valor de cada um deve ser fornecido pelo usuário ao executar a função. Parâmetros com nomes iguais podem ser usados em diferentes funções. Não são aceitos parâmetros no lugar de nomes de tabela, coluna ou outros objetos;
- **tipo_parametro:** É o tipo de dados do parâmetro. O schema ao qual ele pertence pode ser opcionalmente especificado. Quando isso não acontece, o mecanismo de banco de dados procura o tipo de dados, primeiramente no schema que contém nomes de tipos de dados do sistema, depois no schema padrão do usuário no banco de dados atual e, por fim, no schema **dbo** no banco de dados atual;
- **tipo_retorno:** É o valor de retorno da função. Os tipos **timestamp**, **cursor** e **table** não são aceitos como retorno;
- **ENCRYPTION:** Define que o texto original da instrução **CREATE FUNCTION** será codificado e transformado em um formato que não seja visível em views de catálogo. Apenas usuários privilegiados têm acesso ao texto original, que não é acessível a usuários sem acesso a tabelas de sistema ou arquivos do banco de dados;
- **SCHEMABINDING:** Associa a função aos objetos de banco de dados que são referenciados por ela. Desse modo, caso outro objeto associado ao schema fizer referência à função, ela não sofrerá alterações. Uma função só pode ser associada a um schema se for uma função Transact- SQL. As views e funções de usuário referenciadas também devem estar associadas ao schema. Objetos referenciados devem pertencer ao mesmo banco de dados que a função e seus nomes devem possuir duas partes, indicando explicitamente o schema ao qual pertencem. Além disso, o usuário que executar **CREATE FUNCTION** deve ter permissão **REFERENCES** nos objetos referenciados. Quando a função for descartada ou modificada com uma instrução **ALTER** sem que **SCHEMABINDING** seja especificado, a associação entre a função e os objetos referenciados por ela será desfeita;
- **corpo_da_funcao:** Representa uma série de instruções que define o valor da função. É importante ressaltar que essas instruções não podem conter comandos do grupo DML, em que há modificação de uma tabela. Em funções escalares, as instruções são avaliadas como valor escalar;
- **expressao_escalar:** Define o valor escalar retornado pela função.

1.3.1. Funções determinísticas e não determinísticas

As funções, tanto nativas (built-in) quanto definidas pelo usuário, estão divididas, com relação ao resultado que retornam, em duas categorias: **determinísticas** e **não determinísticas**.

As **funções determinísticas** sempre retornam o mesmo resultado para um mesmo conjunto de valores de entrada e em um mesmo estado do banco de dados.

```
-- Exemplos:  
-- Retorna a quantidade de caracteres do texto  
SELECT LEN('CARLOS MAGNO')  
  
-- Valor de PI  
SELECT PI()  
  
-- Raiz quadrada de 144  
SELECT SQRT(144)  
  
-- Valor Exponencial de 12  
SELECT SQUARE( 12 )
```

As **funções não determinísticas**, por sua vez, são aquelas que podem retornar resultados distintos para um mesmo conjunto de valores de entrada a cada vez que são chamadas, ainda que o estado do banco de dados permaneça o mesmo. Funções nativas não determinísticas não podem ser executadas por funções definidas pelo usuário.

```
-- Exemplos:  
-- Valor randômico entre 0 e 1  
SELECT RAND()  
-- Data e hora do servidor  
SELECT GETDATE()  
-- Gera um valor exclusivo  
SELECT NEWID()
```

1.4. Funções tabulares

Funções tabulares são aquelas que, utilizando uma cláusula **SELECT**, retornam um conjunto de resultados em forma de tabela. Ou seja, o valor retornado por uma função tabular é do tipo **table**. As funções tabulares podem ser categorizadas como **funções tabulares in-line** ou como **funções tabulares com várias instruções**.

1.4.1. Funções tabulares in-line

Uma **função tabular in-line** não possui corpo de função e pode ser utilizada para conseguir a funcionalidade de views, porém com a utilização de parâmetros.

A seguir, apresentamos a sintaxe de uma função tabular in-line:

```
CREATE FUNCTION <nome_funcao>
( [@nome_parametro [AS] <tipo_parametro [...] ] )
RETURNS TABLE [ WITH [ENCRYPTION] [,SCHEMABINDING]]
[AS]
    RETURN [() <instrucao_select> ()]
```

Em que:

- **nome_funcao**: Representa o nome da função definida pelo usuário;
- **@nome_parametro**: Representa um parâmetro;
- **tipo_parametro**: É o tipo de dados do parâmetro;
- **TABLE**: Define o valor de retorno como uma tabela, que é definido por uma única instrução **SELECT**. Só podem ser passadas constantes e variáveis locais. Não há variáveis de retorno associadas em funções in-line;
- **ENCRYPTION**: Define que o texto original da instrução **CREATE FUNCTION** será codificado e transformado em um formato que não seja visível em views de catálogo;
- **SCHEMABINDING**: Associa a função aos objetos de banco de dados que são referenciados por ela;
- **instrucao_select**: Representa a instrução **SELECT** que define o valor de retorno da função tabular in-line.

1.4.2. Funções tabulares com várias instruções

Uma **função tabular com várias instruções** consiste em uma combinação de view e stored procedure. Como uma stored procedure, ela pode utilizar múltiplas instruções Transact-SQL para construir uma tabela, inserindo as linhas das diversas instruções à tabela retornada. E, da mesma forma que uma view, pode ser usada em uma cláusula **FROM** em uma instrução.

Apresentamos, a seguir, a sintaxe de uma função tabular com várias instruções:

```
CREATE FUNCTION <nome_funcao>
( [@nome_parametro [AS] <tipo_parametro [, ...] ]
RETURNS [@variavel_tabular] TABLE [<estrutura>] [ WITH [ENCRYPTION]
[, SCHEMABINDING]]
[AS]
BEGIN
    <corpo_da_funcao>
    RETURN
END
```

Em que:

- **nome_funcao**: Representa o nome da função definida pelo usuário;
- **@nome_parametro**: Representa um parâmetro;
- **tipo_parametro**: É o tipo de dados do parâmetro;
- **@variavel_tabular**: Representa uma variável TABLE que armazena linhas que deveriam ser retornadas como valor da função;
- **Estrutura**: Define o tipo de dado de tabela para uma função. As definições de coluna e constraints de tabela ou coluna fazem parte da declaração da tabela; Para criar uma função definida pelo usuário, utilizamos o comando **CREATE FUNCTION**. Como uma função nativa, ela é uma rotina que aceita parâmetros, executa uma ação e retorna, como resultado, um valor;
- **ENCRYPTION**: Define que o texto original da instrução **CREATE FUNCTION** será codificado e transformado em um formato que não seja visível em views de catálogo;
- **SCHEMABINDING**: Associa a função aos objetos de banco de dados que são referenciados por ela;
- **corpo_da_funcao**: Representa uma série de instruções que define o valor da função. É importante ressaltar que essas instruções não podem ter um efeito negativo como a modificação de uma tabela. Em funções tabulares com várias instruções, essas instruções preenchem uma variável de retorno TABLE.

1.5. Funções nativas (built-in)

Funções nativas (built-in) são aquelas já fornecidas pelo próprio SQL Server. Aqui, apresentaremos algumas funções built-in do SQL Server, as quais serão utilizadas nos exemplos mais adiante.

1.5.1. Funções de texto

- **ASCII:** Retorna o código ASC de um caractere;

```
--- Código de um dado caractere
SELECT ASCII( 'A' )
SELECT ASCII( 'B' )
SELECT ASCII( '0' )
```

- **CHAR:** Retorna o caractere correspondente a um determinado código;

```
--- Caractere que tem um determinado código
SELECT CHAR(65)
---

DECLARE @COD INT;
SET @COD = 1;
WHILE @COD <= 255
BEGIN
PRINT CAST(@COD AS VARCHAR(3)) + ' - ' + CHAR(@COD);
SET @COD = @COD + 1;
END
```

- **CHARINDEX:** Retorna a posição de uma string dentro de outra;

```
--- Posição de uma string dentro de outra
SELECT CHARINDEX( 'PA', 'IMPACTA' ) -- Retorna 3
SELECT CHARINDEX( 'MAGNO', 'CARLOS MAGNO' ) -- Retorna 8
SELECT CHARINDEX( 'MANO', 'CARLOS MAGNO' ) -- Retorna 0 (zero)
```

- **LEN:** Retorna a quantidade de caracteres de uma string;

```
--- Tamanho de uma string
SELECT LEN( 'IMPACTA' )
SELECT LEN( 'IMPACTA TECNOLOGIA' )
```

- **LEFT:** Retorna N caracteres a partir da esquerda de uma string;

```
--- Pegar uma parte de uma string
SELECT LEFT( 'IMPACTA TECNOLOGIA', 5 ) -- Retorna IMPAC
```

- **RIGHT:** Retorna N caracteres a partir da direita de uma string;

```
SELECT RIGHT( 'IMPACTA TECNOLOGIA', 5 ) -- Retorna LOGIA
```

- **SUBSTRING:** Retorna N caracteres a partir de determinada posição de uma string;

```
-- Retornar 6 caracteres a partir da quinta (5) posição
SELECT SUBSTRING( 'IMPACTA TECNOLOGIA', 5, 6 ) -- Retorna CTA TE
```

- **RTRIM**: Elimina espaços à direita de uma string;

```
--- Eliminar espaços em branco à direita
SELECT 'IMPACTA' + 'TECNOLOGIA'
SELECT RTRIM('IMPACTA') + 'TECNOLOGIA'
```

- **LTRIM**: Elimina espaços à esquerda de uma string;

```
--- Eliminar espaços em branco à esquerda
SELECT 'IMPACTA' + 'TECNOLOGIA'
SELECT 'IMPACTA' + LTRIM('TECNOLOGIA')
```

- **REVERSE**: Retorna o reverso de uma string;

```
--- Reverso de uma string
SELECT REVERSE('IMPACTA')
SELECT REVERSE('MAGNO')
SELECT REVERSE('TECNOLOGIA')
SELECT REVERSE('COMPUTADOR')
SELECT REVERSE('REVER')
SELECT REVERSE('ANILINA')
SELECT REVERSE('MIRIM')
SELECT REVERSE('RADAR')
SELECT REVERSE('REVIVER')
```

- **REPLICATE**: Repete uma string N vezes;

```
--- Replicar string várias vezes
PRINT REPLICATE('IMPACTA ', 10)
```

- **UPPER**: Converte para maiúsculo;

```
--- Maiúsculo
PRINT UPPER('impacta 123 @@ !!')
```

- **LOWER**: Converte para minúsculo;

```
--- Minúsculo
PRINT LOWER( 'tECnOLOGIA 123 @@ !!' )
```

- **REPLACE**: Realiza busca e substituição.

```
-- Substituição de texto
PRINT REPLACE( 'JOSÉ,CARLOS,ROBERTO,FERNANDO,MARCO', ',', ' - ')
```

1.5.2. Funções de data e hora

- **GETDATE**: Retorna data e hora do servidor;

```
-- Data e hora do servidor  
SELECT GETDATE() AS Data
```

- **EOMONTH**: Retorna o último dia do mês de uma data;

```
-- Último dia do mês corrente  
SELECT EOMONTH(GETDATE(), 0)
```

- **DATEFROMPARTS**: Retorna uma data a partir de parâmetros inteiros de: ano, mês e dia.

```
-- Apresenta a data de 28/10/2014  
SELECT DATEFROMPARTS(2014,10,28)
```

1.5.3. Funções de conversão

- **CAST**: Converte um tipo de dados em outro;

Exemplos:

- Convertendo uma data em número:

```
SELECT CAST(GETDATE() AS FLOAT) AS DATA_NUMERO
```

	Results	Messages
1	DATA_NUMERO 42525,5574235725	

- Convertendo um texto em data:

```
SELECT CAST('2014.1.13' AS DATETIME) AS DATA
```

	Results	Messages
1	DATA 2014-01-13 00:00:00.000	

- Convertendo um número em data:

```
SELECT CAST(42525 AS DATETIME) AS DATA
```

Results		Messages
DATA		
1	2016-06-06 00:00:00.000	

- CONVERT:** O **CONVERT** também realiza a conversão de um tipo de dados em outro, porém, é possível realizar a conversão em um formato específico por um argumento;

Vejamos alguns argumentos:

Código	País	Formato
101	EUA	1 = mm/dd/aa
103	Britânico/francês	3 = dd/mm/aa 103 = dd/mm/aaaa
111	JAPÃO	11 = aa/mm/dd 111 = aaaa/mm/dd
13 ou 113	Padrão Europa + milissegundos	dd mês aaaa hh:mi:ss:mmm (24h)
114	-	hh:mi:ss:mmm(24h)
21 ou 121		aaaa-mm-dd hh:mi:ss.mmm (24h)

Exemplos:

- Convertendo uma data para texto com **CAST**:

```
SELECT CAST (GETDATE() AS VARCHAR(20))
```

No exemplo anterior, o formato é o de data completa.

Results		Messages
(No column name)		
1	Jun 10 2016 4:29PM	

- Utilizando o **CONVERT** para retornar somente a data no formato DD/MM/YYYY:

```
SELECT CONVERT(VARCHAR(10), GETDATE() , 103)
```

Results	
(No column name)	
1	10/06/2016

- Retornando a data no formato AAAA/MM/DD:

```
SELECT CONVERT(VARCHAR(10), GETDATE() , 111)
```

Results	
(No column name)	
1	2016/06/10

- Retornando a hora:

```
SELECT CONVERT(VARCHAR(10), GETDATE() , 114)
```

Results	
(No column name)	
1	16:30:41:0

- **PARSE**: Realiza a conversão de caracteres para data/hora ou números;

A sintaxe do comando é:

```
PARSE ( string_value AS data_type [ USING culture ] )
```

Em que:

- **string_value**: Valor alfanumérico;
- **data_type**: Tipo de dados alvo que será convertido;
- **USING culture**: Código da cultura do formato da cadeia de caracteres.

Vejamos uma listagem com alguns códigos de cultura:

Nome	Língua	Código de paginação	Código da língua
British	Inglês britânico	2057	en-GB
Deutsch	Alemão	1031	de-DE
Français	Francês	1036	fr-FR
Italiano	Italiano	1040	it-IT
Português	Português	2070	pt-PT
Português (Brasil)	Português do Brasil	1046	pt-BR
us_english	Inglês	1033	en-US

Exemplos:

- Convertendo um texto em data:

```
SELECT PARSE('2016.6.10' AS datetime) AS RESULTADO;
```

RESULTADO	
1	2016-06-10 00:00:00.000

- Convertendo um texto em data utilizando o formato americano:

```
SELECT PARSE('2016.6.10' AS datetime USING 'en-US') AS RESULTADO;
```

RESULTADO	
1	2016-06-10 00:00:00.000

- Convertendo um texto em valor numérico. Verifique que o ponto é o separador de decimal e que o resultado estará correto:

```
SELECT PARSE('159.00' AS DECIMAL(10,2)) AS RESULTADO
```

RESULTADO	
1	159.00

Porém, se alterar de ponto para vírgula, a conversão ficará incorreta:

```
SELECT PARSE( '159,00' AS DECIMAL(10,2) ) AS RESULTADO
```

Results	
	Messages
RESULTADO	
1	15900.00

Para resolver este problema, utilize o formato cultural para atender ao texto passado:

```
SELECT PARSE( '159,00' AS DECIMAL(10,2) USING 'pt-BR' ) AS  
RESULTADO
```

Results	
	Messages
RESULTADO	
1	159.00

- **TRY_PARSE:** O **TRY_PARSE** realiza a verificação da conversão do **PARSE** e retorna nulo, caso exista um erro.

Exemplos:

- No exemplo adiante, é passado um caractere indevido. Utilizando o **PARSE**, é apresentado um erro:

```
SELECT PARSE( '159,A0' AS DECIMAL(10,2) USING 'PT-br' ) AS  
RESULTADO
```

Results	
	Messages
Msg 9819, Level 16, State 1, Line 14 Error converting string value '159,A0' into data type numeric using culture 'PT-br'.	

- Utilizando o **TRY_PARSE**, não é retornado um erro e sim **NULL** (nulo):

```
SELECT TRY_PARSE( '159,A0' AS DECIMAL(10,2) USING 'PT-br' ) AS  
RESULTADO
```

Results	
	Messages
RESULTADO	
1	NULL

- **TRY_CONVERT**: Realiza a conversão conforme o comando **CONVERT** e, caso ocorra erro, retorna nulo.

Exemplos:

- Utilizando **CONVERT** em uma conversão indevida:

```
SELECT CONVERT(DATETIME, '2016.06.a' , 103)
```

```
Msg 241, Level 16, State 1, Line 17
Conversion failed when converting date and/or time from character string.
```

- Com o **TRY_CONVERT**, o retorno é nulo:

```
SELECT TRY_CONVERT(DATETIME, '2016.06.a' , 103) AS RESULTADO
```

RESULTADO
1 NULL

1.6. Funções de classificação

Vejamos, nos subtópicos a seguir, as funções de classificação.

1.6.1. ROW_NUMBER

Retorna o número sequencial da linha de um resultado:

```
ROW_NUMBER( )
OVER ( [ PARTITION BY value_expression , ... [ n ] ] order_by_
clause )
```

Em que:

- **Partition_by_clause**: Coluna de agrupamento;
- **Order by**: Coluna para a classificação, que pode ser crescente ou decrescente.

Exemplo 1:

A consulta adiante apresenta o nome do cliente, a quantidade de pedidos e uma coluna com a sequência das linhas, ordenada pela quantidade de pedidos de cada cliente:

```
USE PEDIDOS
GO
SELECT NOME , TOTAL, ROW_NUMBER() OVER (ORDER BY TOTAL DESC) AS
LINHA
FROM
(SELECT C.NOME , SUM(P.VLR_TOTAL) AS TOTAL
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI
GROUP BY C.NOME
) AS A
```

Vejamos o resultado da consulta:

NOME	TOTAL	LINHA
JOSE CAMILO NETO	171689.93	1
JOSE ANTONIO DE OLIVEIRA JUNIOR	132095.01	2
GELSON PEDRO MOSCHEN	129791.07	3
ALAMBRINDES GRAFICA EDITORA LTDA.	112510.02	4
ISAIAS	111033.06	5
PHILADELPHIA COM DE BRINDES	108433.24	6
EDSON TARIFA VARGAS	103513.46	7
J.J.M.	101155.53	8
BRINDES TATUAPE LTDA-ME	100375.40	9
BRIDA BRINDES LTDA.	99976.04	10
MR BRINDES PROMOCIONAIS S/C LTDA	99271.43	11
BIJOUTERIAS SAO PAULO	99226.09	12
PROMOCART BRINDES E FOLINHAS LTDA	98535.92	13
SAVANE COM. SERVICO LTDA	96733.36	14
IRUGRAF COMUNICACAO VISUAL LTDA. ME.	96503.27	15
MAURICIO MARQUES RIBEIRO	95569.20	16
L.NR.COMERCIO E REPRESENTACOES LTDA	94592.18	17
ELISEU BUENO-ME	92659.76	18

Exemplo 2:

Na próxima consulta, foi acrescentado o estado de cada um dos clientes e a sequência foi particionada para sequenciar sobre o estado:

```
SELECT NOME , QTD_PEDIDOS, ESTADO, ROW_NUMBER() OVER (PARTITION BY  
ESTADO ORDER BY QTD_PEDIDOS DESC) AS LINHA  
  
FROM  
(SELECT C.NOME , C.ESTADO, COUNT(*) AS QTD_PEDIDOS  
FROM TB_PEDIDO AS P  
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI  
GROUP BY C.NOME, C.ESTADO  
) AS A
```

Verifique que a cada novo estado é inicializada a contagem de cada linha, gerando uma nova consulta:

NOME	QTD_PEDIDOS	ESTADO	LINHA
TESTE	27	NULL	1
MILTON LOSANO	21	NULL	2
HELP MED APOIO HOSPITALAR E LABORATORIAL LTDA	19	NULL	3
IGLESIAS BRINDES	18	NULL	4
CARLOS FERNANDO LIMA SOUZA-ME	17	NULL	5
SIDNEY	17	NULL	6
AMADEU	13	NULL	7
EFIGENIA	12	NULL	8
JOSE MARIA NASCIMENTO MARTINS	21	AM	1
CIDIA MARIA BARBOSA SOUZA	16	AM	2
MARIA SOCORRO MORAES	14	AM	3
D & D BRINDES LTDA	11	AM	4
PROGRA MART COM REPRES.SERV.SERIG.LTDA	28	BA	1
GRAFICA LINEL LTDA.	27	BA	2
JOAO ANDRADE DE OLIVEIRA ME	26	BA	3
BABY KILO COM.LTDA.	24	BA	4
RICARDO COONGE	21	BA	5
VALE BRINDES IND.COM.DE BRINDES DO V.S.F. LTDA. ME	21	BA	6

1.6.2. RANK

Retorna a classificação da linha em relação a um grupo de dados definidos. Quando o valor da partição é o mesmo, o valor é repetido.

```
RANK ( ) OVER ( [ partition_by_clause ] order_by_clause )
```

Em que:

- **Partition_by_clause:** Coluna que será utilizada para definição do RANK;
- **Order by:** Classificação da pesquisa para geração do ranking.

Exemplo 1:

A consulta adiante apresenta a classificação das vendas dos clientes. Verifique que a função RANK está mostrando a classificação sobre a coluna QTD_PEDIDOS:

```
USE PEDIDOS
GO
SELECT NOME , QTD_PEDIDOS, RANK() OVER (ORDER BY QTD_PEDIDOS DESC)
AS [CLASSIFICAÇÃO]

FROM
(SELECT C.NOME , COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI
GROUP BY C.NOME
) AS A
```

Observe que, ao repetir os valores, o resultado do RANK é repetido. O próximo valor é a sequência após a quantidade de registros anteriores:

NOME	QTD_PEDIDOS	CLASSIFICAÇÃO
PHILADELPHIA COM DE BRINDES	83	1
FORT BRINDES	59	2
BRIDA BRINDES LTDA.	55	3
JOSE CAMILO NETO	55	3
MERCURY CONF E PROD PROM LTDA	54	5
JOSE ANTONIO DE OLIVEIRA JUNIOR	48	6
PROMOCART BRINDES E FOLINHAS LTDA	42	7
COLOR SCREEN PRODUTOS PROMOCIONAIS LTDA.	42	7
SUELENE APARECIDA RIBEIRO	41	9
MAURICIO MARQUES RIBEIRO	39	10
APOLO COM. DE ENVELOPES LTDA.	39	10
GELSON PEDRO MOSCHEN	39	10
FARIA E CAROSELLI C.BRINDES LTDA	36	13
DOUGAR IND. E COM. DE BRINDES LTDA.	35	14
BRINDES PROM.ARTE.S.ART-WEG LTDA-ME	35	14
CLASSIC PEN COMERCIO DE BRINDES LTDA.	33	16
ADRIANO J.SILVA BRINDES-ME	33	16
DORA	33	16

Exemplo 2:

A consulta a seguir apresenta a classificação da quantidade de pedidos dos clientes por estado:

```
SELECT NOME , ESTADO, QTD_PEDIDOS, RANK() OVER (PARTITION BY ESTADO
ORDER BY QTD_PEDIDOS DESC) AS [CLASSIFICAÇÃO]
FROM
(SELECT C.NOME , C.ESTADO, COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI
GROUP BY C.NOME, C.ESTADO
) AS A
Order by Estado
```

A consulta adiante apresenta os resultados da classificação dos clientes por estado, sobre a quantidade de pedidos. Da mesma maneira, caso existam valores iguais, o próximo valor será a sequência acrescida da quantidade de linhas:

NOME	ESTADO	QTD_PEDIDOS	CLASSIFICAÇÃO
PROGRA MART COM REPRES.SERV.SERIG.LTDA	BA	28	1
GRAFICA LINEL LTDA.	BA	27	2
JOAO ANDRADE DE OLIVEIRA ME.	BA	26	3
BABY KILO COM.LTDA.	BA	24	4
RICARDO COONGE	BA	21	5
VALE BRINDES IND.COM.DE BRINDES DO V.S.F. LTDA. ME	BA	21	5
SUL BRINDES COMERCIO E REPRESENTACAO LTDA	BA	20	7
JOSE ALMEIDA N.FILHO	BA	20	7
NOELIA REIS DA SILVA - ME	BA	19	9
SILVANA DUARTE SILVA	BA	18	10
MARIA JONILZA DE OLIVEIRA ME	BA	18	10
ERNANDES RAIMUNDO DE SOUZA	BA	18	10
GABI BRINDES LTDA	BA	17	13
VANDERLEI VIANA DOS SANTOS	BA	17	13
LUCANEL COM E REPRES DE BRINDES E FOLH.LTDA	BA	17	13
BRINDESLAY COMERCIO DE BRINDES E PRESENTES LT...	BA	17	13
ANTONIO JOSE ALVES DE MARINHO-ME	BA	17	13
EDUARDO VARJAO RODRIGUES	BA	15	18

1.6.3. DENSE_RANK

Retorna a classificação da linha em relação a um grupo de dados definidos. O DENSE_RANK continuará a sequência da classificação.

```
DENSE_RANK ( ) OVER ( [ partition_by_clause ] order_by_clause )
```

Em que:

- **Partition_by_clause:** Coluna que será utilizada para definição do **DENSE_RANK**;
- **Order by:** Classificação da pesquisa para geração do ranking.

Exemplo 1:

A consulta a seguir apresenta a classificação das vendas dos clientes. Verifique que a função **DENSE_RANK** está mostrando a classificação sobre a coluna **QTD_PEDIDOS**:

```
USE PEDIDOS
GO

SELECT NOME , QTD_PEDIDOS, DENSE_RANK() OVER (ORDER BY QTD_PEDIDOS
DESC) AS [CLASSIFICAÇÃO]

FROM
(SELECT C.NOME , COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI
GROUP BY C.NOME
) AS A
```

Observe que o resultado da classificação continuará o valor da sequência:

NOME	QTD_PEDIDOS	CLASSIFICAÇÃO
PROMOCART BRINDES E FOLINHAS LTDA	42	6
COLOR SCREEN PRODUTOS PROMOCIONAIS LTDA.	42	6
SUELENE APARECIDA RIBEIRO	41	7
MAURICIO MARQUES RIBEIRO	39	8
APOLÓ COM. DE ENVELOPES LTDA.	39	8
GELSON PEDRO MOSCHEN	39	8
FARIA E CAROSELLI C.BRINDES LTDA	36	9
DOUGAR IND. E COM. DE BRINDES LTDA	35	10
BRINDES PROM.ARTE.S.ART WEG LTDA-ME	35	10
CLASSIC PEN COMERCIO DE BRINDES LTDA.	33	11
ADRIANO J.SILVA BRINDES-ME	33	11
DORA	33	11
EDSON TARIFA VARGAS	33	11
DIVINO ETERNO DAS GRACAS-ME	33	11
MARCO ANTONIO DINARDI MIGLIARI	33	11
VIA BRINDES LTDA -ME	32	12
HIRAFUJI COM REPRRES DE BRINDES LTDA	32	12
BRINDES TATUAPE LTDA-ME	32	12

Exemplo 2:

A consulta adiante apresenta a classificação da quantidade de pedidos dos clientes por estado:

```
SELECT NOME , ESTADO, QTD_PEDIDOS, DENSE_RANK() OVER (PARTITION BY  
ESTADO ORDER BY QTD_PEDIDOS DESC) AS [CLASSIFICAÇÃO]  
FROM  
(SELECT C.NOME , C.ESTADO, COUNT(*) AS QTD_PEDIDOS  
FROM TB_PEDIDO AS P  
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI  
GROUP BY C.NOME, C.ESTADO  
) AS A  
Order by Estado
```

Verifique que o resultado não apresenta saltos de valores:

NOME	ESTADO	QTD_PEDIDOS	CLASSIFICAÇÃO
TESTE	NULL	27	1
MILTON LOSANO	NULL	21	2
HELP MED APOIO HOSPITALAR E LABORATORIAL LTDA	NULL	19	3
IGLESIAS BRINDES	NULL	18	4
CARLOS FERNANDO LIMA SOUZA-ME	NULL	17	5
SIDNEY	NULL	17	5
AMADEU	NULL	13	6
EFIGENIA	NULL	12	7
JOSE MARIA NASCIMENTO MARTINS	AM	21	1
CIDIA MARIA BARBOSA SOUZA	AM	16	2
MARIA SOCORRO MORAES	AM	14	3
D & D BRINDES LTDA	AM	11	4
PROGRA MART COM REPRES.SERV.SERIG.LTDA	BA	28	1
GRAFICA LINEL LTDA.	BA	27	2
JOAO ANDRADE DE OLIVEIRA ME.	BA	26	3
BABY KILO COM.LTDA	BA	24	4
RICARDO COONGE	BA	21	5
VALE BRINDES IND.COM.DE BRINDES DO V.S.F. LTDA. ME	BA	21	5

1.6.4. NTILE

Cria uma coluna de agrupamento sobre o total de linhas.

```
NTILE (integer_expression) OVER ( [ <partition_by_clause> ] < order_<br/>by_clause > )
```

Exemplo 1:

Na consulta adiante, o resultado será dividido em 10 grupos:

```
SELECT NOME , TOTAL, ESTADO, NTILE (10) OVER (ORDER BY TOTAL DESC)
AS GRUPO
FROM
(SELECT C.NOME , C.ESTADO, SUM(P.VLR_TOTAL) AS TOTAL
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI
GROUP BY C.NOME, C.ESTADO
) AS A
```

O resultado apresenta as linhas agrupadas em 10 grupos:

NOME	TOTAL	ESTADO	GRUPO
JOAO ANDRADE DE OLIVEIRA ME.	84013.23	BA	1
TESTE	83756.15	NULL	1
ALEXANDRE MAURO ALVES	83144.67	SP	1
FABIO SANTANA PIRES	81774.09	RJ	1
RAMOS ILHA COMERCIO E REPRESENTAC...	80991.96	RJ	1
GILMAR PEREIRA PINTO	80184.47	SP	1
MAGDA CONTIGIO	80124.26	GO	1
BIJOUTERIAS FAN LTDA.	79603.56	SP	1
DINAMICA BRINDES E PUBLICIDADE LTDA.	79379.52	MG	1
FACTORING ALISON DE FORMENTOS COM...	79310.23	SP	1
JOINVILLE COM DE BRINDES	78731.56	SC	1
MARCOS ANTONIO ALVES	78417.66	SP	1
JOAO RONALDO DA SILVA	78321.41	SP	1
CLEMENTE GONCALVES CUNHA	78228.68	MG	1
CELSO PEREIRA	78052.57	PR	1
VIA BRINDES LTDA -ME	77790.68	SP	1
MARCO ANTONIO DINARDI MIGLIARI	77200.67	SP	1
DISTR. SAO PAULO ARMARIMHOS LTDA.	76865.41	SP	1
SIRUZ DISTR. DE BRINDES E EMBALAGEN...	76600.38	PR	1
PROGERAL IND. DE ARTEFATOS PLASTICO...	76591.55	SP	1
KATIA VICENTE	76179.65	SP	1
MARCIO C. CORDEIRO	75727.80	MG	1
SHOPING BRINDES MARANBAIA LTDA	75707.86	RJ	1
COLLATO PRODUCOES SERIGRAFICAS E C...	75664.08	SP	2
(NINO)ANTONIO ROSA FILHO	75465.14	SP	2
ATLAS BRINDES	75277.66	SP	2
VILSON CORDEIRO DO ROSARIO	75093.06	SP	2

1.6.5. ROW_NUMBER, RANK, DENSE_RANK e NTILE

A consulta adiante apresenta as colunas com as funções respectivas de classificação:

```

SELECT NOME , ESTADO, QTD_PEDIDOS,
ROW_NUMBER() OVER (PARTITION BY ESTADO ORDER BY QTD_PEDIDOS DESC) AS
ROW_NUMBER,
RANK()    OVER (PARTITION BY ESTADO ORDER BY QTD_PEDIDOS DESC) AS
RANK,
DENSE_RANK() OVER (PARTITION BY ESTADO ORDER BY QTD_PEDIDOS DESC) AS
DENSE_RANK,
NTILE (10)      OVER (PARTITION BY ESTADO ORDER BY QTD_PEDIDOS DESC)
AS NTILE
FROM
(SELECT C.NOME ,C.ESTADO, COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.CODCLI = P.CODCLI
GROUP BY C.NOME,C.ESTADO
) AS A

```

Resultado:

NOME	ESTADO	QTD_PEDIDOS	ROW_NUMBER	RANK	DENSE_RANK	NTILE
TESTE	NULL	27	1	1	1	1
MILTON LOSANO	NULL	21	2	2	2	2
HELP MED APOIO HOSPITALAR E LABORATORIAL LTDA	NULL	19	3	3	3	3
IGLESIAS BRINDES	NULL	19	4	4	4	4
CARLOS FERNANDO LIMA SOUZA-ME	NULL	17	5	5	5	5
SIDNEY	NULL	17	6	5	5	6
AMADEU	NULL	13	7	7	6	7
EFIGENIA	NULL	12	8	8	7	8
JOSE MARIA NASCIMENTO MARTINS	AM	21	1	1	1	1
CIDIA MARIA BARBOSA SOUZA	AM	16	2	2	2	2
MARIA SOCORRO MORAES	AM	14	3	3	3	3
D & D BRINDES LTDA	AM	11	4	4	4	4
PROGRA MART COM REPPRES.SERV.SERIG.LTDA	BA	28	1	1	1	1
GRAFICA LINEL LTDA	BA	27	2	2	2	1
JOAO ANDRADE DE OLIVEIRA ME.	BA	26	3	3	3	2
BABY KILO COM.LTDA.	BA	24	4	4	4	2
RICARDO COONGE	BA	21	5	5	5	3
VALE BRINDES IND.COM.DE BRINDES DO V.S.F. LTDA. ME	BA	21	6	5	5	3
SUL BRINDES COMERCIO E REPRESENTACAO LTDA	BA	20	7	7	6	4
JOSE ALMEIDA N.FILHO	BA	20	8	7	6	4
NOELIA REIS DA SILVA - ME	BA	19	9	9	7	5
SILVANA DUARTE SILVA	BA	18	10	10	8	5
MARIA JONILZA DE OLIVEIRA ME	BA	18	11	10	8	6
ERNANDES RAIMUNDO DE SOUZA	BA	18	12	10	8	6
GABI BRINDES LTDA	BA	17	13	13	9	7
VANDERLEI VIANA DOS SANTOS	BA	17	14	13	9	7
LUCANEL COM E REPRES DE BRINDES E FOLH.LTDA	BA	17	15	13	9	8

1.7. Funções definidas pelo usuário

Como uma função nativa, uma **função definida pelo usuário** é uma rotina que aceita parâmetros, executa uma ação e retorna, como resultado, um valor. Elas podem ser chamadas em uma consulta, instrução ou expressão. Para criar uma função definida pelo usuário, utilizamos o comando **CREATE FUNCTION**.

Utilizar funções definidas pelo usuário proporciona os seguintes benefícios:

- Programação modular, já que a função, uma vez criada, pode ser armazenada no banco de dados e ser chamada quantas vezes forem necessárias;
- Execução mais rápida, pois reduz o custo de compilação do código, armazenando os planos em cache e reutilizando-os em repetidas execuções. Assim, não há necessidade de analisar uma função cada vez que ela for utilizada;
- Redução no tráfego de rede, já que uma função pode filtrar dados com base em uma restrição complexa e que não pode ser expressa em somente uma expressão escalar. Assim, pode ser usada em uma cláusula **WHERE** para reduzir o número de linhas enviadas ao cliente.

Ao criar uma função de usuário, é necessário que seu nome especifique também o nome do schema no qual está inserida. Para a execução dessa função, é exigida uma permissão de **SELECT**. Além disso, em uma função, as instruções contidas em um bloco **BEGIN...END** não podem ter efeitos negativos. Erros que interrompem uma instrução e continuam com a instrução seguinte são tratados de modo diferente em uma função. Uma função especificada em uma consulta pode ser executada diversas vezes, dependendo dos planos de execução que foram definidos pelo otimizador.

Uma função de usuário não pode ser usada para alterar dados, nem pode definir ou criar novos objetos no banco de dados. Ou seja, em uma função definida pelo usuário, não se pode utilizar os comandos **UPDATE**, **INSERT** e **DELETE**. Todo objeto referenciado por uma função deve ter sido criado ou declarado antes, com exceção de um tipo escalar. Além disso, em uma função de usuário não podem ser realizadas transações.

! Cuidado
Não se pode criar uma função de usuário com uma cláusula **OUTPUT INTO** que tenha como destino uma tabela.

1.7.1. Funções escalares

Adiante, temos diversos exemplos de funções escalares:

```
USE PEDIDOS
```

- Função para receber dois números **INT** e retornar o maior dos dois:

```
CREATE FUNCTION FN_MAIOR( @N1 INT, @N2 INT )
RETURNS INT
AS BEGIN
DECLARE @RET INT;
IF @N1 > @N2
    SET @RET = @N1
ELSE
    SET @RET = @N2;
RETURN (@RET)
END

GO
-- Testando
SELECT DBO.FN_MAIOR( 5,3 )
SELECT DBO.FN_MAIOR( 7,11 )
```

- Função para receber uma data e retornar o nome do dia da semana sempre em português, independentemente das configurações do servidor:

```
CREATE FUNCTION FN_NOME_DIA_SEMANA( @DT DATETIME )
RETURNS VARCHAR(15)
AS BEGIN
DECLARE @NUM_DS INT, @NOME_DS VARCHAR(15);
SET @NUM_DS = DATEPART( WEEKDAY, @DT );
/*
IF @NUM_DS = 1 SET @NOME_DS = 'DOMINGO';
IF @NUM_DS = 2 SET @NOME_DS = 'SEGUNDA-FEIRA';
IF @NUM_DS = 3 SET @NOME_DS = 'TERÇA-FEIRA';
IF @NUM_DS = 4 SET @NOME_DS = 'QUARTA-FEIRA';
IF @NUM_DS = 5 SET @NOME_DS = 'QUINTA-FEIRA';
IF @NUM_DS = 6 SET @NOME_DS = 'SEXTA-FEIRA';
IF @NUM_DS = 7 SET @NOME_DS = 'SÁBADO';
*/
-- OU
SET @NOME_DS = CASE @NUM_DS
WHEN 1 THEN 'DOMINGO'
WHEN 2 THEN 'SEGUNDA-FEIRA'
WHEN 3 THEN 'TERÇA-FEIRA'
WHEN 4 THEN 'QUARTA-FEIRA'
WHEN 5 THEN 'QUINTA-FEIRA'
WHEN 6 THEN 'SEXTA-FEIRA'
WHEN 7 THEN 'SÁBADO'
```

```
END -- CASE
RETURN (@NOME_DS)
END
GO
-- TESTANDO
SELECT NOME, DATA ADMISSAO, DATENAME(WEEKDAY,DATA ADMISSAO),
DBO.FN_NOME_DIA_SEMANA( DATA ADMISSAO )
FROM TB_EMPREGADO
--
SELECT DBO.FN_NOME_DIA_SEMANA('2013.11.12' )

SELECT DBO.FN_NOME_DIA_SEMANA(GETDATE() )
```

- Função para gerar um número aleatório em um determinado intervalo. Passaremos para a função o valor mínimo e o valor máximo para o sorteio:

```
-- Versão 1
GO

CREATE FUNCTION FN_SORTEIO( @MIN FLOAT, @MAX FLOAT )
RETURNS FLOAT
AS BEGIN
RETURN @MIN + (@MAX - @MIN) * RAND();
END

GO
```

Resultado:

Messages

Msg 443, Level 16, State 1, Procedure FN_SORTEIO, Line 166
Invalid use of a side-effecting operator 'rand' within a function.

Não conseguiremos criar esta função da forma como ela foi escrita. A maioria das funções não determinísticas não pode ser utilizada em funções de usuário, inclusive `RAND()`. Será necessária, então, outra saída:

```
-- Versão 2
CREATE VIEW VIE_RAND
AS SELECT RAND() AS NUM_RAND

GO

CREATE FUNCTION FN_SORTEIO( @MIN FLOAT, @MAX FLOAT )
RETURNS FLOAT
AS BEGIN
DECLARE @RAND FLOAT;
SELECT @RAND = NUM_RAND FROM VIE_RAND

RETURN @MIN + (@MAX - @MIN + 1) * @RAND;
END
-- Testando
GO
SELECT DBO.FN_SORTEIO( 1,60 )
```

- Função para eliminar a parte **hora** de uma variável ou campo **DATETIME**. Os comandos **INSERT** a seguir inserem dois registros na tabela **TB_EMPREGADO** com o campo **DATA_ADMISSAO** preenchido com data e hora atuais:

```
INSERT INTO TB_EMPREGADO
(NOME, COD_DEPTO, COD_CARGO, SALARIO, DATA_ADMISSAO)
VALUES ('ZÉ LUIZ', 1, 1, 1000, GETDATE())

INSERT INTO TB_EMPREGADO
(NOME, COD_DEPTO, COD_CARGO, SALARIO, DATA_ADMISSAO)
VALUES ('ZÉ DA SILVA', 2, 2, 2000, GETDATE())
```

Se formos consultar os funcionários admitidos numa data específica, teremos um resultado vazio, caso seja executado o **SELECT** a seguir:

```
SELECT * FROM TB_EMPREGADO
WHERE DATA_ADMISSAO = '2006.6.24'
```

Isso ocorre porque o campo **DATA_ADMISSAO** foi preenchido também com a hora. Teríamos, então, que fazer o seguinte:

```
SELECT * FROM TB_EMPREGADO
WHERE DATA_ADMISSAO BETWEEN '2006.6.24' AND '2006.6.24 23:59'
```

A função FN_TRUNCA_DATA criada a seguir resolve esse problema:

```
CREATE FUNCTION FN_SORTEIO( @MIN FLOAT, @MAX FLOAT )
RETURNS FLOAT
AS BEGIN
RETURN @MIN + (@MAX - @MIN) * RAND();
END

GO

CREATE FUNCTION FN_TRUNCA_DATA( @DT DATETIME ) RETURNS DATETIME
AS BEGIN
RETURN CAST( FLOOR( CAST(@DT AS FLOAT) ) AS DATETIME )
END
GO
-- Testando
SELECT * FROM TB_EMPREGADO
WHERE DBO.FN_TRUNCA_DATA( DATA ADMISSAO ) = '2006.2.24'
-- OU
SELECT * FROM TB_EMPREGADO
WHERE DBO.FN_TRUNCA_DATA( DATA ADMISSAO ) =
DBO.FN_TRUNCA_DATA( GETDATE() )
```

- Função para gerar a primeira data do mês referente a uma data fornecida (por exemplo, a primeira data do mês referente a 20/03/2014 é 01/03/2014):

```
CREATE FUNCTION FN_PRIM_DATA( @DT DATETIME )
RETURNS DATETIME
AS BEGIN
DECLARE @RET DATETIME;

SET @RET = DATETIMEFROMPARTS( YEAR(@DT) , MONTH(@DT) , 1,0,0,0,0);

RETURN @RET;
END
GO
-- Testando
SELECT NOME, DATA ADMISSAO, DBO.FN_PRIM_DATA(DATA ADMISSAO)
FROM TB_EMPREGADO
--
```

- Função para calcular a "N-ésima" data útil a partir de uma data fornecida. Passaremos para esta função uma data e a quantidade de dias úteis que queremos contar. A função retornará uma data futura, somando a quantidade de dias e descontando sábados e domingos:

```
CREATE FUNCTION FN_N_ESIMA_DATA_UTIL ( @DT DATETIME,
                                         @N INT )
    RETURNS DATETIME
AS BEGIN
    DECLARE @I INT;
    SET @I = 0;
    WHILE @I < @N
        BEGIN
            SET @DT = @DT + 1;
            IF DATEPART(WEEKDAY, @DT) IN (1,7) CONTINUE
            SET @I = @I + 1;
        END
    RETURN ( @DT );
END
GO
-- TESTANDO
SELECT DBO.FN_N_ESIMA_DATA_UTIL( GETDATE(), 5 )
```

Se quisermos descontar também os feriados, precisaremos criar uma tabela onde eles serão cadastrados:

```
CREATE TABLE FERIADOS
( DATA DATETIME, MOTIVO VARCHAR(40) )
GO
--
INSERT INTO FERIADOS VALUES ('2014.1.25','ANIV. SÃO PAULO')
INSERT INTO FERIADOS VALUES ('2014.2.21','CARNAVAL')
INSERT INTO FERIADOS VALUES ('2014.2.22','CARNAVAL')
INSERT INTO FERIADOS VALUES ('2014.2.23','CARNAVAL')
INSERT INTO FERIADOS VALUES ('2014.2.24','CARNAVAL')

INSERT INTO FERIADOS VALUES ('2014.3.2','RESSACA')

INSERT INTO FERIADOS VALUES ('2014.3.10','PÁSCOA')
INSERT INTO FERIADOS VALUES ('2014.4.19','TIRADENTES')

INSERT INTO FERIADOS VALUES ('2014.5.1','TRABALHO')
INSERT INTO FERIADOS VALUES ('2014.5.2','TRABALHO PROLONGAMENTO')

INSERT INTO FERIADOS VALUES ('2014.6.11','CORPUS CHRISTI')
INSERT INTO FERIADOS VALUES ('2014.6.12','CORPUS CHRISTI
PROLONGAMENTO')
INSERT INTO FERIADOS VALUES ('2014.6.13','CORPUS CHRISTI
PROLONGAMENTO')

INSERT INTO FERIADOS VALUES ('2014.2.16','DIA DA RESSACA')
```

Alterando a função, temos:

```
CREATE FUNCTION FN_N_ESIMA_DATA_UTIL ( @DT DATETIME,
                                         @N INT )
    RETURNS DATETIME
AS BEGIN
    DECLARE @I INT;
    SET @I = 0;
    WHILE @I < @N
        BEGIN
            SET @DT = @DT + 1;
            IF DATEPART(WEEKDAY, @DT) IN (1,7) OR
                EXISTS( SELECT * FROM FERIADOS
                         WHERE DATA = DBO.FN_TRUNCA_DATA( @DT ) ) CONTINUE
            SET @I = @I + 1;
        END
    RETURN ( @DT );
END
GO
-- Testando
SELECT DBO.FN_N_ESIMA_DATA_UTIL( GETDATE(), 5 )
```

- Função para calcular a diferença entre duas datas:

O SQL Server possui uma função chamada **DATEDIFF**, que serve para calcular a diferença entre duas datas.

```
SELECT DATEDIFF(DAY, '2009.1.1', '2009.1.15')
```

Porém, na maioria das vezes, ela não funciona de forma adequada. No exemplo a seguir, note que a diferença entre as datas ainda não é de um (1) mês, somente seria se o dia da segunda data fosse maior ou igual a 20. Contudo, a função retorna 1.

```
SELECT DATEDIFF(MONTH, '2008.12.20', '2009.1.15')
```

Neste outro exemplo, a diferença é de quase sete meses, mas a função retorna um ano de diferença:

```
CREATE FUNCTION FN_DIF_DATAS( @TIPO CHAR(1),
                               @DT1 DATETIME,
                               @DT2 DATETIME )
    RETURNS FLOAT
AS BEGIN
    DECLARE @DIA1 INT, @MES1 INT, @ANO1 INT;
    DECLARE @DIA2 INT, @MES2 INT, @ANO2 INT;
    DECLARE @RET FLOAT;

    SET @DIA1 = DAY(@DT1);
    SET @MES1 = MONTH(@DT1);
    SET @ANO1 = YEAR(@DT1);
```

```
SET @DIA2 = DAY(@DT2);
SET @MES2 = MONTH(@DT2);
SET @ANO2 = YEAR(@DT2);

IF @TIPO = 'D'
    SET @RET = CAST(@DT2 - @DT1 AS INT);
ELSE IF @TIPO = 'M'
    BEGIN
        IF @MES1 <= @MES2
            BEGIN
                SET @RET = 12 * (@ANO2 - @ANO1) + (@MES2 - @MES1)
                IF @DIA1 > @DIA2 SET @RET = @RET - 1;
            END
        ELSE
            BEGIN
                SET @RET = 12 * (@ANO2 - (@ANO1 + 1)) + (12 - (@MES1 - @
MES2));
                IF @DIA1 > @DIA2 SET @RET = @RET - 1;
            END
        END
    ELSE
        BEGIN
            SET @RET = @ANO2 - @ANO1;
            IF @MES1 > @MES2 SET @RET = @RET - 1;
            IF @MES1 = @MES2 AND @DIA1 > @DIA2 SET @RET = @RET - 1;
        END
    RETURN @RET;
END
GO
-- TESTANDO
SELECT DBO.FN_DIF_DATAS('D', '2009.1.1', '2009.1.15')

SELECT DBO.FN_DIF_DATAS('M', '2008.1.2', '2009.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2008.1.20', '2009.2.15')

SELECT DBO.FN_DIF_DATAS('M', '2008.8.2', '2009.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2007.8.2', '2009.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2007.8.20', '2009.2.15')

SELECT DBO.FN_DIF_DATAS('A', '2008.8.2', '2009.2.15')
SELECT DBO.FN_DIF_DATAS('A', '2006.8.2', '2009.10.15')
SELECT DBO.FN_DIF_DATAS('A', '2006.10.2', '2009.10.15')
SELECT DBO.FN_DIF_DATAS('A', '2006.11.2', '2009.10.15')
SELECT DBO.FN_DIF_DATAS('A', '2006.10.20', '2009.10.20')
```

- Função para retornar a primeira palavra de um nome completo:

```
CREATE FUNCTION FN_PRIM_NOME( @S VARCHAR(200) )
    RETURNS VARCHAR(200)
AS BEGIN
DECLARE @RET VARCHAR(200);
DECLARE @CONT INT;
SET @S = LTRIM( @S );
SET @RET = '';
SET @CONT = 1;
WHILE @CONT <= LEN(@S)
BEGIN
IF SUBSTRING(@S, @CONT, 1) = ' ' BREAK;
SET @RET = @RET + SUBSTRING(@S, @CONT, 1);
SET @CONT = @CONT + 1;
END
RETURN @RET;
END
GO
-- TESTANDO
SELECT DBO.FN_PRIM_NOME( 'CARLOS MAGNO' )
SELECT DBO.FN_PRIM_NOME( ' CARLOS MAGNO' )
```

- Função para formatar nomes próprios, com a primeira letra de cada nome em caixa alta e as restantes em caixa baixa:

```
CREATE FUNCTION FN_PROPER( @S VARCHAR(200) )
    RETURNS VARCHAR(200)
AS BEGIN
DECLARE @RET VARCHAR(200);
DECLARE @CONT INT;
SET @RET = UPPER(LEFT(@S,1));
SET @CONT = 2;
WHILE @CONT <= LEN(@S)
BEGIN
IF SUBSTRING(@S, @CONT - 1, 1) = ' '
    SET @RET = @RET + UPPER( SUBSTRING(@S, @CONT, 1) )
ELSE
    SET @RET = @RET + LOWER( SUBSTRING(@S, @CONT, 1) )

    SET @CONT = @CONT + 1
END
RETURN @RET;
END
GO
-- Testando
SELECT NOME, DBO.FN_PROPER( NOME ) FROM TB_EMPREGADO
```

- Função para substituir caracteres acentuados por caracteres sem acento:

Nas duas consultas adiante, estamos procurando na tabela **TB_EMPREGADO** pessoas que tenham a palavra **JOSÉ** contida no nome. Precisamos fazer duas consultas, uma utilizando acento agudo no E e outra sem acento, visto que as pessoas que cadastraram os nomes podem ter utilizado acento ou não. Pode haver, inclusive, nomes com acentuação errada, como **JÓSE** ou **JÔSE**, por exemplo.

```
SELECT * FROM TB_EMPREGADO WHERE NOME LIKE '%JOSE%'  
SELECT * FROM TB_EMPREGADO WHERE NOME LIKE '%JOSÉ%'
```

Para resolver esse problema, podemos criar a função a seguir:

```
CREATE FUNCTION FN_TIRA_ACENTO( @S VARCHAR(200) )  
RETURNS VARCHAR( 200 )  
AS BEGIN  
DECLARE @I INT, @RET VARCHAR(200), @C CHAR(1);  
SET @I = 1;  
SET @RET = '';  
-- Enquanto @I for menor que o tamanho de @S  
WHILE @I <= LEN(@S)  
BEGIN  
SET @C = SUBSTRING( @S, @I, 1 );  
SET @RET = @RET +  
CASE  
WHEN ASCII(@C) IN (ASCII('Ã'), ASCII('Á'),  
ASCII('À'),  
ASCII('Â')) THEN 'A'  
WHEN ASCII(@C) IN (ASCII('ã'), ASCII('á'),  
ASCII('à'),  
ASCII('â')) THEN 'a'  
WHEN ASCII(@C) IN (ASCII('É'), ASCII('Ê')) THEN 'E'  
WHEN ASCII(@C) IN (ASCII('é'), ASCII('ê')) THEN 'e'  
WHEN ASCII(@C) IN (ASCII('Í')) THEN 'I'  
WHEN ASCII(@C) IN (ASCII('í')) THEN 'i'  
WHEN ASCII(@C) IN (ASCII('Ó'), ASCII('Õ'), ASCII('Ô'))  
THEN 'O'  
WHEN ASCII(@C) IN (ASCII('ó'), ASCII('õ'), ASCII('ô'))  
THEN 'o'  
WHEN ASCII(@C) IN (ASCII('Ú'), ASCII('Ü')) THEN 'U'  
WHEN ASCII(@C) IN (ASCII('ú'), ASCII('ü')) THEN 'u'  
WHEN ASCII(@C) = ASCII('Ç') THEN 'C'  
WHEN ASCII(@C) = ASCII('ç') THEN 'c'  
ELSE @C  
END -- CASE  
SET @I = @I + 1  
END  
RETURN (@RET);  
END  
GO
```

```
-- TESTANDO
SELECT NOME FROM TB_EMPREGADO
WHERE DBO.FN_TIRA_ACENTO( NOME ) LIKE '%JOSE%'

SELECT NOME FROM TB_EMPREGADO
WHERE DBO.FN_TIRA_ACENTO( NOME ) LIKE '%JOAO%'
```

1.7.2. Funções tabulares

Adiante, temos exemplos de funções tabulares:

- Função para retornar o valor do maior pedido vendido em cada um dos meses de determinado período:

```
CREATE FUNCTION FN_MAIOR_PEDIDO( @DT1 DATETIME,
                                  @DT2 DATETIME )
RETURNS TABLE
AS
RETURN ( SELECT MONTH( DATA_EMISSAO ) AS MES,
            YEAR( DATA_EMISSAO ) AS ANO,
            MAX( VLR_TOTAL ) AS MAIOR_VALOR
        FROM TB_PEDIDO
        WHERE DATA_EMISSAO BETWEEN @DT1 AND @DT2
        GROUP BY MONTH( DATA_EMISSAO ),
                 YEAR( DATA_EMISSAO ) )
GO
-- Testando
SELECT * FROM DBO.FN_MAIOR_PEDIDO( '2014.1.1', '2014.12.31' )
ORDER BY ANO, MES
```

- Função para listar os funcionários de cada departamento e a soma dos salários de cada departamento:

```
CREATE FUNCTION FN_TOT_DEPTOS()
RETURNS @TotDeptos TABLE ( COD_DEPTO INT,
                           NOME VARCHAR(40),
                           TIPO CHAR(1),
                           VALOR NUMERIC(12,2) )
AS BEGIN
DECLARE @I INT; -- Contador
DECLARE @TOT INT; -- Total de departamentos existentes
SELECT @TOT = MAX(COD_DEPTO) FROM TB_DEPARTAMENTO
SET @I = 1;
WHILE @I <= @TOT
BEGIN
    -- Se existir o departamento de código = @I...
    IF EXISTS( SELECT * FROM TB_DEPARTAMENTO
                WHERE COD_DEPTO = @I )
```

```
BEGIN
    -- Inserir na tabela de retorno os funcionários do
    -- departamento código @I
    INSERT INTO @TotDeptos
    SELECT COD_DEPTO, NOME, 'D', SALARIO
    FROM TB_EMPREGADO WHERE COD_DEPTO = @I;
    -- Inserir na tabela de retorno uma linha contendo
    -- o total de salários do departamento @I
    -- Coloque no campo NOME a mensagem 'TOTAL' e no
    -- campo TIPO a letra 'T'.
    -- O campo VALOR vai armazenar o total de salários
    INSERT INTO @TotDeptos
    SELECT COD_DEPTO, 'TOTAL DO DEPTO.:', 'T',
           SUM( SALARIO )
    FROM TB_EMPREGADO WHERE COD_DEPTO = @I
    GROUP BY COD_DEPTO;
    END -- IF EXISTS
    SET @I = @I + 1;
END -- WHILE
RETURN
END -- FUNCTION

GO
-- 
SELECT * FROM FN_TOT_DEPTOS()
```

Rpare que o total do departamento será mostrado na consulta:

COD_DEPTO	NOME	TIPO	VALOR
8	RENAN CARLOS DE OLIVE...	D	3300.00
8	TOTAL DO DEPTO.:	T	5100.00
9	ALBERTO HELENA SILVA	D	3330.00
9	TOTAL DO DEPTO.:	T	3330.00
11	CARLOS ALBERTO SILVA	D	4500.00
11	TOTAL DO DEPTO.:	T	4500.00
12	JOÃO JÓSE DE SOUZA	D	800.00
12	TOTAL DO DEPTO.:	T	800.00
14	MANOEL SANTOS	D	3300.00
14	TOTAL DO DEPTO.:	T	3300.00

1.8. Campos computados com funções

Anteriormente, vimos a utilização de campos computados, ou seja, campos que o SQL realiza o cálculo automaticamente. Campos calculados melhoram o desempenho, simplificando o código, e auxiliam na normalização da tabela.

Uma implementação para este recurso é a utilização de um campo calculado com uma função que realiza a consulta e cálculo em outra tabela. Vejamos:

Na tabela **TB_PEDIDO** existe o campo **VLR_TOTAL** que é carregado a partir da soma dos itens da tabela **TB_ITENSPEIDO**. O cálculo para preencher este campo é:

```
SUM(PR_UNITARIO * QUANTIDADE * (1-DESCONTO/100))
```

Utilizando um campo calculado, não será mais necessária a realização deste cálculo. Lembrando apenas que, em alguns casos, o valor é necessário devido ao histórico da informação armazenada.

Primeiramente, é necessário criar a função que calcula o valor total do pedido:

```
CREATE FUNCTION FN_CALCULA_VALOR_TOTAL (@PEDIDO INT) RETURNS  
NUMERIC(10,2) AS  
BEGIN  
  
RETURN ( SELECT SUM(PR_UNITARIO * QUANTIDADE * (1-DESCONTO/100))  
FROM TB_ITENSPEIDO  
WHERE NUM_PEDIDO =@PEDIDO)  
  
END
```

Como a tabela já existe, utilizaremos o comando **ALTER TABLE**:

```
ALTER TABLE TB_PEDIDO  
ADD VALOR_CALC AS DBO.FN_CALCULA_VALOR_TOTAL(NUM_PEDIDO)
```

Realizando a consulta, o campo **VALOR_CALC** realiza o cálculo automaticamente:

```
SELECT NUM_PEDIDO, VLR_TOTAL, VALOR_CALC FROM TB_PEDIDO
```

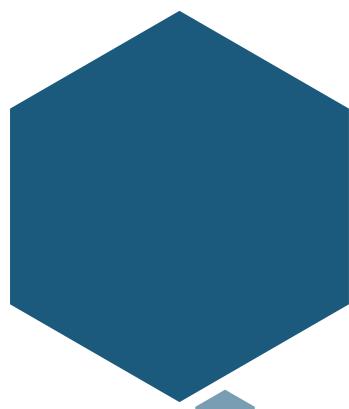
Resultado:

NUM_PEDIDO	VLR_TOTAL	VALOR_CALC
1	2.00	NULL
2	30.70	30.70
3	59.76	59.76
4	603.25	603.25
5	72.39	72.39
6	970.57	970.57
7	153.90	153.90
8	874.00	874.00
9	3163.50	3163.50
10	229.90	229.90
11	399.00	399.00

Pontos principais

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

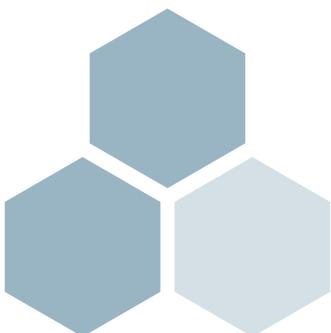
- Uma **função (function)** é um objeto que contém um bloco de comandos Transact-SQL responsável por executar um procedimento e retornar um valor ou uma série de valores;
- Tanto funções quanto stored procedures consistem em uma maneira simplificada de realizar consultas complexas. Por isso, têm muitas semelhanças. Contudo, funções não podem realizar operações que alteram dados no sistema;
- **Funções escalares** retornam um valor único, cujo tipo é definido por uma cláusula **RETURNS**, exceto os tipos: **text**, **ntext**, **image**, **cursor** e **timestamp**;
- **Funções tabulares** utilizam uma cláusula **SELECT** e retornam um conjunto de resultados em forma de tabela, ou seja, um valor do tipo **table**;
- **Funções nativas (built-in)** são aquelas já fornecidas pelo próprio SQL Server;
- Tal qual uma função nativa, uma **função definida pelo usuário** é uma rotina que aceita parâmetros, executa uma ação e retorna um valor como resultado. Elas podem ser chamadas em uma consulta, instrução ou expressão. Para criar uma função definida pelo usuário, utilizamos o comando **CREATE FUNCTION**;
- As funções **TRY_PARSE** e **TRY_CONVERT** retornam nulo, caso o valor convertido seja inválido;
- As funções **ROW_NUMBER**, **RANK**, **DENSE_RANK** e **NTILE** classificam e comparam o resultado da consulta.



Funções

Teste seus conhecimentos

Estes testes referem-se ao conteúdo da Aula 22.



1. O que são funções?

- a) São programas como procedures.
- b) São programas como procedures, porém não executam alterações de dados.
- c) São blocos de comandos que retornam e alteram dados.
- d) São programas que só retornam valores escalares.
- e) Blocos de comandos acessados pelo comando EXECUTE.

2. Qual afirmação a seguir está errada?

- a) Uma função escalar retorna um único valor dentro de uma escala de valores.
- b) Uma função retorna um valor escalar ou um tipo tabular.
- c) Os tipos: text, ntext, image, cursor e timestamp não são retornados pelas funções escalares.
- d) A função tabular IN-LINE executa a instrução SELECT diretamente.
- e) Não podemos utilizar códigos em funções tabulares.

3. Qual das funções a seguir não é determinística?

- a) LEN
- b) RIGHT
- c) GETDATE
- d) DATEPART
- e) LEFT

4. Verifique, a seguir, o código da função FN_MEDIA que calcula a média de quatro valores:

```
CREATE FUNCTION FN_MEDIA( @N1 INT, @N2 INT , @N3 INT , @N4 INT)
RETURNS FLOAT
AS BEGIN
    DECLARE @RET FLOAT;
    SET @RET = (@N1 + @N2 + @N3 + @N4)
    RETURN (@RET)
END
```

Qual será o resultado se executarmos a expressão adiante?

```
SELECT DBO.FN_MEDIA(10,4,5,1)
```

- a) 4
- b) 5
- c) 20
- d) 0 (zero), pois não é possível converter um INT em FLOAT.
- e) A sintaxe está errada.

5. Verifique, a seguir, o código da função FN_MEDIA que calcula a média de quatro valores:

```
CREATE FUNCTION FN_MEDIA( @N1 INT, @N2 INT , @N3 INT , @N4 INT)
RETURNS FLOAT
AS BEGIN
DECLARE @RET FLOAT;
SET @RET = (@N1 + @N2 + @N3 + @N4) / 4
RETURN (@RET)
END
```

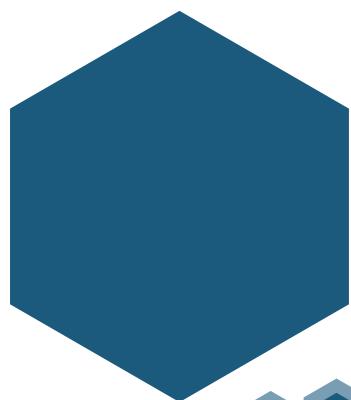
Qual será o resultado se executarmos a expressão adiante?

```
SELECT DBO.FN_MEDIA(10,4,5,NULL)
```

- a) 4
- b) 5
- c) 20
- d) 0 (zero), pois não é possível converter um INT em FLOAT.
- e) NULO.

SQL 2016 - Programação em T-SQL (online)

Gabriel M Grigorio
497.459.498-67



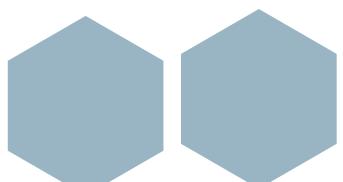
Funções



Mãos à obra!

Gabriel M. Grigorio
98-67
497. A59.

Este laboratório refere-se ao conteúdo da Aula 22.



Laboratório 1

A – Trabalhando com funções

1. Crie uma função, chamada **FN_MENOR**, que receba dois números INT como parâmetros e que retorne o menor dos dois;
2. Crie uma função que receba uma data como parâmetro e que retorne o nome do mês dessa data em português;
3. Crie uma função que receba uma data como parâmetro e que retorne a última data do mês correspondente;

! É importante lembrar que a data é um número em que cada unidade corresponde a 1 dia. Desta forma, se gerarmos a primeira data do mês seguinte, ao subtrair-se 1, teremos a última data do mês que desejamos.

4. Crie uma função que receba duas datas e que retorne a quantidade de dias úteis entre essas duas datas. Devem ser considerados os feriados da tabela **FERIADOS**;

```
CREATE TABLE FERIADOS
( DATA DATETIME, MOTIVO VARCHAR(40) )
```

5. Crie uma função tabular que receba duas datas e que retorne a quantidade total vendida e o valor total vendido de cada produto no período.

```
SELECT
    Pr.ID_PRODUTO, Pr.DESCRICAO, SUM( I.QUANTIDADE ) AS QTD_
TOTAL,
    SUM( I.QUANTIDADE * I.PR_UNITARIO ) AS VALOR_TOTAL
FROM ITENSPEDIDO I
JOIN PRODUTOS Pr ON I.ID_PRODUTO=Pr.ID_PRODUTO
JOIN PEDIDOS Pe ON I.NUM_PEDIDO=Pe.NUM_PEDIDO
WHERE
    Pe.DATA_EMISSAO BETWEEN @DT1 AND @DT2
GROUP BY Pr.ID_PRODUTO, Pr.DESCRICAO
```

B – Funções de classificação

1. Faça uma consulta que apresente o nome do cliente, o número do pedido e o valor total. Além desses campos, crie uma coluna que seja numerada automaticamente;
2. Utilizando a mesma consulta, adicione uma coluna que apresente o ranking das vendas dos clientes.

C – Campos calculados com função

1. Crie uma função que apresente o total de funcionários de um departamento;
2. Adicione uma coluna calculada na tabela **TB_DEPARTAMENTO**. Esta coluna deve utilizar a função criada no exercício anterior;
3. Realize uma consulta na tabela **TB_DEPARTAMENTO** que apresente os departamentos e a quantidade de funcionários.



Stored procedures

- ◆ Stored procedures;
- ◆ Declarando parâmetros;
- ◆ Retornando valores;
- ◆ Cursor;
- ◆ Depuração de stored procedures;
- ◆ Recompilação de stored procedures;
- ◆ Query dinâmicas;
- ◆ Tratamento de erros.

Esta Leitura Complementar refere-se ao conteúdo das Aulas 23 e 24.



1.1. Introdução

Uma coleção de comandos SQL criada para ser utilizada de forma permanente ou temporária, em uma sessão de usuário ou por todos os usuários, é chamada de **stored procedure**. Quando uma stored procedure temporária é utilizada em uma sessão de usuário, ela é chamada de procedure temporária local. Quando é utilizada por todos os usuários, a procedure temporária é chamada de global.

Podemos programar a execução de stored procedures. A seguir, citamos alguns tipos de programação que podemos aplicar às stored procedures:

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

Procedimentos criados por meio do SQL Server podem ser armazenados no próprio banco de dados como stored procedures. Assim, podemos criar aplicações que podem executar essas stored procedures, dividindo a tarefa de processamento entre a aplicação e o banco de dados.

1.2. Stored procedures

O SQL Server possui stored procedures parecidas com aquelas apresentadas por outras linguagens de programação. As stored procedures do SQL Server podem aceitar parâmetros de entrada e de saída.

A fim de indicar o sucesso ou a falha, bem como o motivo da falha, as stored procedures retornam um valor status para a aplicação (batch ou procedure).

As stored procedures não retornam valores no lugar dos seus nomes. Além disso, não podem ser utilizadas no lugar de expressões. Isso difere stored procedures de funções.

Na criação de uma stored procedure, temos o seguinte processo:

1. Os comandos de criação são analisados para correção de erros de sintaxe;
2. Caso não existam erros de sintaxe, o SQL realiza as seguintes ações:
 - Armazena o nome da stored procedure na tabela do sistema **sysobjects**;
 - Armazena o texto da criação (comandos) da procedure na tabela **syscomments** do banco de dados atual.

Assim que uma stored procedure é criada, podemos referenciá-la a objetos ainda inexistentes no banco de dados. Nesse caso, os objetos deverão existir apenas quando a procedure for executada.

1.2.1. Vantagens

A seguir, iremos expor as razões pelas quais é mais vantajoso utilizar stored procedures do que comandos SQL armazenados no **Client**:

O termo Client denomina o aplicativo que faz acesso ao banco de dados local ou remotamente.

- **Execução rápida**

A execução das stored procedures é mais rápida do que comandos SQL armazenados no **Client** porque elas já tiveram sua sintaxe previamente verificada e foram otimizadas durante sua criação. Assim, as stored procedures poderão ser acessadas a partir do cache, depois de sua primeira execução.

- **Tráfego na rede**

As stored procedures são capazes de diminuir a quantidade de dados que trafega pela rede.

- **Segurança**

As stored procedures podem ser aproveitadas como um mecanismo de segurança, restringindo o acesso às tabelas.

- **Programação modular**

As stored procedures, após serem criadas, podem ser chamadas a partir de qualquer aplicação, ou seja, elas oferecem uma programação modular.

1.2.2. Considerações

Antes de criarmos uma stored procedure, é importante estarmos cientes de alguns aspectos. Em primeiro lugar, devemos ter em mente que cada stored procedure deve ser planejada de modo a realizar apenas uma tarefa. Depois de criada, recomenda-se que a stored procedure seja testada e que tenha seus erros depurados no servidor para que, então, seja testada a partir do cliente.

Devemos lembrar, também, que é aconselhável evitar o uso do prefixo **sp_** no momento de atribuir nomes às stored procedures locais, evitando que haja confusão para diferenciá-las das stored procedures do sistema.

É recomendável que se minimize o uso de stored procedures temporárias. Dessa forma, evita-se a contenção nas tabelas de sistema que estão em **tempdb**, ocorrência que pode causar problemas no desempenho.

Na criação de uma stored procedure, não devemos esquecer que a cláusula **WITH ENCRYPTION** impedirá que a fonte da procedure permaneça legível, assim como impedirá que ela seja restaurada à sua forma original a partir dos metadados.

Vale lembrar que os seguintes comandos não podem ser utilizados dentro de stored procedures:

- **CREATE PROCEDURE;**
- **CREATE DEFAULT;**
- **CREATE RULE;**
- **CREATE TRIGGER;**
- **CREATE VIEW.**

As stored procedures podem referenciar tabelas temporárias, tabelas permanentes, views e outras stored procedures. Uma tabela temporária criada por uma stored procedure existirá enquanto a procedure estiver sendo executada.

A utilização de stored procedures merece, ainda, algumas últimas observações, que estão descritas a seguir:

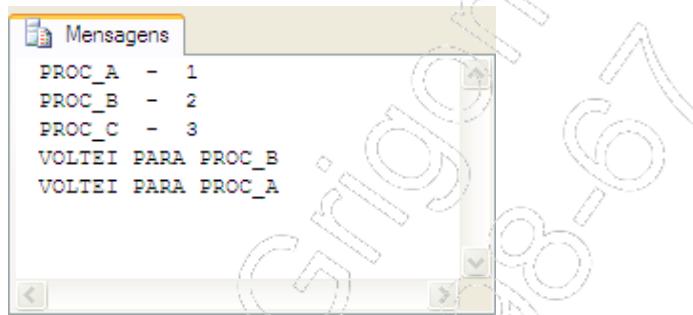
- Chamamos de aninhamento a quantidade de chamadas consecutivas que uma procedure faz para outra procedure ou para ela própria;
- O nível máximo de aninhamento suportado é 32;
- Dentro do código da procedure, podemos utilizar **@@nestlevel** para saber qual é o nível de aninhamento atual. Por exemplo:

```
CREATE DATABASE TESTE_PROC
GO
USE TESTE_PROC
GO
CREATE PROCEDURE PROC_A
AS BEGIN
PRINT 'PROC_A - ' + CAST( @@NESTLEVEL AS VARCHAR(2) );
EXEC PROC_B;
PRINT 'VOLTEI PARA PROC_A'
END
GO

CREATE PROCEDURE PROC_B
AS BEGIN
PRINT 'PROC_B - ' + CAST( @@NESTLEVEL AS VARCHAR(2) );
```

```
EXEC PROC_C;
PRINT 'VOLTEI PARA PROC_B'
END
GO
CREATE PROCEDURE PROC_C
AS BEGIN
PRINT 'PROC_C - ' + CAST( @@NESTLEVEL AS VARCHAR(2) );
END
GO
EXEC PROC_A
```

O resultado será o seguinte:



Uma stored procedure pode chamar outra stored procedure. Quando isso ocorre, a segunda procedure tem direito ao acesso de todos os objetos criados na primeira stored procedure. As tabelas temporárias estão entre esses objetos.

1.2.3. CREATE PROCEDURE

Para criarmos um stored procedure, devemos utilizar a instrução **CREATE PROCEDURE**. Vale lembrar que as únicas stored procedures que podem ser criadas fora do banco de dados atual são as temporárias. Os demais tipos de stored procedure devem todos ser criados no banco de dados atual.

Podemos comparar a criação de stored procedures com a criação de views, ou seja, antes de criarmos a stored procedure, primeiro devemos escrever e testar as instruções, a fim de verificar se os resultados obtidos são realmente os esperados.

No corpo de uma stored procedure podem ser incluídas instruções T-SQL, instruções lógicas e transações. Para chamar uma stored procedure, utilizamos a instrução **EXECUTE**.

A seguir, temos a sintaxe de **CREATE PROCEDURE**:

```
CREATE PROC[EDURE] [nome_schema.]nome_procedure  
[( {@parametro1} tipo1 [ VARYING ] [= default1] [OUTPUT]) ] {, ...}  
[WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'nome_usuario'}]  
[FOR REPLICATION]  
AS batch | EXTERNAL NAME metodo
```

Em que:

- **nome_schema**: É o nome do esquema ao qual está atribuído o proprietário da stored procedure a ser criada;
- **nome_procedure**: É o nome da procedure a ser criada;
- **@parametro1**: É um parâmetro a ser declarado na procedure. Podemos declarar mais de um. Uma procedure pode ter até 2100 parâmetros. O nome do parâmetro deve começar com o caractere @;
- **tipo1**: É o tipo de dados do parâmetro;
- **VARYING**: Aplicável somente para parâmetros cursor. Define o conjunto de resultados suportado como um parâmetro de saída;
- **default1**: É um valor padrão (constante ou NULL) para o parâmetro. A procedure poderá ser executada sem que especifiquemos um valor para o parâmetro, caso um valor padrão seja definido;
- **OUTPUT**: Indica que o parâmetro é de retorno. Ele poderá ser retornado tanto para o sistema quanto para a procedure de chamada;
- **WITH RECOMPILE**: Não pode ser utilizado com **FOR REPLICATION** e determina que o Database Engine não armazene em cache um plano para a procedure, a qual será compilada em tempo de execução;
- **WITH ENCRYPTION**: Faz com que o texto original do comando **CREATE PROCEDURE** seja convertido em um formato que não seja diretamente visível em views de catálogo do SQL Server;
- **EXECUTE AS**: Define o contexto de segurança no qual a stored procedure será executada;
- **FOR REPLICATION**: É utilizado como filtro de stored procedure e executado somente durante a replicação. Caso especifiquemos **FOR REPLICATION**, a declaração de parâmetros não pode ser feita.

1.2.4. Alterando stored procedures

Normalmente, as alterações em stored procedures são realizadas devido a pedidos de usuários ou porque as tabelas-base foram modificadas. Seja qual for o motivo da alteração, temos à disposição, para realizá-la, a instrução **ALTER PROCEDURE**, a qual modifica a stored procedure, mas conserva as atribuições de permissão e a criptografia dos dados. Ao utilizarmos **ALTER PROCEDURE**, as definições existentes da stored procedure são substituídas. Vale lembrar que tal instrução não é capaz de alterar mais de uma stored procedure por vez, sendo que stored procedures aninhadas, mesmo quando chamadas pela stored procedure alterada, não sofrem nenhum tipo de modificação.

Nas situações em que desejarmos alterar uma stored procedure que tenha sido criada com o uso de alguma opção (como **WITH ENCRYPTION**, por exemplo), tal opção não pode deixar de ser incluída na instrução **ALTER PROCEDURE** se quisermos mantê-la em funcionamento.

A sintaxe de **ALTER PROCEDURE** é a mesma de **CREATE PROCEDURE**, bastando apenas trocar o **CREATE** por **ALTER**.

1.2.5. Excluindo stored procedures

A instrução **DROP PROCEDURE** é utilizada para excluir do banco de dados as stored procedures definidas pelos usuários. Devemos lembrar que é necessário executar a stored procedure **sp_depends** antes de usar **DROP PROCEDURE** para que, assim, possamos determinar se há objetos dependentes da procedure a ser excluída.

A seguir, temos a sintaxe de **DROP PROCEDURE**:

```
DROP PROCEDURE nome_procedure
```

1.3. Declarando parâmetros

Para que seja estabelecida a comunicação entre uma stored procedure e o programa que a chama, é utilizada uma lista capaz de conter até 2100 parâmetros.

Para que uma stored procedure aceite parâmetros de entrada, basta declarar variáveis como parâmetros na instrução **CREATE PROCEDURE**. A utilização de parâmetros de entrada merece as seguintes considerações:

- Ao definirmos valores padrão para parâmetros, os usuários poderão executar stored procedures sem que haja a necessidade de especificar os valores para tais parâmetros;
- Recomenda-se validar os valores de parâmetro de entrada no início de uma stored procedure. Ao realizarmos essa medida, poderemos logo detectar valores perdidos ou inválidos. Pode ser que seja necessário verificar se o parâmetro é NULL.

1.3.1. Exemplos

O exemplo a seguir retorna o total vendido em cada um dos meses de um determinado ano:

```
USE PEDIDOS;
GO
CREATE PROCEDURE STP_TOT_VENDIDO @ANO INT
AS BEGIN
SELECT MONTH( DATA_EMISSAO ) AS MES,
       YEAR( DATA_EMISSAO ) AS ANO,
       SUM( VLR_TOTAL ) AS TOT_VENDIDO
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO) = @ANO
GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
ORDER BY MES
END

GO
--- Testando
EXEC STP_TOT_VENDIDO 2013
EXEC STP_TOT_VENDIDO 2014
```

Este outro exemplo retorna todos os itens de pedido, permitindo filtro por período, cliente e vendedor. Neste caso, os parâmetros **@CLIENTE** e **@VENDEDOR** não são obrigatórios. Se forem omitidos, assumirão '%' como default.

```
CREATE PROCEDURE STP_ITENS_PEDIDO @DT1 DATETIME,
                                    @DT2 DATETIME,
                                    @CLIENTE VARCHAR(40) = '%',
                                    @VENDEDOR VARCHAR(40) = '%'
AS BEGIN
SELECT
    I.NUM_PEDIDO, I.NUM_ITEM, I.ID_PRODUTO, I.COD_PRODUTO,
    I.QUANTIDADE, I.PR_UNITARIO, I.DESCONTO, I.DATA_ENTREGA,
```

```
PE.DATA_EMISSAO, PR.DESCRICAO, C.NOME AS CLIENTE,  
V.NOME AS VENDEDOR  
FROM TB_PEDIDO PE  
JOIN TB_CLIENTE C ON PE.CODCLI = C.CODCLI  
JOIN TB_VENDEDOR V ON PE.CODVEN = V.CODVEN  
JOIN TB_ITENS_PEDIDO I ON PE.NUM_PEDIDO = I.NUM_PEDIDO  
JOIN TB_PRODUTO PR ON I.ID_PRODUTO = PR.ID_PRODUTO  
WHERE PE.DATA_EMISSAO BETWEEN @DT1 AND @DT2 AND  
C.NOME LIKE @CLIENTE AND V.NOME LIKE @VENDEDOR  
ORDER BY I.NUM_PEDIDO  
END
```

1.3.2. Passagem de parâmetros posicional

A procedure anterior possui quatro parâmetros de entrada. A forma mais comum de passarmos os parâmetros é a posicional, ou seja, na mesma posição em que eles foram declarados dentro da procedure. Dessa forma, podemos ter:

```
-- Passando todos os parâmetros  
EXEC STP_ITENS_PEDIDO '2014.1.1', '2014.1.31', '%BRINDES%', 'LEIA'
```

Podemos também omitir o nome do vendedor, que é o último parâmetro:

```
-- Omitindo o nome do vendedor  
EXEC STP_ITENS_PEDIDO '2014.1.1', '2014.1.31', '%BRINDES%'
```

Podemos, ainda, omitir os nomes do vendedor e do cliente:

```
-- Omitindo o nome do vendedor e do cliente  
EXEC STP_ITENS_PEDIDO '2014.1.1', '2014.1.31'
```

Nesses exemplos, como os parâmetros **@CLIENTE** e **@VENDEDOR** assumem valor default, é possível omiti-los.

Se tentarmos, contudo, omitir apenas o nome do cliente, a passagem posicional não será adequada, como podemos observar a seguir:

```
EXEC STP_ITENS_PEDIDO '2013.1.1', '2014.1.31', '%LEIA%'  
-- @DT1, @DT2, @CLIENTE
```

Se utilizarmos esse tipo de passagem, estaremos associando '%LEIA%' ao terceiro parâmetro, ou seja, associaremos o nome do vendedor ao nome do cliente. Neste caso, o adequado seria utilizar a passagem de parâmetros nominal, que veremos a seguir.

1.3.3. Passagem de parâmetros nominal

Os parâmetros também podem ser passados nominalmente, conforme vemos a seguir:

```
EXEC STP_ITENS_PEDIDO @DT1 = '2014.1.1',
@DT2 = '2014.1.31',
@VENDEDOR = 'LEIA%'
```

1.4. Retornando valores

Por meio do comando **RETURN**, é possível fazer com que a procedure retorne um valor, que deve ser um número inteiro, no seu próprio nome.

O retorno de valor com **RETURN** é utilizado normalmente para sinalizar algum tipo de erro na execução ou para indicar que a procedure não conseguiu executar o que foi solicitado. A procedure a seguir retorna a última data de compra de um cliente:

```
CREATE PROCEDURE STP_ULT_DATA_COMPRA @CODCLI INT
AS BEGIN
IF NOT EXISTS( SELECT * FROM TB_PEDIDO
                WHERE CODCLI = @CODCLI )
    RETURN -1;

SELECT MAX(DATA_EMISSAO) AS ULT_DATA_COMPRA
FROM TB_PEDIDO WHERE CODCLI = @CODCLI;
END
```

Podemos testar a procedure das seguintes maneiras:

- **Teste 1**

```
DECLARE @RET INT;
EXEC @RET = STP_ULT_DATA_COMPRA 3
IF @RET < 0 PRINT 'NÃO EXISTE PEDIDO DESTE CLIENTE'
```

- **Teste 2**

```
DECLARE @RET INT;
EXEC @RET = STP_ULT_DATA_COMPRA 1
IF @RET < 0 PRINT 'NÃO EXISTE PEDIDO DESTE CLIENTE'
```

1.4.1. PRINT

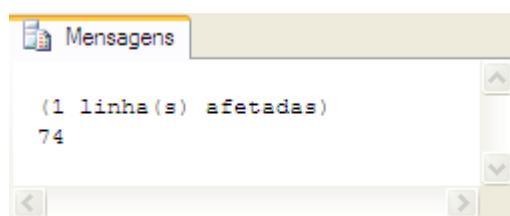
PRINT é um comando utilizado para retornar mensagens definidas pelo programador ao cliente. Esse comando adota como parâmetro uma expressão de string Unicode ou de caracteres e retorna essa string como uma mensagem para a aplicação.

Apesar de existir a possibilidade de uma aplicação capturar a mensagem retornada por um comando **PRINT**, esse procedimento é muito complexo, por isso, costumamos utilizar o **PRINT** apenas em testes e rotinas administrativas utilizadas dentro do SQL Server Management Studio.

Vejamos como utilizar **PRINT** em stored procedures a fim de retornar informações. Imagine que precisamos cadastrar um novo produto, mas suas características são quase idênticas às de outro produto já cadastrado. Seria interessante, então, termos um recurso que copiasse os dados do produto já existente para o novo e que permitisse ao usuário apenas alterar aquilo que fosse diferente:

```
CREATE PROCEDURE STP_COPIA_PRODUTO @ID_PRODUTO INT
AS BEGIN
DECLARE @ID_PRODUTO_NOVO INT;
-- Copia o registro existente para um novo registro
INSERT INTO TB_PRODUTO
( COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_CUSTO,
PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
IPI, PESO_LIQ )
SELECT COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_CUSTO,
PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
IPI, PESO_LIQ
FROM TB_PRODUTO
WHERE ID_PRODUTO = @ID_PRODUTO;
-- Descobre qual foi o ID_PRODUTO gerado
SET @ID_PRODUTO_NOVO = SCOPE_IDENTITY();
-- Retorna para a aplicação cliente o novo ID_PRODUTO gerado
PRINT @ID_PRODUTO_NOVO;
END
GO
-- Testando
EXEC STP_COPIA_PRODUTO 10
```

O retorno será visual no próprio SSMS, mas o aplicativo que chama a procedure não recebe nenhum retorno:



1.4.2. SELECT

Outra forma de retornar valor é utilizando **SELECT**, que pode ser capturado pela aplicação cliente como um conjunto de dados (**DATASET**). Os exemplos adiante demonstram o retorno de valor:

- **Exemplo 1**

```
ALTER PROCEDURE STP_COPIA_PRODUTO @ID_PRODUTO INT
AS BEGIN
DECLARE @ID_PRODUTO_NOVO INT;
-- Copia o registro existente para um novo registro
INSERT INTO TB_PRODUTO
( COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_CUSTO,
  PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
  IPI, PESO_LIQ )
SELECT COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_CUSTO,
  PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
  IPI, PESO_LIQ
FROM TB_PRODUTO
WHERE ID_PRODUTO = @ID_PRODUTO;
-- Descobre qual foi o ID_PRODUTO gerado
SET @ID_PRODUTO_NOVO = SCOPE_IDENTITY();
-- Retorna para a aplicação cliente o novo ID_PRODUTO gerado
SELECT @ID_PRODUTO_NOVO AS ID_PRODUTO_NOVO;
END
GO
-- Testando
EXEC STP_COPIA_PRODUTO 10
```

O retorno no SSMS será o seguinte:

ID_PRODUTO_NOVO
75

- **Exemplo 2**

A procedure também pode ser executada através do C#. Para isso, devemos executar o seguinte:

```
private void button1_Click(object sender, EventArgs e)
{
    string strConn = @"Provider=SQLOLEDB.1;Integrated
Security=SSPI;Persist Security Info=False;Initial
Catalog=PEDIDOS;Data Source=SOMA5\SQLEXPRESS2008;Use Procedure
for Prepare=1;Auto Translate=True;Packet Size=4096;Workstation
ID=SOMA5;Use Encryption for Data=False;Tag with column collation
```

```
when possible=False";
        OleDbConnection conn = new OleDbConnection(strConn);
        OleDbCommand cmd = new OleDbCommand("STP_COPIA_PRODUTO",
                                            conn);
        OleDbDataReader dr;
        cmd.CommandType = CommandType.StoredProcedure;
        OleDbParameter p1 = new OleDbParameter("@ID_PRODUTO",
1);
        cmd.Parameters.Add(p1);
        try
        {
            conn.Open();
            dr = cmd.ExecuteReader();
            dr.Read();
            int idProd = dr.GetInt32(0);
            MessageBox.Show("Gerado Produto ID_PRODUTO = " +
                            idProd.ToString());
            dr.Close();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
        finally
        {
            conn.Close();
        }
    }
}
```

O resultado visual na aplicação cliente será o seguinte:

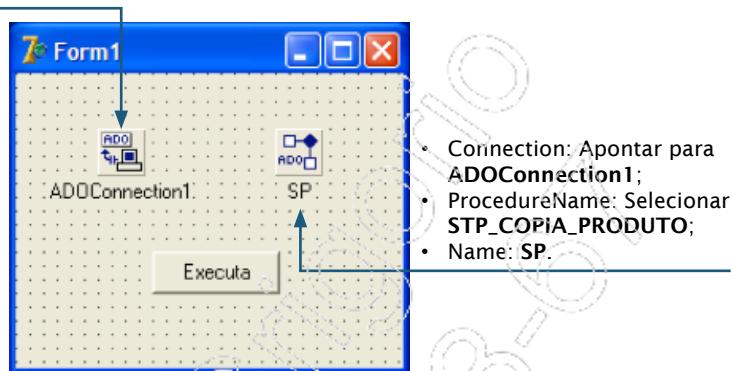


- **Exemplo 3**

A procedure também pode ser executada através do Delphi. Para isso, faça o seguinte:

1. Traga para o formulário um componente **ADOConnection** e um componente **ADOStoredProc**, e configure suas propriedades como mostra a imagem a seguir:

- Configurar a conexão na propriedade Connection;
- Propriedade LoginPrompt = False.



2. Adicione o seguinte código ao botão Executa:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
try
  SP.Parameters.ParamByName(' @ID_PRODUTO').Value := 1;
  SP.Open;
  ShowMessage('Gerado produto ID_PRODUTO = ' +
              SP.Fields[0].AsString)
except
  on e: Exception do showmessage(e.Message);
end
end;
```

O resultado visual na aplicação cliente será o seguinte:



1.4.3. Parâmetros de saída (OUTPUT)

Um tipo de passagem de parâmetro que pode ser utilizado é o que ocorre por referência. Na passagem de parâmetros por referência, é utilizada a palavra **OUTPUT**, a fim de retornar parâmetros de saída.

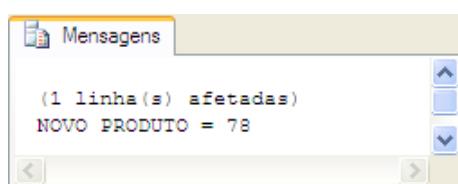
Na passagem de parâmetros por referência, tanto o comando **EXECUTE** quanto o comando **CREATE PROCEDURE** utilizam **OUTPUT**.

Quando utilizamos **OUTPUT**, podemos manter qualquer valor atribuído ao parâmetro enquanto a procedure é executada, mesmo depois que ela tenha sido finalizada.

O exemplo a seguir demonstra o uso de **OUTPUT**:

```
ALTER PROCEDURE STP_COPIA_PRODUTO @ID_PRODUTO INT,
                                    @ID_PRODUTO_NOVO INT OUTPUT
AS BEGIN
    -- Copia o registro existente para um novo registro
    INSERT INTO TB_PRODUTO
    ( COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_CUSTO,
      PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
      IPI, PESO_LIQ )
    SELECT COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_CUSTO,
           PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
           IPI, PESO_LIQ
    FROM TB_PRODUTO
    WHERE ID_PRODUTO = @ID_PRODUTO;
    -- Descobre qual foi o ID_PRODUTO gerado
    SET @ID_PRODUTO_NOVO = SCOPE_IDENTITY();
END
GO
-- Testando
DECLARE @IDPROD INT;
EXEC STP_COPIA_PRODUTO 10, @IDPROD OUTPUT;
PRINT 'NOVO PRODUTO = ' + CAST(@IDPROD AS VARCHAR(5));
```

O retorno visual dentro do SSMS será o seguinte:



Normalmente, quando executamos uma procedure a partir de uma linguagem de programação (C#, Delphi, VB, Java etc.), é mais simples trabalharmos com retornos gerados por **SELECT** do que com parâmetros de **OUTPUT**.

1.5. Cursor

Cursor é o resultado de uma consulta para o processamento individual de cada linha. Esse mecanismo permite um aumento considerável de possibilidades no processamento de informações.

Vejamos, a seguir, suas características:

- Os cursores podem ser locais ou globais;
- Na declaração do cursor é realizada a associação com a consulta;
- Para realizar a movimentação do ponteiro do cursor, utilizamos o **FETCH**;
- Para testarmos a existência de valores no cursor, utilizamos a variável **@@FETCH_STATUS** com valor igual a 0;
- Os cursores podem ser:
 - **READ ONLY**: Somente leitura;
 - **FORWARD_ONLY**: O cursor pode ser rolado apenas da primeira até a última linha;
 - **STATIC**: Cursor que recebe uma cópia temporária dos dados e que não reflete as alterações da tabela;
 - **KEYSET**: Consegue avaliar se os registros são modificados nas tabelas e retorna um valor -2 para a variável **@@FETCH_STATUS**;
 - **DYNAMIC**: O cursor reflete as alterações das linhas dos dados originais.

! É importante lembrar que cursores são lentos.

No exemplo a seguir, será utilizado um cursor para buscar o nome do supervisor e a quantidade de subordinados de cada um:

```
--Declara as variáveis de apoio
DECLARE @COD_SUP INT, @SUPERVISOR VARCHAR(35), @QTD INT

--Declara o cursor selecionando os supervisores
DECLARE CURSOR_SUPERVISOR CURSOR FORWARD_ONLY FOR
SELECT DISTINCT COD_SUPERVISOR FROM TB_EMPREGADO WHERE COD_
SUPERVISOR IS NOT NULL

-- Abre o Cursor
OPEN CURSOR_SUPERVISOR
```

```
-- Movimenta o Cursor para a 1ª linha
FETCH NEXT FROM CURSOR_SUPERVISOR INTO @COD_SUP;

-- Enquanto o cursor possuir linhas, a variável @@FETCH_STATUS
estará com valor 0
WHILE @@FETCH_STATUS = 0
BEGIN
    -- Busca o nome do Supervisor
    SELECT      @SUPERVISOR = NOME      FROM TB_EMPREGADO
    WHERE CODFUN = @COD_SUP
    -- Busca a quantidade de subordinados do supervisor
    SELECT      @QTD = COUNT(*)      FROM TB_EMPREGADO
    WHERE COD_SUPERVISOR = @COD_SUP
    AND CODFUN <> @COD_SUP
    -- Apresenta a informação
    PRINT @SUPERVISOR + ' - ' + CAST(@QTD AS VARCHAR(3)) + '
Subordinados'

    -- Move o cursor para a próxima linha
    FETCH NEXT FROM CURSOR_SUPERVISOR INTO @COD_SUP;
END
--Fecha o cursor
CLOSE   CURSOR_SUPERVISOR
-- Remove da memória
DEALLOCATE CURSOR_SUPERVISOR
```

1.6. Depurando stored procedures

Por meio do SQL Server Management Studio, podemos depurar uma stored procedure. Essa depuração é realizada a fim de detectar erros de lógica na procedure.

- **Exemplo:**

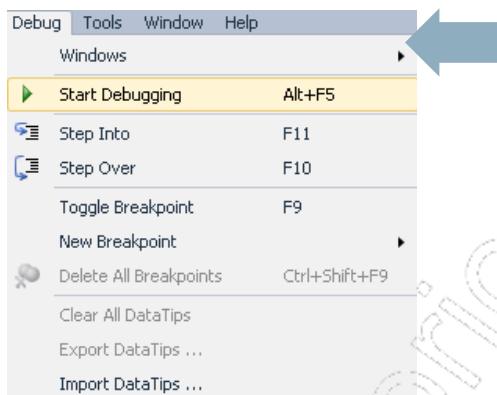
Para este exemplo, vamos usar uma stored procedure que utiliza uma variável do tipo **CURSOR**, que é utilizada quando precisamos processar, linha a linha, dados retornados por uma instrução **SELECT**.

A procedure exibida a seguir gera um texto a partir da tabela **TB_CLIENTE** (do banco de dados **PEDIDOS**), que contém os nomes dos clientes de Minas Gerais, exibindo três nomes por linha:

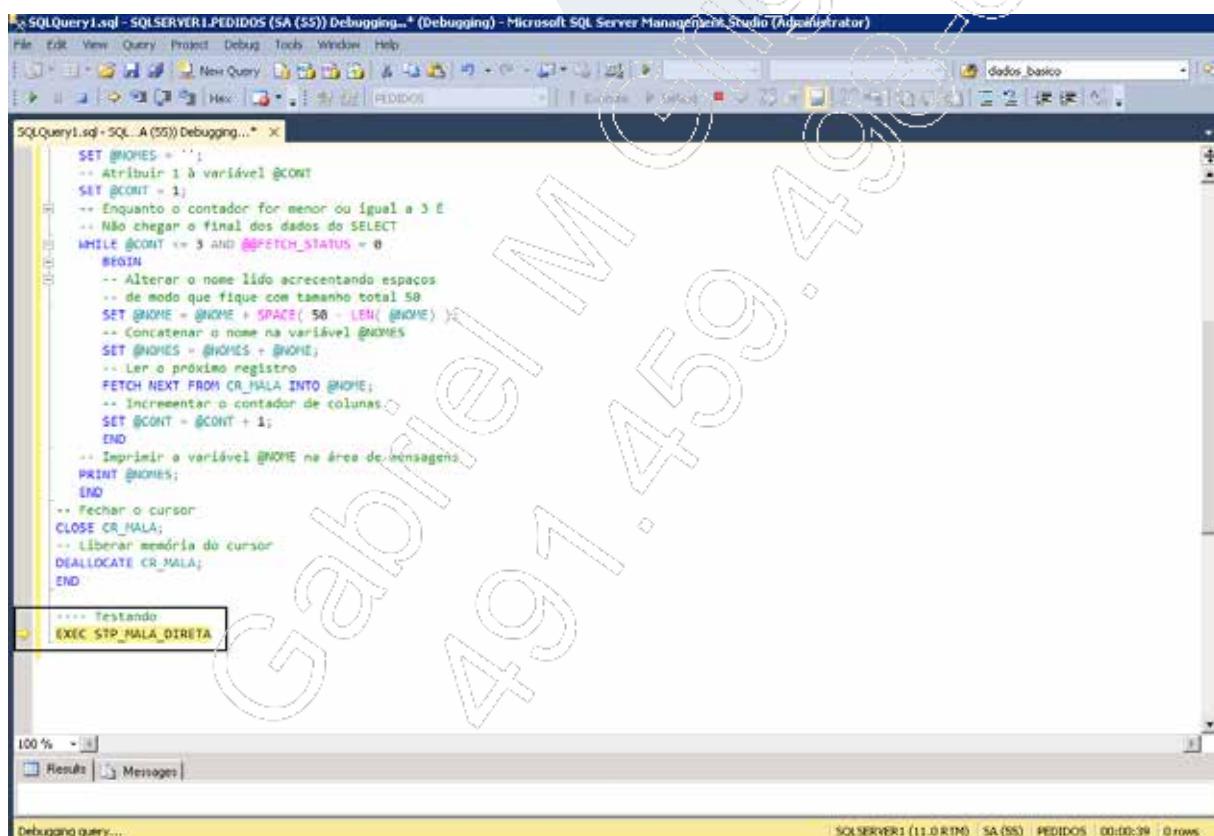
```
CREATE PROCEDURE STP_MALA_DIRETA
AS BEGIN
    -- Declarar variável do tipo CURSOR para "percorrer" um SELECT
    DECLARE CR_MALA CURSOR KEYSET
        FOR SELECT NOME FROM TB_CLIENTE
        WHERE ESTADO = 'MG';
```

```
-- Declarar uma variável para cada campo do cursor
DECLARE @NOME VARCHAR(50);
-- Contador de colunas
DECLARE @CONT INT;
-- Acumulador de nomes. Será usada para armazenar os 3
-- nomes que serão gerados para cada linha do texto
DECLARE @NOMES VARCHAR(150)
-- Abrir o cursor
OPEN CR_MALA;
-- Ler a primeira linha do cursor
FETCH FIRST FROM CR_MALA INTO @NOME;
-- Enquanto não chegar no final dos dados
WHILE @@FETCH_STATUS = 0
BEGIN
    -- "Limpar" a variável @NOMES
    SET @NOMES = '';
    -- Atribuir 1 à variável @CONT
    SET @CONT = 1;
    -- Enquanto o contador for menor ou igual a 3 e
    -- não chegar no final dos dados do SELECT
    WHILE @CONT <= 3 AND @@FETCH_STATUS = 0
    BEGIN
        -- Alterar o nome lido acrescentando espaços
        -- de modo que fique com tamanho total 50
        SET @NOME = @NOME + SPACE( 50 - LEN( @NOME) );
        -- Concatenar o nome na variável @NOMES
        SET @NOMES = @NOMES + @NOME;
        -- Ler o próximo registro
        FETCH NEXT FROM CR_MALA INTO @NOME;
        -- Incrementar o contador de colunas
        SET @CONT = @CONT + 1;
    END
    -- Imprimir a variável @NOME na área de mensagens
    PRINT @NOMES;
END
-- Fechar o cursor
CLOSE CR_MALA;
-- Liberar memória do cursor
DEALLOCATE CR_MALA;
END
GO
---- Testando
EXEC STP_MALA_DIRETA
```

Para depurar a stored procedure, devemos selecionar o comando **EXEC STP_MALA_DIRETA**, clicar no menu **Debug** e selecionar **Start Debugging**:



Uma seta amarela aparecerá à esquerda do comando, como mostra a imagem a seguir:



Ao clicar no botão para executar passo a passo (StepThrough), a seta amarela vai se posicionar na primeira instrução dentro da procedure. Podemos pressionar o botão repetidamente e observar as variáveis de memória sendo exibidas.

Quando as variáveis possuem valor, é possível visualizá-las, conforme mostrado a seguir:

The screenshot shows the Microsoft SQL Server Management Studio interface. In the center, there is a code editor window displaying a T-SQL script named 'SQLQuery1.sql'. The script contains a cursor loop that reads from a table 'CR_MALA' and prints the value of the variable '@NOME' to the message area. The script ends with a 'CLOSE' command for the cursor.

Below the code editor, the 'Locals' window is open, showing the values of three variables:

Name	Value	Type
@NOME	BRINDES ART LTDA.	varchar
@CONT	1	int
@NOMES	BRINDES ART LTDA.	varchar

On the right side of the interface, the 'Call Stack' window is also visible, showing the execution path of the current query.

Também é possível inspecionar conteúdos de funções de sistema. Para isso, faça o seguinte:

1. Selecione a guia **Locals** (local indicado por A);

2. Digite o nome da função a ser consultada (local indicado por B).

The screenshot shows the Microsoft SQL Server Management Studio interface. In the center, there is a code editor window titled "SQLSERVER1.[PEDIDOS].[dbo].[STP_MALA_DIRETA] (Debugging) - Microsoft SQL Server Management Studio (Administrator)". The code listed is:

```

BEGIN
    -- Alterar o nome lido acrescentando espaços
    -- de modo que fique com tamanho total 50
    SET @NOME = @NOME + SPACE( 50 - LEN( @NOME ) );
    -- Concatenar o nome na variável @NOMES
    SET @NOMES = @NOMES + @NOME;
    -- Ler o próximo registro
    FETCH NEXT FROM CR_MALA INTO @NOME;
    -- Incrementar o contador de colunas
    SET @CONT = @CONT + 1;
END
-- Imprimir a variável @NOME na área de mensagens
PRINT @NOMES;
END
-- Fechar o cursor

```

Below the code editor are two debugger panes. The "Locals" pane (labeled A) shows variable values:

Name	Value	Type
@NOME	RINDES ART LTDA.	varchar
@CONT	1	int
@NOMES	BRINDES ART LTDA.	varchar

The "Call Stack" pane (labeled B) shows the execution path:

Name	Language
STP_MALA_DIRETA(SQLSERVER1.PEDIDOS)	Transact-SQL
SQLQuery1.sql0 Line 49	Transact-SQL

1.6.1. Parâmetros tabulares (table-valued)

Para declarar parâmetros do tipo **table-valued**, devemos usar os tipos de tabela definidos pelos usuários. A finalidade desses parâmetros é enviar linhas de dados para uma rotina ou instrução, o que inclui, portanto, stored procedures e funções. Uma das características dos parâmetros tabulares é que, quando os utilizamos para enviar tais linhas de dados, não é preciso criar uma tabela temporária ou vários parâmetros.

O exemplo a seguir demonstra como declarar parâmetros tabulares:

1. Crie um datatype tabular:

```

CREATE TYPE TypeTabCargos AS TABLE
(
    CARGO          VARCHAR(40),
    SALARIO_INIC NUMERIC(10,2)
)

```

2. Crie uma procedure para inserir dados em **TB_CARGO**:

```
CREATE PROCEDURE STP_INSERE_CARGOS( @DADOS TypeTabCargos READONLY)
AS BEGIN
    INSERT INTO TB_CARGO (CARGO, SALARIO_INIC)
    SELECT CARGO, SALARIO_INIC FROM @DADOS;
END
```

3. Selecione e execute o seguinte bloco:

```
DECLARE @DADOS_CARGO TypeTabCargos;
INSERT INTO @DADOS_CARGO VALUES ('TESTANDO 1', 500);
INSERT INTO @DADOS_CARGO VALUES ('TESTANDO 2', 600);
INSERT INTO @DADOS_CARGO VALUES ('TESTANDO 3', 700);
INSERT INTO @DADOS_CARGO VALUES ('TESTANDO 4', 800);
EXEC STP_INSERE_CARGOS @DADOS_CARGO;
```

Dessa forma, se consultarmos a tabela **TB_CARGO**, veremos os novos registros:

```
SELECT * FROM TB_CARGO
```

O resultado da consulta é este:

	COD_CARGO	CARGO	SALARIO_INIC
16	16	CAPTURADOR DE RELENFLEX	500
17	17	ARREMESSADOR DE GARBU...	2100
18	18	TESTANDO 1	500
19	19	TESTANDO 2	600
20	20	TESTANDO 3	700
21	21	TESTANDO 4	800

1.6.2. Boas práticas

Em alguns casos, a customização de uma procedure é importante para garantir a execução e transferência das informações corretamente.

Vejamos, a seguir, algumas boas práticas para melhorar o desenvolvimento de Procedures:

- **NOCOUNT**: A instrução **SET NOCOUNT ON** bloqueia o envio da quantidade de linhas afetadas por uma instrução de **INSERT**, **UPDATE** e **DELETE**. Algumas aplicações entendem que esse retorno é um erro desconhecido;

- Exemplo:

Para aumentar em 20% o preço de venda de todos os produtos, utilizamos o seguinte comando:

```
UPDATE TB_PRODUTO SET PRECO_VENDA = PRECO_VENDA * 1.2
```

O resultado apresenta as quantidade de linhas afetadas:

```
Messages  
(69 row(s) affected)
```

Utilizando o **SET NOCOUNT ON**, essa mensagem é descartada:

```
SET NOCOUNT ON  
  
UPDATE TB_PRODUTO SET PRECO_VENDA = PRECO_VENDA * 1.2
```

Resultado:

```
Messages  
Command(s) completed successfully
```

- Sempre retorne as informações através de um **SELECT**. Assim, a aplicação pode capturar o conjunto de dados;

Para retornar a quantidade de linha de uma transação, utilize **@@ROWCOUNT**:

```
UPDATE TB_PRODUTO SET PRECO_VENDA = PRECO_VENDA * 1.2  
  
SELECT @@ROWCOUNT AS QTD
```

Resultado:

```
Results Messages  
QTD  
1 69
```

- Faça o tratamento de erros adequado, gerenciando a execução e passando sempre um retorno que a aplicação possa utilizar;
- Evite dar permissão diretamente nas tabelas. Desenvolva as procedures, funções e views.

1.7. Recompilando stored procedures

Algumas vezes, o SQL Server julga vantajosa a recompilação de stored procedures e triggers e, então, realiza-a automaticamente. Quando isso ocorre, apenas a instrução que causou a recompilação é compilada, e não a procedure inteira.

Vejamos algumas situações que ocasionam a recompilação automática da procedure:

- Alteração das estatísticas para um índice ou tabela que a procedure referenciou;
- Alteração da versão do schema;
- Diferença entre o ambiente em que a procedure é executada e o ambiente em que ela foi compilada.

No entanto, podemos provocar a recompilação de uma stored procedure por meio da stored procedure de sistema **sp_recompile**, ou por meio da inclusão da opção **WITH RECOMPILE** à stored procedure.

1.8. Query dinâmicas

Uma **query dinâmica** é um bloco de comandos TSQL que está dentro de uma string. Esses comandos são executados através do **EXEC ()** ou da procedure **SP_EXECUTESQL**.

Sua vantagem está na construção de uma query pontual com campos e parâmetros distintos. Já a desvantagem é que, ao executar esse comando, o SQL não guarda o plano de execução da consulta e não gera estatísticas para melhoria de performance.

Vejamos alguns exemplos:

- Executando um comando diretamente pelo comando **EXEC**:

```
EXEC('SELECT * FROM TB_PEDIDO')
```

- Utilizando uma variável:

```
DECLARE @SQL VARCHAR(300)  
  
SET @SQL = 'SELECT * FROM TB_PEDIDO'  
EXEC( @SQL )
```

- Compondo um comando com variáveis:

```
DECLARE @SQL VARCHAR(300) , @CODCLI INT  
  
SET @CODCLI = 5  
  
SET @SQL = 'SELECT * FROM TB_PEDIDO WHERE CODCLI=' + CAST(@CODCLI AS  
VARCHAR(5))  
EXEC( @SQL )
```

1.9. Tratamento de erros

Ao executar batches e stored procedures remotas para o cliente, a partir de uma instância local do SQL Server, podem ocorrer erros. Esses erros podem interromper a execução de uma instrução ou mesmo de um batch ou stored procedure. Para tratar eventuais erros, o SQL Server disponibiliza alguns recursos, que serão abordados nos subtópicos a seguir.

1.9.1. Severidade de um erro

Os erros gerados no Mecanismo de Banco de Dados do SQL Server possuem diferentes níveis de severidade, que têm por função indicar qual o tipo de problema que gerou o erro. Na tabela a seguir, estão relacionados os níveis de severidade dos erros gerados no SQL:

Nível de Severidade	Descrição
0 a 9	Mensagens com informação de status ou que reportam erros não severos. Não são gerados erros de sistema para esse tipo de severidade.
10	Mensagens com informação de status ou que reportam erros não severos. A severidade 10 é convertida em 0 antes que o Mecanismo de Banco de Dados retorne as informações de erro ao aplicativo.
11 a 16	Estes níveis indicam erros que podem ser corrigidos pelo usuário.
11	Indica que não existe determinado objeto ou entidade.
12	Especial para consultas que não usam bloqueios.
13	Indica erros de deadlock em uma transação.
14	Indica erros de segurança.

Nível de Severidade	Descrição
15	Indica erros de sintaxe em um comando.
16	Indica erros gerais que podem ser corrigidos pelo próprio usuário.
17 a 19	Estes níveis apontam erros de software que não podem ser corrigidos pelo próprio usuário.
17	Ocorre quando o SQL fica sem recursos, como memória ou espaço em disco, ou excede limites definidos pelo administrador do sistema.
18	Indica problema no Mecanismo de Banco de Dados. A instrução é concluída e a conexão com a instância do Mecanismo de Banco de Dados é mantida. Deve-se informar o administrador do sistema.
19	Ocorre quando um limite não configurável do Mecanismo de Banco de Dados é excedido e um processo em um batch é interrompido. Mensagens de erros com nível de severidade igual ou maior que 19 causam a interrupção do batch atual e são gravadas no log de erros. Deve-se informar o administrador do sistema.
20 a 25	Indica erros fatais, nos quais a tarefa do Mecanismo de Banco de Dados que executa uma instrução ou batch é interrompida, após gravar as informações sobre o que gerou o erro. Mensagens de erro nestes níveis de severidade podem afetar todos os processos de acesso aos dados em um banco de dados, podendo indicar ainda que um objeto ou banco de dados está danificado.
20	Indica que ocorreu um problema em uma instrução, afetando apenas a tarefa atual. É improvável que o banco de dados tenha sido danificado.
21	Indica que ocorreu um problema que afeta todas as tarefas no banco de dados. É improvável que o banco de dados tenha sido danificado.
22	Indica uma tabela ou índice danificado por problema de software ou hardware.
23	Indica que um problema de software ou hardware põe em risco a integridade do banco de dados inteiro.
24	Indica uma falha de mídia. Pode ser necessária a restauração do banco de dados.

1.9.2. @@ERROR

A função de **@@ERROR** é retornar um número referente a um erro ocorrido na última instrução Transact-SQL a ser executada. Caso não seja encontrado nenhum erro, o valor retornado será **0**. Se o erro encontrado for um dos erros definidos na view de catálogos **sys.messages**, o valor de **@@ERROR** será equivalente ao valor do erro da coluna **sys.messages.message_id**.

Como o valor de **@@ERROR** é excluído e reiniciado a cada instrução executada, ele deve ser verificado imediatamente ou, então, deve ser salvo em uma variável local para ser verificado posteriormente.

1.9.3. TRY...CATCH

Com o objetivo de manipular erros de maneira estruturada, a construção **TRY...CATCH** foi introduzida pelo Mecanismo de Banco de Dados do SQL Server 2005.

Tal construção consiste em um bloco **TRY**, cuja função é conter as transações que serão executadas e que podem eventualmente gerar erros, e um bloco **CATCH**, que contém um código que deve ser executado caso aconteça um erro no bloco **TRY**.

A sintaxe da construção **TRY...CATCH** é a seguinte:

```
-- Inicia bloco de comandos "protegidos de erro"  
BEGIN TRY  
    { comando_sql | bloco_comando}  
END TRY  
  
-- Inicia bloco de comandos de tratamento de erro  
BEGIN CATCH  
    [ { comando_sql | bloco_comando } ]  
END CATCH  
[ ; ]
```

Em que:

- **comando_sql**: É qualquer comando Transact-SQL;
- **bloco_comando**: Representa um grupo de comandos Transact-SQL, em um bloco ou batch.

Diante de um erro no bloco **TRY**, o primeiro comando no bloco **CATCH** associado irá assumir o controle. Porém, se o último comando em um trigger ou uma stored procedure for o comando **END CATCH**, o comando que chamou o trigger ou a stored procedure irá assumir o controle.

Para utilizar a construção **TRY...CATCH**, é importante atentarmos para as seguintes considerações:

- Qualquer erro de execução com o valor de severidade superior a 10 e que não finalize a conexão de banco de dados é capturado por **TRY** e **CATCH**;
- Qualquer comando entre os comandos **END TRY** e **BEGIN CATCH** irá provocar um erro de sintaxe. Portanto, é imprescindível que o bloco **TRY** seja imediatamente seguido por um bloco **CATCH** associado;
- Uma das limitações dos comandos **TRY** e **CATCH** é não poder transpor muitos blocos de comandos do Transact-SQL e diversos batches;
- Um aplicativo normalmente não terá os erros identificados por um bloco **CATCH**. Porém, por meio de alguns mecanismos utilizados no bloco **CATCH**, é possível retornar essas informações de erro ao aplicativo. Dentre esses mecanismos, temos o comando **RAISERROR** e conjuntos de resultado **SELECT**;
- O comando que vem logo após o comando **END CATCH** irá assumir o controle assim que o código do bloco **CATCH** tiver sido finalizado;
- Blocos **TRY** e **CATCH** podem conter comandos **TRY** e **CATCH** aninhados;
- Se houver um erro no bloco **TRY** de uma construção **TRY...CATCH** aninhada em um bloco **CATCH**, o controle será passado para o **CATCH** aninhado;
- Triggers e stored procedures podem alternativamente ter construções **TRY...CATCH** próprias, cuja função é controlar erros criados pelos triggers e stored procedures;
- Stored procedures ou triggers executados pelo código de um bloco **TRY** podem conter erros não controlados. Estes podem ser identificados por construções **TRY...CATCH**;
- Caso a stored procedure não tenha uma construção **TRY...CATCH** própria, um erro em seu escopo retornará o controle para o bloco **CATCH** ligado ao bloco **TRY** que possui o comando **EXECUTE**. Se tiver, o controle é transferido pelo erro ao bloco **CATCH** na stored procedure. Então, o controle é retornado ao comando localizado logo após o comando **EXECUTE** responsável por chamar a stored procedure, assim que o código do bloco **CATCH** tiver sido executado;
- O SQL Server possui comandos chamados **GOTO**, que podem ser utilizados para ir a um label específico localizado no mesmo bloco **TRY** ou **CATCH**. Os comandos **GOTO** também servem para sair de um bloco **TRY** ou **CATCH**;
- Em uma construção **TRY...CATCH**, tanto o bloco **TRY** quanto o **CATCH** devem, necessariamente, situar-se na mesma stored procedure, batch ou trigger;
- Em uma função definida pelo usuário, não é possível utilizar a construção **TRY...CATCH**.

Informações sobre erros que levaram à execução de um bloco **CATCH** podem ser recuperadas através de funções de sistema. Contudo, outras informações de erro não podem ser identificadas por uma construção **TRY...CATCH**, sendo elas:

- Avisos de atenção, dentre os quais estão incluídos aqueles referentes a conexões de cliente interrompidas e pedidos de interrupção de cliente;
- Mensagens informativas ou avisos que apresentam um valor de severidade igual ou inferior a **10**;
- Sessões finalizadas por meio do comando **KILL**;
- Erros que finalizam o processamento de tarefa do Mecanismo de Banco de Dados para a sessão e que apresentam um valor de severidade igual ou superior a **20**. Caso a conexão com o banco de dados não seja afetada, enquanto um erro com severidade igual ou maior que **20** ocorrer, esse erro será controlado pela construção **TRY...CATCH**.

Alguns tipos de erro, ao ocorrerem no mesmo nível de execução da construção **TRY...CATCH**, não são controlados por um bloco **CATCH**. É o que acontece, por exemplo, durante a recompilação ao nível de comando.

Outro tipo de erro não controlado por um bloco **CATCH** refere-se aos erros de compilação que evitam a execução de um batch.

Um erro em nível inferior a **TRY...CATCH** será manipulado pelo bloco **CATCH** caso aconteça no momento da compilação ou recompilação ao nível de comando, em um nível de execução inferior no interior de **TRY**.

1.9.4. Funções para tratamento de erros

Como dissemos anteriormente, a fim de obter informações sobre um erro que provocou a execução do bloco **CATCH**, dispomos de várias funções de sistema que devem ser utilizadas em qualquer parte do escopo de um bloco **CATCH**. São elas:

- **ERROR_NUMBER()**

Esta função retorna o número do erro ocorrido, um valor de tipo **int**.

- **ERROR_SEVERITY()**

Esta função retorna a severidade do erro. O valor retornado é de tipo **int**.

- **ERROR_STATE()**

Esta função retorna o número do estado de erro, um valor de tipo **int**.

- **ERROR_PROCEDURE()**

Esta função retorna o nome do trigger ou stored procedure onde aconteceu o erro. O valor retornado é do tipo **nvarchar(126)**. Ela retornará **NULL** caso o erro não aconteça dentro de um trigger ou stored procedure.

- **ERROR_LINE()**

Esta função retorna o número da linha em que ocorreu o erro. Se o erro aconteceu dentro de um trigger ou stored procedure, retorna o número da linha em uma rotina. O valor retornado é de tipo **int**.

- **ERROR_MESSAGE()**

Esta função retorna o texto completo da mensagem de erro. Valores fornecidos por quaisquer parâmetros substituíveis são incluídos no texto. Esses valores incluem nomes de objetos, extensões ou tempos. O valor retornado é de tipo **nvarchar(2048)**.



Se utilizadas fora do escopo de um bloco CATCH, as funções retornam NULL.

1.10. Trabalhando com mensagens de erro

Apresentamos, a seguir, três modos de se trabalhar com mensagens de erro, as quais podem ser enviadas ao aplicativo caso algum erro seja gerado no código.

1.10.1. SP_ADDMESSAGE

O SQL permite que mensagens sejam armazenadas no banco MASTER. Com este recurso, é possível padronizar todas as mensagens dentro do servidor.

Para armazenar uma mensagem, é necessário utilizar a procedure **SP_ADDMESSAGE**:

```
sp_addmessage [@msgnum =] msg_id,  
[@severity =] severity,  
[@msgtext =] 'msg'  
[, [@lang =] 'language']  
[, [@with_log =] 'with_log']  
[, [@replace =] 'replace']
```

Em que:

- **[@msgnum =] msg_id**: Refere-se ao número relativo à mensagem de erro. Os valores das mensagens de erro definidos pelo usuário devem ser superiores a 50000. Além disso, é preciso que a combinação entre **msg_id** e **language** seja única;
- **[@severity =] severity**: Refere-se ao nível de severidade do erro. Os administradores de sistema são os únicos capazes de adicionar mensagens que apresentem nível de severidade entre 19 e 25;
- **[@msgttext =] 'msg'**: Refere-se ao texto da mensagem. O tipo de dado utilizado neste argumento é o **varchar(255)**;
- **[, [@lang =] 'language']**: Refere-se à linguagem utilizada para escrever a mensagem. Quando este argumento for omitido, a linguagem utilizada para escrever a mensagem será o padrão da sessão;
- **[, [@with_log =] 'with_log']**: Determina se a mensagem deve ser escrita no log de aplicações do Windows NT e no Error log do SQL Server;
- **[, [@replace =] 'replace']**: Permite alterar o texto de uma mensagem de erro.

Vale destacar que podemos adicionar a mesma mensagem de erro em outro idioma utilizando a mesma numeração. Para tanto, basta que exista uma versão da mensagem em inglês. Além disso, é preciso que a severidade da mensagem também seja igual.

Vejamos os exemplos a seguir:

- Adicionando uma mensagem de erro devido à quantidade nula, severidade 16 e código 50001:

```
EXEC SP_ADDMESSAGE 50001,16, 'Proibido inserir quantidade nula.';
```

- Para consultar as mensagens, utilize a tabela de sistemas **SYSMESSAGES**:

```
SELECT * FROM SYS.messages WHERE MESSAGE_id>=50001
```

- Para excluir uma mensagem, utilize a procedure **SP_DROPMESSAGE**:

- Incluindo uma nova mensagem:

```
EXEC SP_ADDMESSAGE 50002,16, 'Proibido inserir quantidade nula.';
```

- Excluindo a mensagem:

```
EXEC SP_DROPMESSAGE 50002
```

1.10.2. RAISERROR

A função do comando **RAISERROR** é criar uma mensagem de erro que poderá ser utilizada em uma construção **TRY...CATCH** para exibir uma mensagem padronizada.

RAISERROR pode também fazer referência a uma mensagem definida pelo usuário, armazenada em **sys.messages**. Essa mensagem é retornada para o aplicativo que fez a chamada ou para o bloco **CATCH** associado. Esse retorno ocorre como se a mensagem fosse um erro do servidor.

Os erros gerados por **RAISERROR** funcionam de maneira semelhante aos erros gerados pelo código do Mecanismo de Banco de Dados e têm seus valores relatados pelas seguintes funções de sistema:

- **@@ERROR**;
- **ERROR_LINE**;
- **ERROR_MESSAGE**;
- **ERROR_NUMBER**;
- **ERROR_PROCEDURE**;
- **ERROR_SEVERITY**;
- **ERROR_STATE**.

RAISERROR também pode ser utilizado em um bloco **CATCH** para lançar novamente o erro que acionou este bloco, utilizando funções de sistema para recuperar dados sobre o erro original.

A sintaxe do comando **RAISERROR** é apresentada a seguir:

```
RAISERROR ( { msg_id | msg_str | @variavel_local }  
    { ,severidade ,estado }  
    [ ,argumento [ ,...n ] ] )  
    [ WITH opcao [ ,...n ] ]
```

Em que:

- **msg_id**: É um número de mensagem de erro e deve estar armazenado na view do catálogo **sys.messages**. O valor deve ser maior que 50000, caso seja uma exceção definida pelo usuário. Se o **msg_id** não for especificado, o número da mensagem de erro será 50000;
- **msg_str**: É uma mensagem definida pelo usuário, com formatação semelhante à função **printf**, e que pode ter no máximo 2047 caracteres. Quando é especificado, é gerada uma mensagem de erro com número maior que 50000;

- **@variavel_local:** É uma variável de qualquer tipo de dados de caractere válido e que possua uma cadeia de caracteres formatada da mesma forma que **msg_str**;
- **severidade:** É o nível de severidade associado à mensagem;
- **estado:** É um número inteiro entre 0 e 255, que pode ser utilizado para encontrar uma seção de código que está gerando erro, caso o mesmo erro ocorra em vários locais;
- **argumento:** Representa os parâmetros usados na substituição de uma variável definida em **msg_str** ou na mensagem correspondente a **msg_id**. Eles podem ser uma variável local ou os seguintes tipos de dados: **tinyint**, **smallint**, **int**, **char**, **varchar**, **nchar**, **nvarchar**, **binary** ou **varbinary**;
- **opcao:** É uma opção personalizada de erro, que aceita os seguintes valores: **LOG**, **NOWAIT** e **SETERROR**.

Visto que o **RAISERROR**, ao contrário do comando **PRINT**, é capaz de suportar a substituição de caracteres, ele pode ser utilizado como uma alternativa a este comando, com a finalidade de retornar mensagens às aplicações que as chamaram. Para que o **RAISERROR** possa retornar uma mensagem do bloco **TRY** sem ter que invocar o bloco **CATCH**, é preciso especificar um valor de severidade de 10 ou inferior. O bloco **TRY** não afeta o comando **PRINT**. Vale destacar que os argumentos substituem as especificações de conversão de forma sucessiva.

Quando o **RAISERROR** é chamado, a partir de uma stored procedure remota, com o valor de severidade inferior a 20, ocorre um erro que aborta o comando no servidor remoto. Visto que apenas os erros que abortam batches remotos são manipulados por uma construção **TRY...CATCH** no servidor local, caso o **RAISERROR** seja chamado por uma stored procedure remota, que se encontra no escopo do bloco **TRY** no servidor local, com severidade inferior a 20, o **RAISERROR** não faz com que o controle passe para o bloco **CATCH** da construção **TRY...CATCH**. Já nas situações em que a severidade é maior que 20, a execução no servidor local passa para o bloco **CATCH**.

O **RAISERROR** transfere o controle ao bloco **CATCH** associado nas situações em que ele é executado em um bloco **TRY** com severidade igual ou superior a 11. O erro retorna ao aplicativo que chamou **RAISERROR** somente quando ele é executado com severidade igual ou inferior a 10 em um bloco **TRY**, quando ele é executado fora do escopo deste bloco ou quando ele é executado com severidade igual ou superior a 20, que encerra a conexão com o banco de dados.

No exemplo a seguir, será retornada a mensagem de código 50001 criada com o **sp_addmessage**:

```
RAISERROR (50001,-1,-1, 'Erro!!!');
```

Resultado:

```
Messages
Msg 50001, Level 16, State 1, Line 30
Proibido inserir quantidade nula.
```

1.10.3. THROW

O comando **THROW**, introduzido a partir da versão SQL Server 2012, facilita a criação de mensagens de erro para serem utilizadas nas construções **TRY...CATCH**. Ao contrário do comando **RAISERROR**, o **THROW** gera somente exceções definidas pelos usuários, além de não haver um argumento para definir o nível de severidade vinculado ao erro, tendo este sempre o valor padrão (default) 16.

A sintaxe do comando **THROW** é semelhante à do **RAISERROR**. Podemos verificá-la a seguir:

```
THROW [ { msg_id | @variavel_local },
         { msg_str | @variavel_local },
         { estado | @variavel_local } ]
```

Em que:

- **msg_id**: É um número de mensagem de erro que, no caso do comando **THROW**, não precisa estar armazenado em **sys.messages**. O valor sempre deverá ser maior que 50000, já que só é possível trabalhar com erros definidos pelo usuário. Caso o **msg_id** não seja especificado, o número da mensagem de erro será 50000;
- **msg_str**: É uma mensagem definida pelo usuário que não aceita a formatação semelhante à função **PRINTF**. Quando é especificado, é gerada uma mensagem de erro com número maior que 50000;
- **estado**: É um número inteiro entre 0 e 255, que pode ser utilizado para encontrar uma seção de código que está gerando erro, caso o mesmo erro ocorra em vários locais;
- **@variavel_local**: É uma variável de qualquer tipo de dados de caractere válido e que possua uma cadeia de caracteres formatada da mesma forma que os argumentos que serão substituídos.

O comando **THROW** aceita como um de seus argumentos apenas mensagens passadas de forma ad hoc, portanto não há necessidade de inserir previamente as mensagens em **sys.messages**.

O exemplo a seguir mostra como o comando **THROW** pode ser utilizado no tratamento de erros:

```
USE PEDIDOS

-- Faz o PRECO_VENDA de produtos ficar menor que o PRECO_CUSTO
UPDATE TB_PRODUTO SET PRECO_VENDA = PRECO_CUSTO * 0.9
WHERE ID_PRODUTO = 1

-- Cria CONSTRAINT que impede que o PRECO_VENDA seja menor que
PRECO_CUSTO
-- a cláusula WITH NOCHECK é necessária, pois já existe uma linha
violando esta CONSTRAINT
ALTER TABLE TB_PRODUTO WITH NOCHECK
ADD CONSTRAINT CK_PRECO_VENDA CHECK(PRECO_VENDA >= PRECO_CUSTO)
GO

-- procedure para criar uma cópia de um produto
CREATE PROCEDURE STP_COPIA_PRODUTO @ID_PRODUTO_FONTE INT
AS BEGIN
    -- declara variável para o ID do novo produto gerado
    DECLARE @ID_PRODUTO_NOVO INT;
    BEGIN TRY
        -- se o ID_PRODUTO passado como parâmetro não existir
        IF NOT EXISTS(SELECT * FROM TB_PRODUTO
                      WHERE ID_PRODUTO = @ID_PRODUTO_FONTE)
            BEGIN
                -- gera um erro, o que provoca a execução do bloco CATCH
                THROW 60000, 'PRODUTO NÃO EXISTE',1
            END

        -- executa INSERT de SELECT na tabela TB_PRODUTO para
        -- efetuar a cópia
        INSERT INTO TB_PRODUTO ( COD_PRODUTO, DESCRICAO, COD_
UNIDADE, COD_TIPO, PRECO_CUSTO,
                           PRECO_VENDA, QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
                           IPI, PESO_LIQ)
            SELECT COD_PRODUTO, DESCRICAO, COD_UNIDADE, COD_TIPO, PRECO_
CUSTO, PRECO_VENDA,
                   QTD_ESTIMADA, QTD_REAL, QTD_MINIMA, CLAS_FISC, IPI, PESO_LIQ
            FROM TB_PRODUTO WHERE ID_PRODUTO = @ID_PRODUTO_FONTE

        -- descobre o novo ID_PRODUTO gerado
        SET @ID_PRODUTO_NOVO = SCOPE_IDENTITY()
        -- se os dados de TB_PRODUTO estivessem contidos em mais de
        uma
```

```
-- tabela, aqui viriam os outros INSERTs

-- retorna o novo ID_PRODUTO para a aplicação que executou
a procedure
    SELECT @ID_PRODUTO_NOVO AS ID_PRODUTO_NOVO, 'SUCESSO' AS MSG
END TRY
BEGIN CATCH

    -- recupera informações sobre o erro ocorrido
    -- que pode ser o erro de constraint, que já preparamos,
    -- ou pode ser o erro gerado dentro do bloco TRY
    DECLARE @ERRO VARCHAR(1000) = ERROR_MESSAGE();
    DECLARE @NUM INT = ERROR_NUMBER();
    DECLARE @STATUS INT = ERROR_STATE();
    /*
    -- mantendo o SELECT, a procedure não gera erro no
    -- aplicativo, apenas retorna uma linha com ID_PRODUTO
    -- igual a -1 e a mensagem de erro
    SELECT -1 AS ID_PRODUTO_NOVO, @ERRO AS MSG;
    */
    -- se quisermos provocar um erro no aplicativo
    -- que executa a procedure, podemos usar
    -- THROW erroCodigo, erroMsg, erroStatus
    THROW @NUM, @ERRO, @STATUS
END CATCH
END
GO

SELECT ID_PRODUTO, PRECO_VENDA, PRECO_CUSTO
FROM TB_PRODUTO
WHERE PRECO_VENDA < PRECO_CUSTO
GO

-- Verifique que, ao executar a procedure, ocorrerá
-- um ERRO de constraint
EXEC STP_COPIA_PRODUTO 1

-- ERRO gerado no bloco TRY
EXEC STP_COPIA_PRODUTO 999
```

1.10.4. Exemplo de tratamento de erros

Adiante, temos uma stored procedure que efetua a cópia de um pedido de venda (conforme visto anteriormente). Faremos com que essa stored procedure gere erro e demonstraremos, então, o uso de @@ERROR e TRY...CATCH no tratamento de erros:

```
CREATE PROCEDURE STP_COPIA_PEDIDO @NUM_PEDIDO_ANTIGO INT
AS BEGIN
DECLARE @NUM_PEDIDO_NOVO INT
-- Inserir em TB_PEDIDO
INSERT INTO TB_PEDIDO (CODCLI,CODVEN,DATA_EMISSAO,VLR_TOTAL,
SITUACAO,OBSERVACOES)
SELECT CODCLI,CODVEN,DATA_EMISSAO,VLR_TOTAL,
SITUACAO,OBSERVACOES
FROM TB_PEDIDO WHERE NUM_PEDIDO = @NUM_PEDIDO_ANTIGO
-- Descobrir o num. pedido gerado
SET @NUM_PEDIDO_NOVO = SCOPE_IDENTITY()
-- Inserir em TB_ITENSPEDIDO
INSERT INTO TB_ITENSPEDIDO( NUM_PEDIDO,NUM_ITEM,ID_PRODUTO,
COD_PRODUTO,CODCOR,QUANTIDADE,PR_UNITARIO,
DATA_ENTREGA,SITUACAO,DESCONTO)
SELECT @NUM_PEDIDO_NOVO,NUM_ITEM,ID_PRODUTO,
COD_PRODUTO,CODCOR,QUANTIDADE,PR_UNITARIO,
DATA_ENTREGA,SITUACAO,DESCONTO
FROM TB_ITENSPEDIDO WHERE NUM_PEDIDO = @NUM_PEDIDO_ANTIGO
-- Retornar com o num. pedido gerado
SELECT @NUM_PEDIDO_NOVO AS NUM_PEDIDO_NOVO
END
```

Agora, devemos fazer com que essa procedure provoque erro em alguns casos. Criaremos uma **CONSTRAINT** que impedirá o cadastramento de um item de pedido (TB_ITENSPEDIDO) com **QUANTIDADE** igual a zero (0):

```
ALTER TABLE TB_ITENSPEDIDO WITH NOCHECK
ADD CONSTRAINT UQ_ITENSPEDIDO_QUANTIDADE
CHECK( QUANTIDADE > 0 )
```

Ao tentar fazer uma cópia do pedido número 999, conforme o código seguinte, um erro será gerado:

```
EXEC STP_COPIA_PEDIDO 999
```

Teremos aqui uma violação da **CHECK CONSTRAINT**, porque esse pedido possui um item com **QUANTIDADE** igual a zero (0). Temos, ainda, um outro problema: como o erro ocorreu no **INSERT** de **TB_ITENSPEDIDO**, a procedure inseriu o registro na tabela **TB_PEDIDO**, mas não conseguiu inserir nenhum item. Temos, assim, um pedido que não possui itens.

Em uma situação dessas, em que há vários **INSERT**, **DELETE** e **UPDATE** na mesma procedure, se um dos comandos falhar, tudo deverá ser revertido.

A seguir, veremos duas maneiras de tratar esse erro:

- **Tratando o erro com @@ERROR (SQL 2000)**

```
ALTER PROCEDURE STP_COPIA_PEDIDO @NUM_PEDIDO_ANTIGO INT
AS BEGIN
DECLARE @NUM_PEDIDO_NOVO INT

-- Abrir processo de transação
BEGIN TRAN

-- Inserir em TB_PEDIDO
INSERT INTO TB_PEDIDO (CODCLI,CODVEN,DATA_EMISSAO,VLR_TOTAL,
SITUACAO,OBSERVACOES)
SELECT CODCLI,CODVEN,DATA_EMISSAO,VLR_TOTAL,
SITUACAO,OBSERVACOES
FROM TB_PEDIDO WHERE NUM_PEDIDO = @NUM_PEDIDO_ANTIGO
-- Verificar se houve erro
IF @@ERROR <> 0
BEGIN
ROLLBACK
RETURN
END

-- Descobrir o num. pedido gerado
SET @NUM_PEDIDO_NOVO = SCOPE_IDENTITY()
-- Inserir em TB_ITENSPEDIDO
INSERT INTO TB_ITENSPEDIDO( NUM_PEDIDO,NUM_ITEM,ID_PRODUTO,
COD_PRODUTO,CODCOR,QUANTIDADE,PR_UNITARIO,
DATA_ENTREGA,SITUACAO,DESCONTO)
SELECT @NUM_PEDIDO_NOVO,NUM_ITEM,ID_PRODUTO,
COD_PRODUTO,CODCOR,QUANTIDADE,PR_UNITARIO,
DATA_ENTREGA,SITUACAO,DESCONTO
FROM TB_ITENSPEDIDO WHERE NUM_PEDIDO = @NUM_PEDIDO_ANTIGO
-- Verificar se houve erro
IF @@ERROR <> 0
BEGIN
ROLLBACK
RETURN
END

-- Retornar com o num. pedido gerado
SELECT @NUM_PEDIDO_NOVO AS NUM_PEDIDO_NOVO
-- Finalizar transação gravando
COMMIT
END
```

Ao seguirmos os passos apresentados, se tentarmos efetuar a cópia do pedido 999, o erro continuará ocorrendo, mas nenhum pedido será inserido.

- **Tratando o erro com TRY...CATCH (SQL 2005 em diante)**

```
ALTER PROCEDURE STP_COPIA_PEDIDO @NUM_PEDIDO_ANTIGO INT
AS BEGIN
DECLARE @NUM_PEDIDO_NOVO INT

-- Abrir processo de transação
BEGIN TRAN
-- Abrir bloco protegido de erro
BEGIN TRY
    -- Inserir em TB_PEDIDO
    INSERT INTO TB_PEDIDO (CODCLI,CODVEN,DATA_EMISSAO,VLR_TOTAL,
    SITUACAO,OBSERVACOES)
    SELECT CODCLI,CODVEN,DATA_EMISSAO,VLR_TOTAL,
    SITUACAO,OBSERVACOES
    FROM TB_PEDIDO WHERE NUM_PEDIDO = @NUM_PEDIDO_ANTIGO

    -- Descobrir o num. pedido gerado
    SET @NUM_PEDIDO_NOVO = SCOPE_IDENTITY()
    -- Inserir em TB_ITENSPEDIDO
    INSERT INTO TB_ITENSPEDIDO( NUM_PEDIDO,NUM_ITEM,ID_PRODUTO,
        COD_PRODUTO,CODCOR,QUANTIDADE,PR_UNITARIO,
        DATA_ENTREGA,SITUACAO,DESCONTO )
    SELECT @NUM_PEDIDO_NOVO,NUM_ITEM,ID_PRODUTO,
        COD_PRODUTO,CODCOR,QUANTIDADE,PR_UNITARIO,
        DATA_ENTREGA,SITUACAO,DESCONTO
    FROM TB_ITENSPEDIDO WHERE NUM_PEDIDO = @NUM_PEDIDO_ANTIGO

    -- Retornar com o num. pedido gerado
    SELECT @NUM_PEDIDO_NOVO AS NUM_PEDIDO_NOVO,
        'SUCESSO' AS MSG_ERRO
    -- Finalizar transação gravando
    COMMIT
END TRY
BEGIN CATCH
    ROLLBACK
    SELECT -1 AS NUM_PEDIDO_NOVO,
        ERROR_MESSAGE() AS MSG_ERRO
END CATCH
END
```

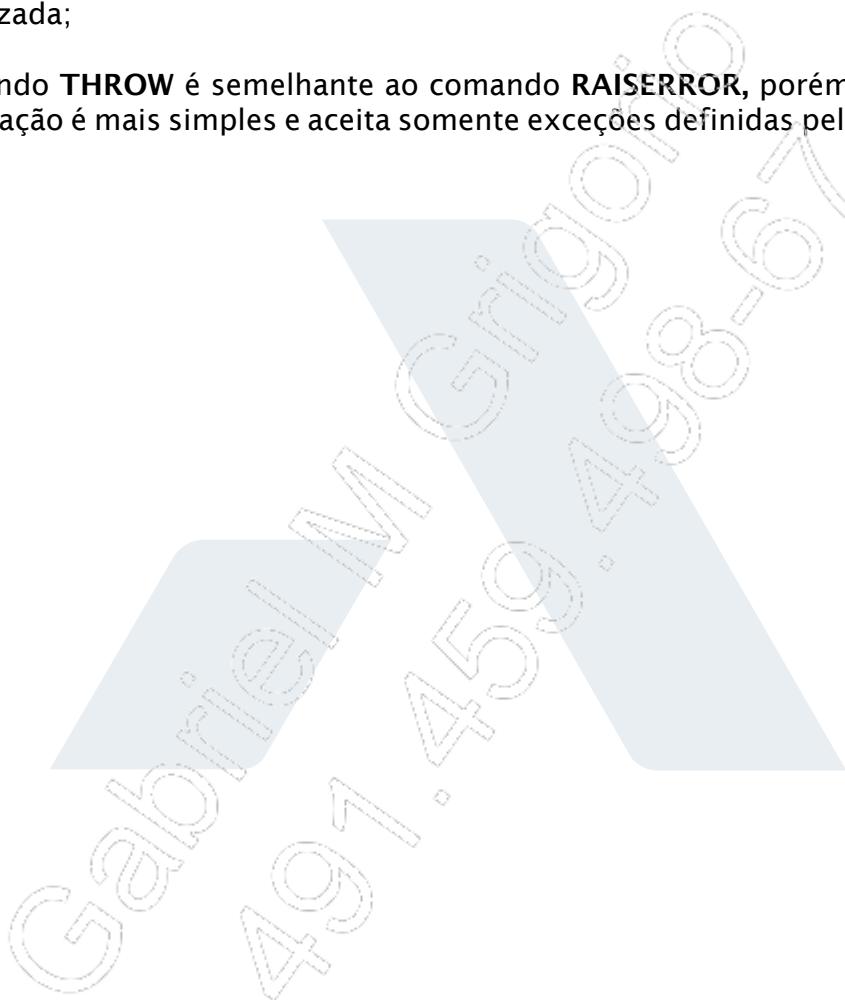
Dessa forma, ao tentarmos efetuar a cópia do pedido 999, nenhum erro será disparado para o aplicativo. Será retornado apenas o **SELECT** contido no bloco **CATCH**.

Pontos principais

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

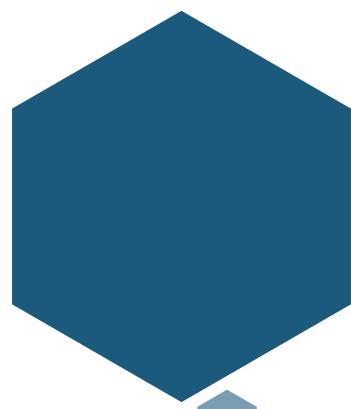
- Uma **stored procedure** é uma coleção de comandos SQL criada para ser utilizada de forma permanente ou temporária em uma sessão de usuário, ou por todos os usuários;
- As stored procedures do SQL Server podem aceitar parâmetros de entrada e de saída. Para indicar o sucesso ou falha dos comandos, ela retorna um valor para a aplicação;
- A comunicação entre uma stored procedure e o programa que a chama é feita através de uma lista capaz de conter até 2100 parâmetros. Para declarar parâmetros, utilizamos a instrução **CREATE PROCEDURE**;
- Por meio do comando **RETURN**, é possível fazer com que a procedure retorne um valor, que deve ser um número inteiro, e é utilizado normalmente para sinalizar algum tipo de erro na execução ou para indicar que a procedure não conseguiu executar o que foi solicitado;
- A fim de retornar parâmetros de saída, utiliza-se a palavra **OUTPUT**, tanto em um comando **EXECUTE** quanto em um comando **CREATE PROCEDURE**;
- A fim de detectar erros de lógica em uma stored procedure, podemos realizar a depuração, utilizando recursos oferecidos pelo SQL Server Management Studio;
- Quando há alteração das estatísticas para um índice ou tabela, ou alteração da versão do schema ou diferença entre os ambientes em que a procedure é compilada e executada, pode ocorrer recompilação automática da stored procedure. A recompilação também pode ser feita através de **sp_recompile** ou **WITH RECOMPILE**;
- Os erros gerados no Mecanismo de Banco de Dados do SQL Server possuem diferentes níveis de severidade, que têm por função indicar qual o tipo de problema que gerou o erro;
- A função de **@@ERROR** é retornar um número referente a um erro ocorrido na última instrução Transact-SQL a ser executada;

- A construção **TRY...CATCH** foi introduzida pelo Mecanismo de Banco de Dados do SQL Server 2005 com o objetivo de manipular erros de maneira estruturada;
- Para inserir uma mensagem de erro definida pelo usuário, utilizamos **SP_ADDMESSAGE**;
- A função do comando **RAISERROR** é criar uma mensagem de erro que poderá ser utilizada em uma construção **TRY...CATCH** para exibir uma mensagem padronizada;
- O comando **THROW** é semelhante ao comando **RAISERROR**, porém, sua forma de utilização é mais simples e aceita somente exceções definidas pelos usuários.



SQL 2016 - Programação em T-SQL (online)

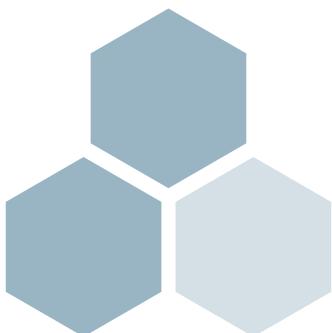
Gabriel M Grigorio
497.459.498-67



Stored procedures

Teste seus conhecimentos

Estes testes referem-se ao conteúdo das Aulas 23 e 24.



1. Qual das características a seguir não está correta com relação à utilização de procedures?

- a) Execução rápida.
- b) Segurança.
- c) Programação modular.
- d) Diminui tráfego na rede.
- e) Não é segura.

2. Qual comando não é permitido na utilização de uma procedure?

- a) SELECT
- b) CREATE TABLE
- c) CREATE RULE
- d) INSERT
- e) DELETE

3. Sobre parâmetros de procedures, qual afirmação está correta?

- a) Podemos utilizar somente parâmetros de entrada.
- b) São limitados a 210.
- c) Não existem parâmetros do tipo OUTPUT.
- d) Uma procedure pode receber parâmetros de entrada e de saída.
- e) São lentos e devem ser evitados.

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

```
BEGIN TRY  
  
    UPDATE CLIENTES SET ESTADO = 'wSP' WHERE CODCLI=13  
  
BEGIN CATCH  
    PRINT 'OCORREU UM ERRO'  
  
END CATCH
```

- a) Caso ocorra erro, será mostrada a mensagem "OCORREU UM ERRO".
- b) Gera um erro de sintaxe.
- c) Caso ocorra erro, será mostrada a mensagem "OCORREU UM ERRO" e realizado um ROLLBACK.
- d) O correto é utilizar o comando @@ERROR.
- e) Este comando não gera erro.

5. Qual afirmação está incorreta sobre cursos?

- a) A variável @@FETCH_STATUS é utilizada para verificar a existência de linhas no cursor.
- b) São utilizados para ler o resultado de um SELECT.
- c) Podem ser locais e globais.
- d) São úteis e não são lentos.
- e) O tipo FORWARD_ONLY realiza a leitura apenas da primeira até a última linha.

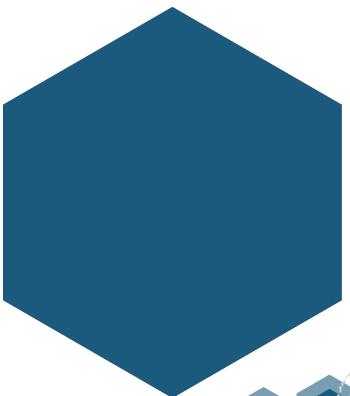
6. Verifique o comando a seguir e selecione a afirmação correta:

```
UPDATE CLIENTES SET ESTADO ='SP' WHERE CODCLI=13
IF @@ERROR <>0
    PRINT 'OCORREU UM ERRO'
```

- a) Caso ocorra erro, será mostrada a mensagem "OCORREU UM ERRO".
- b) Gera um erro de sintaxe.
- c) Caso ocorra erro, será mostrada a mensagem "OCORREU UM ERRO" e realizado um ROLLBACK.
- d) Só podemos utilizar o comando TRY...CATCH.
- e) Este comando não gera erro.

SQL 2016 - Programação em T-SQL (online)

Gabriel M Grigorio
497.459.498-67



Stored procedures



Mãos à obra!

Este laboratório refere-se ao conteúdo das Aulas 23 e 24.



Laboratório 1

A – Criando procedures

Neste exercício, criaremos procedures com diferentes funções:

1. Crie uma procedure que retorne os clientes (código, nome, valor e número do pedido), com parâmetro **ANO** e ordenado pelo nome do cliente;
2. Teste a procedure criada com os seguintes anos: **2012, 2013 e 2014**;
3. Crie uma procedure para inserir departamentos na tabela **TB_DEPARTAMENTO**;
4. Insira os departamentos **Mensageria** e **TI**;
5. Crie uma procedure para inserir tipo de produto (**TB_TIPOPRODUTO**);
6. Insira os tipos **TESTE** e **TESTE2**;
7. Crie uma procedure que exclua um tipo de produto (**TB_TIPOPRODUTO**). Antes de excluir, é necessário que seja verificado se o tipo de produto é utilizado em produtos. O parâmetro deve ser a descrição do tipo e não o código. O retorno deve ser um **OK** ou **NOK** para tipos que são utilizados por produtos;
8. Exclua o tipo de produto **TESTE**;
9. Exclua o tipo de produto **REGUA**;
10. Crie a tabela **TB_Resumo**, com os seguintes campos:

```
ID_Resumo  INT auto numerável chave primária  
Ano        INT  
MÊS       INT  
Valor      DECIMAL(10,2)
```

11. Crie uma procedure que carregue as informações da tabela de pedidos. Utilize os parâmetros **@ANO INT** para filtrar as informações. Siga os passos adiante:

- Não insira valores duplicados;
- Exclua os registros do ano antes de realizar a carga;
- Utilize transações;
- Faça o tratamento de erros com o **TRY CATCH**;
- Retorne a quantidade de registros carregados;
- Caso ocorra erro, retorne a mensagem.

12. Faça o teste carregando os seguintes anos: 2012, 2013 e 2014;
13. Faça a consulta na tabela TB_RESUMO e verifique se as informações estão corretas.

Laboratório 2

A – Criando uma stored procedure cujo objetivo é criar um campo novo em todas as tabelas do banco de dados

Para a realização deste exercício, considere as seguintes informações:

- Para saber os nomes de todas as tabelas de usuário do banco de dados em uso, execute:

```
SELECT ID, NAME FROM SYSOBJECTS WHERE XTYPE = 'U'
```

- Podemos executar um comando SQL contido em uma variável do tipo VARCHAR da seguinte forma:

```
EXEC( @COMANDO )
```

- A procedure receberá dois parâmetros:
 - **@CAMPO VARCHAR(200)**: Conterá o nome do campo;
 - **@TIPO VARCHAR(200)**: Conterá o tipo e outras características de um campo. Por exemplo:

```
EXEC STP_CRIA_CAMPO 'COD_USUARIO', 'INT NOT NULL DEFAULT 0'
```

Para montar o conteúdo da procedure, siga o roteiro adiante:

1. Crie a procedure;
2. Declare variáveis **@COMANDO VARCHAR(200)**, **@TABELA VARCHAR(200)** e **@ID INT**;
3. Declare cursor para:

```
SELECT ID, NAME FROM SYSOBJECTS WHERE XTYPE = 'U'
```

4. Abra o cursor;

5. Leia a primeira linha do cursor. Enquanto não chegar ao final dos dados, teremos:

```
WHILE @@FETCH_STATUS = 0
BEGIN
```

6. Execute, então, os seguintes passos:

- Armazene na variável **@COMANDO** a seguinte instrução:

```
'ALTER TABLE ' + @TABELA + ' ADD ' + @CAMPO + ' ' + @TIPO
```

- Execute o comando contido na variável **@COMANDO**;
- Imprima na área de mensagens o comando que foi executado;
- Leia a próxima linha da tabela;
- Finalize o loop.

7. Feche o cursor;

8. Desaloque o cursor da memória.

B – Alterando a procedure anterior para testar se o campo já existe na tabela e imprimindo-o, caso ele exista

Para testar se a tabela **PRODUTOS** tem um campo chamado **PRECO_VENDA**, devemos, primeiramente, executar a seguinte linha:

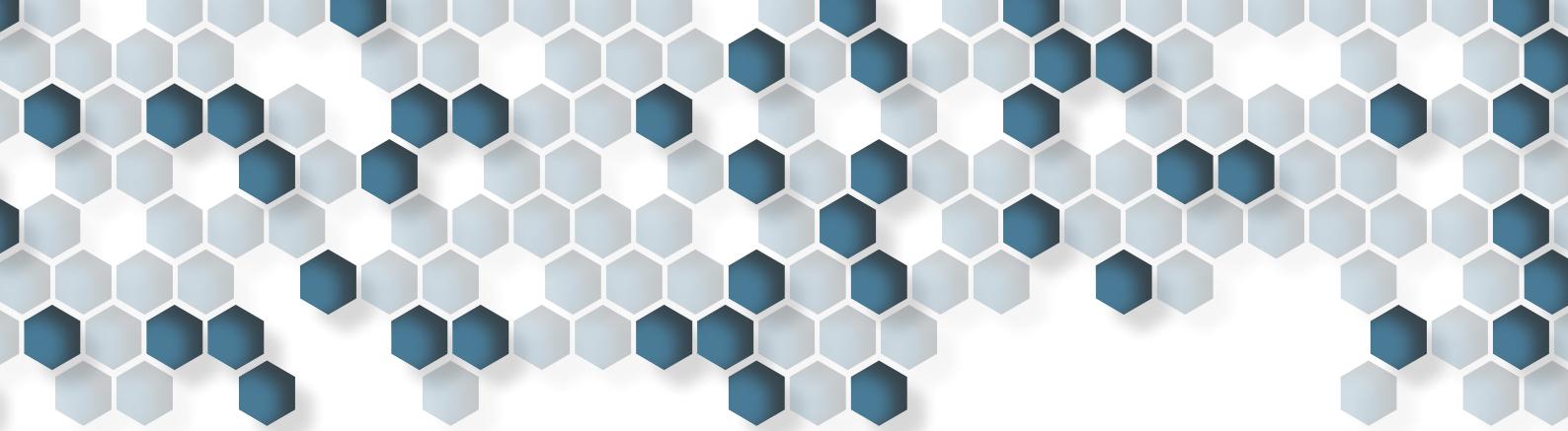
```
SELECT ID FROM SYSOBJECTS WHERE NAME = 'PRODUTOS'
```

Em que o ID da tabela **PRODUTOS** é 357576312. Assim:

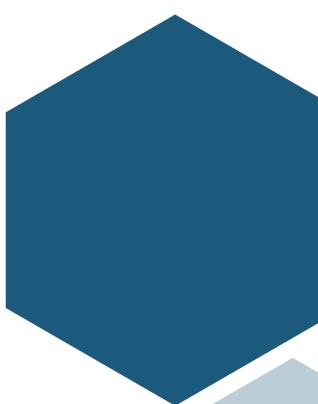
```
SELECT * FROM SYSCOLUMNS
WHERE NAME = 'PRECO_VENDA' AND ID = 357576312
```

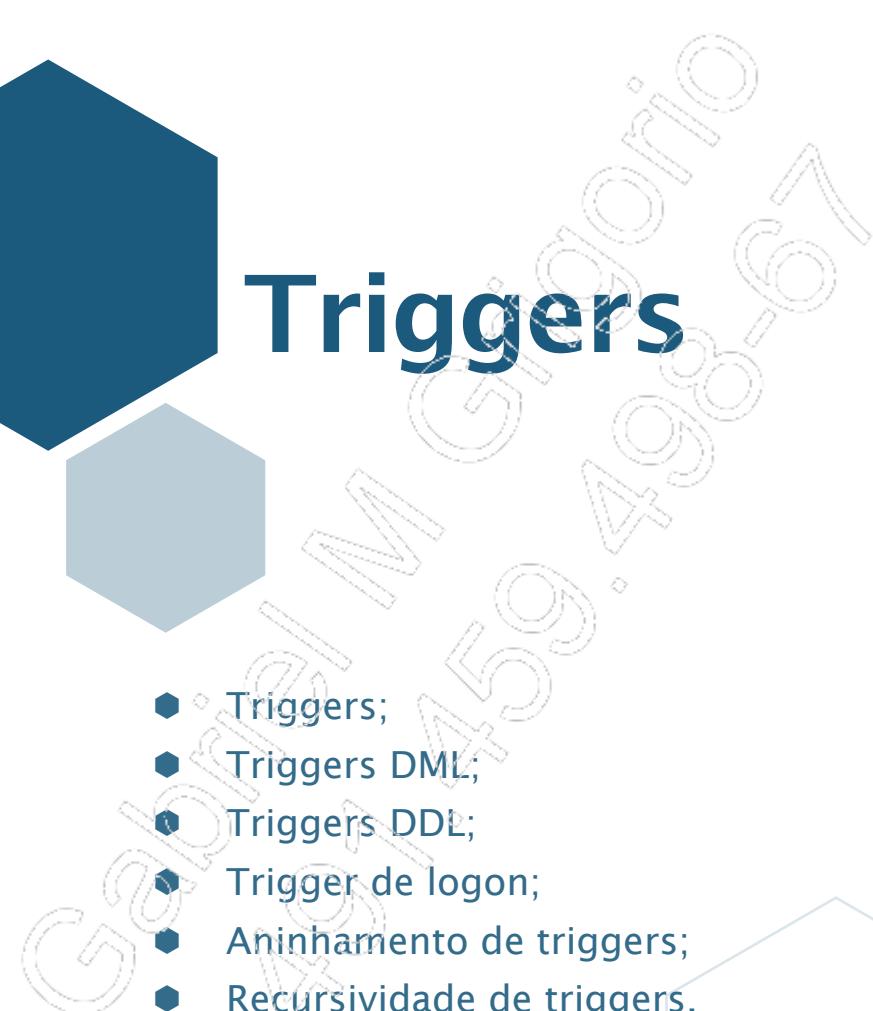
Adaptando para a procedure, podemos fazer o seguinte:

```
IF EXISTS(SELECT * FROM SYSCOLUMNS
WHERE NAME = @CAMPO AND ID = @ID)
```

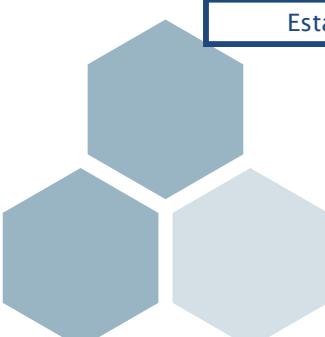


Triggers



- Triggers;
 - Triggers DML;
 - Triggers DDL;
 - Trigger de logon;
 - Aninhamento de triggers;
 - Recursividade de triggers.
- 

Esta Leitura Complementar refere-se ao conteúdo da Aula 25.



1.1. Introdução

O SQL Server oferece dois mecanismos primários para reforçar as regras de negócio e a integridade dos dados: as constraints e os triggers.

Nesta leitura, serão descritos os tipos de triggers que o SQL Server oferece e que podem ser criados, e as ações executadas por eles.

1.2. Triggers

Um trigger, também chamado de gatilho, é um tipo especial de stored procedure que é automaticamente disparado quando há um evento de linguagem, isto é, quando é realizada alguma alteração nos dados de uma tabela. A sua principal vantagem é automatizar processos.

O SQL Server inclui três tipos genéricos de triggers:

- Triggers DML (Data Manipulation Language);
- Triggers DDL (Data Definition Language);
- Triggers de logon.

Os triggers DDL podem ser utilizados com a finalidade de realizar tarefas administrativas. Sua função é disparar stored procedures para responder aos comandos DDL, os quais são, principalmente, iniciados com ALTER, DROP e CREATE.

Já os triggers DML são utilizados nas situações em que os comandos INSERT, DELETE e UPDATE realizam a alteração dos dados presentes em views ou em tabelas. Os triggers DML são classificados em triggers INSTEAD OF e triggers AFTER, estes subdivididos, ainda, em triggers de inclusão, de exclusão e de alteração.

Os triggers de logon, como o próprio nome indica, disparam stored procedures diante de eventos do tipo logon (iniciados no momento em que uma sessão de usuário é estabelecida com uma instância do SQL Server). Triggers desse tipo podem ser utilizados para gerenciar e auditar sessões de servidor.

O acionamento dos triggers ocorre de forma automática. Eles não recebem e, tampouco, devolvem parâmetros. Apesar disso, eles podem gerar erros com o comando RAISERROR. Os triggers podem ser criados a fim de que seu acionamento ocorra quando os seguintes comandos forem utilizados para alterar os dados de uma tabela: INSERT, DELETE e UPDATE.

Tanto o trigger como o comando que o acionou são tratados como sendo uma única transação. Esta pode ser desfeita em qualquer lugar de dentro do trigger, o que significa que podemos utilizar um **ROLLBACK TRANSACTION** de dentro do trigger, que ele será responsável por desfazer tudo o que foi feito pelo trigger.

O comando **INSERT** é capaz de acionar um trigger que executará seu código interno. Esse tipo de trigger, cujo acionamento ocorre no momento em que dados são incluídos na tabela, é capaz de executar internamente qualquer um dos seguintes comandos: **INSERT**, **SELECT**, **UPDATE** e **DELETE**. Os únicos comandos que não podem ser executados por triggers são:

- **CREATE** (todos);
- **DROP** (todos);
- **DISK** (todos);
- **GRANT**;
- **LOAD**;
- **REVOKE**;
- **ALTER TABLE**;
- **ALTER DATABASE**;
- **TRUNCATE TABLE**;
- **UPDATE STATISTICS**;
- **RECONFIGURE**;
- **RESTORE DATABASE**;
- **RESTORE LOG**;
- **SELECT INTO**.

! Os triggers podem ser empregados para implementar as alterações em cascata que devem ser realizadas no sistema.

1.2.1. Diferenças entre triggers e constraints

As constraints que já estão prontas podem ser adicionadas na tabela a fim de que as regras de negócio comuns sejam mantidas. Já para construir regras de negócio mais complexas, é preciso utilizar os triggers, os quais, diferentemente das constraints, são reativos. Isso significa que, utilizando os triggers, o SQL Server primeiramente executa o comando que o acionou e, depois, verifica a ocorrência de erros; caso haja um erro, o comando é desfeito.

Já as constraints são pré-ativas, ou seja, elas são verificadas pelo SQL Server a fim de assegurar que não estejam violadas antes que o comando seja executado. Caso as constraints estejam com problemas, uma mensagem de erro é enviada e a operação é abortada. O comando apenas é executado se a constraint não apresentar problemas.

1.2.2. Considerações

O SQL Server verifica primeiramente as constraints para depois verificar os triggers. Em uma tabela, podemos ter vários triggers de qualquer ação. É importante ter em mente que a criação, a alteração e a exclusão de um trigger são tarefas que podem ser realizadas apenas pelo objeto schema da tabela, sendo que essa permissão não pode ser transferida. Além disso, é preciso que o schema da tabela tenha permissão para executar todos os comandos em todas as tabelas que são afetadas pelo trigger. É essencial que as permissões sejam atribuídas de forma adequada, pois, caso contrário, a transação é desfeita.

Embora os triggers não possam ser criados em tabelas temporárias, eles podem fazer referências a elas. Triggers podem ser criados apenas no banco de dados atual, mas podem fazer referências aos objetos que se encontram fora desse banco de dados.

O comando **CREATE TRIGGER** deve ser o primeiro em um batch e sua aplicação pode ser feita em apenas uma tabela. Em um único comando **CREATE TRIGGER**, podemos definir uma ação do trigger a ser realizada para mais de uma ação do usuário. Caso o trigger seja qualificado por meio do nome do esquema trigger, da mesma forma deve ser qualificado o nome da tabela.

O comando **CREATE TRIGGER** permite adicionar triggers àqueles já existentes nas situações em que os nomes desses triggers são distintos. Este é o comportamento padrão do comando **CREATE TRIGGER** quando o nível de compatibilidade está configurado como 70. Nas situações em que os nomes dos triggers não são distintos, uma mensagem de erro é retornada pelo SQL Server.

Contudo, caso o nível de compatibilidade esteja configurado como 65 ou menos, triggers que são criados por meio do comando **CREATE TRIGGER** substituem os triggers de mesmo tipo já existentes, até mesmo nas situações em que os nomes das triggers são distintos.

Quando trabalhamos com tabelas que possuem chaves estrangeiras com uma ação **DELETE/UPDATE** em cascata, os triggers dos comandos **INSTEAD OF DELETE/UPDATE** não podem ser definidos. Dentro de um trigger, podemos definir qualquer comando **SET**, sendo que a opção selecionada tem efeito apenas durante a execução do trigger.

Os resultados obtidos após o trigger ter sido disparado são retornados à aplicação que o chamou. Para evitar que tais resultados sejam retornados, basta não incluir no trigger comandos **SELECT** que retornam resultados ou comandos que realizam a atribuição de variáveis. É preciso manipular de forma específica os triggers que possuem esses tipos de comandos. Para as situações em que se faz necessário realizar a atribuição de variáveis em um trigger, basta utilizar o comando **SET NOCOUNT** no início do trigger.

Vale destacar, no entanto, que a capacidade de retornar resultados a partir de triggers é um recurso que será excluído em uma versão futura do SQL Server. Portanto, é importante evitar a utilização do retorno de resultados a partir de triggers em novos projetos e, até mesmo, alterar os projetos que já utilizam este recurso.

O comando **TRUNCATE TABLE** não é capaz de ativar um trigger devido ao fato de não registrar a exclusão individual de linhas. Já o comando **WRITETEXT** não ativa um trigger.

1.2.3. Visualizando triggers

Podemos obter uma lista de todos os triggers disponíveis em um banco de dados. Para isso, devemos efetuar uma consulta na view de catálogo **sys.triggers**, conforme o seguinte código:

```
SELECT name  
FROM sys.triggers
```

A visualização dos triggers de um banco de dados também pode ser feita a partir do **Object Explorer**, no SQL Server Management Studio.

Para retornar a definição de um trigger, podemos efetuar uma consulta na view de catálogo **sys.sql_modules**.

1.2.4. Alterando triggers

Para alterar o código interno de um trigger, utiliza-se a instrução **ALTER TRIGGER**, que tem a mesma sintaxe do **CREATE TRIGGER**.

Quando mudamos o nome de um objeto que é referenciado por um trigger DDL, há a necessidade de mudar o trigger a fim de refletir o novo nome. Portanto, antes de renomear um objeto, devemos exibir suas dependências, para que possamos determinar quais triggers serão afetados pela alteração.

1.2.5. Desabilitando e excluindo triggers

Quando um trigger não é mais necessário, podemos excluí-lo definitivamente ou apenas desabilitá-lo. Quando desabilitamos um trigger, ele continua existindo no banco de dados, entretanto, ele não será executado quando a instrução T-SQL na qual foi programado for executada. Os triggers desabilitados podem ser habilitados posteriormente, conforme a necessidade. Quaisquer objetos ou dados relacionados não são afetados.

1.2.5.1. DISABLE TRIGGER

Para desabilitar um trigger DDL, utilizamos a instrução **DISABLE TRIGGER**, cuja sintaxe é a seguinte:

```
DISABLE TRIGGER { [ nome_schema . ] nome_trigger [ ,...n ] | ALL }
ON { nome_objeto | DATABASE | ALL SERVER } [ ; ]
```

Em que:

- **nome_schema**: Trata-se do nome do schema ao qual a trigger pertence;
- **nome_trigger**: É o nome do trigger a ser desabilitado;
- **ALL**: Determina a desativação de todos os triggers definidos no escopo da cláusula **ON**;
- **nome_objeto**: Representa o nome da tabela ou view na qual o trigger a ser desabilitado foi criado para execução;
- **DATABASE**: Indica, para triggers DDL, que o trigger a ser desabilitado foi criado ou modificado para ser executado com o escopo de banco de dados;
- **ALL SERVER**: Indica, para triggers DDL e até triggers de logon, que o trigger a ser desabilitado foi criado ou modificado para ser executado com o escopo de servidor.

1.2.5.2. ENABLE TRIGGER

Para habilitar um trigger DDL, utilizamos a instrução **ENABLE TRIGGER**, cuja sintaxe é exibida a seguir:

```
ENABLE TRIGGER { [ nome_schema . ] nome_trigger [ ,...n ] | ALL }
ON { nome_objeto | DATABASE | ALL SERVER } [ ; ]
```

Em que:

- **nome_schema**: Trata-se do nome do schema ao qual a trigger pertence;
- **nome_trigger**: É o nome do trigger a ser habilitado;
- **ALL**: Determina a habilitação de todos os triggers definidos no escopo da cláusula **ON**;
- **nome_objeto**: Representa o nome da tabela ou view na qual o trigger a ser habilitado foi criado para execução;
- **DATABASE**: Indica, para triggers DDL, que o trigger a ser habilitado foi criado ou modificado para ser executado com o escopo de banco de dados;
- **ALL SERVER**: Indica, para triggers DDL e até triggers de logon, que o trigger a ser habilitado foi criado ou modificado para ser executado com o escopo de servidor.

1.2.5.3. DROP TRIGGER

Para excluir triggers de um banco de dados, utiliza-se a instrução **DROP TRIGGER**. Vejamos as três sintaxes utilizadas para ela, de acordo com o tipo de trigger a ser excluído:

- **Triggers DML**

```
DROP TRIGGER [ nome_schema.]nome_trigger [ ,...n ] [ ; ]
```

- **Triggers DDL**

```
DROP TRIGGER nome_trigger [ ,...n ]
ON { DATABASE | ALL SERVER }
[ ; ]
```

- **Triggers de logon**

```
DROP TRIGGER nome_trigger [ ,...n ]
ON ALL SERVER
```

A seguir, temos a descrição dos argumentos utilizados nas três sintaxes de **DROP TRIGGER**:

- **nome_schema**: Representa o nome do schema ao qual pertence o trigger DML;
- **nome_trigger**: É o nome do trigger a ser excluído;
- **DATABASE**: Determina que o escopo do trigger DDL aplica-se ao banco de dados atual. Sua utilização é obrigatória caso tenha sido especificada na criação ou modificação do trigger;
- **ALL SERVER**: Determina que o escopo do trigger DDL aplica-se ao servidor atual. Sua utilização é obrigatória caso tenha sido especificada na criação ou modificação do trigger. Triggers de logon também aceitam **ALL SERVER**.

1.3. Triggers DML

Um **trigger DML** é uma ação programada para executar quando um evento DML ocorre em um servidor de banco de dados. As instruções **UPDATE**, **INSERT** ou **DELETE** executadas em uma tabela ou view são exemplos de eventos DML.

Vejamos quais são as características principais dos triggers DML:

- Podem afetar outras tabelas e pode incluir instruções Transact-SQL complexas;
- Podem prevenir operações incorretas ou mal intencionadas de **INSERT**, **UPDATE** ou **DELETE** e fazer com que restrições mais complexas do que aquelas definidas com constraints **CHECK** sejam aplicadas;
- Podem referenciar colunas em outras tabelas, ao contrário de constraints **CHECK**;
- Podem avaliar o estado de uma tabela antes e depois de uma modificação dos dados e realizar ações com base nas diferenças entre o estado anterior e posterior à modificação dos dados;
- Como resposta a uma mesma instrução de modificação de dados, os triggers DML de mesmo tipo (**INSERT**, **UPDATE** ou **DELETE**) localizados em uma mesma tabela permitem que diferentes ações sejam realizadas.

Os triggers DML podem ser divididos nas seguintes categorias:

- **Triggers AFTER**: Especificados somente em tabelas, os triggers **AFTER** são executados após a conclusão de ações realizadas por uma instrução **INSERT** (inserção), **UPDATE** (alteração) ou **DELETE** (exclusão). Especificar **AFTER** é o mesmo que especificar **FOR**;

- **Triggers INSTEAD OF:** Também conhecidos como **BEFORE**, são executados no lugar das ações de trigger comuns. Os triggers **INSTEAD OF** podem ser definidos em views com uma ou mais tabelas base, nas quais podem ampliar os tipos de alteração suportados por uma view;
- **Trigger CLR:** Pode ser tanto um trigger **AFTER** quanto um trigger **INSTEAD OF**, ou mesmo um trigger DDL. Ele não executa uma stored procedure da Transact-SQL. Em vez disso, executa um ou mais métodos escritos em um código gerenciado e que são membros de um assembly criado na plataforma .NET e transferidos para o SQL Server.

1.3.1. Tabelas INSERTED e DELETED

O SQL Server cria na memória uma ou duas tabelas no decorrer da execução de um trigger. Essas tabelas têm a função de armazenar os dados com os quais trabalhamos. A criação de uma ou duas tabelas depende do tipo de trigger que está sendo utilizado.

Vejamos, a seguir, quais são esses tipos de triggers e as tabelas criadas pelo SQL Server:

- Dentro de triggers do tipo **INSERT** podemos acessar duas tabelas temporárias chamadas **INSERTED** e **DELETED**. A tabela **INSERTED** armazena os registros que acabaram de ser inseridos e provocaram a execução do trigger, enquanto que a tabela **DELETED** estará sempre vazia;
- Dentro de triggers do tipo **DELETE** podemos acessar duas tabelas temporárias chamadas **INSERTED** e **DELETED**. A tabela **DELETED** armazena os registros que acabaram de ser excluídos e provocaram a execução do trigger, enquanto que a tabela **INSERTED** estará sempre vazia;
- Dentro de triggers do tipo **UPDATE** podemos acessar duas tabelas temporárias chamadas **INSERTED** e **DELETED**. A tabela **DELETED** armazena os registros com os dados anteriores à alteração e a tabela **INSERTED** armazena os dados posteriores à alteração.

Essas tabelas podem ser utilizadas em caráter temporário, com a finalidade de realizar um teste sobre os efeitos causados pela alteração de alguns dados, bem como para determinar as condições necessárias para a execução de algumas ações do trigger DML.

Instrução	INSERTED	DELETED
INSERT	X	
DELETE		X
UPDATE	Novo	Antigo

1.3.2. Triggers de inclusão

Um trigger **INSERT** é aquele que é executado quando uma instrução **INSERT** insere dados em uma tabela ou view na qual o trigger está configurado.

Temos a agregação de novas linhas, tanto à tabela do trigger quanto à tabela **INSERTED**, quando um trigger **INSERT** é disparado.

A tabela **INSERTED** é uma tabela lógica constituída de uma cópia das linhas que foram inseridas. Ela contém o registro da atividade da instrução **INSERT**. As linhas na tabela **INSERTED** são sempre duplicadas de uma ou mais linhas da tabela do trigger.

A tabela **INSERTED** permite referenciar os dados registrados de uma instrução **INSERT** inicial. O trigger pode examinar a tabela **INSERTED** a fim de determinar se as ações do trigger devem ser executadas ou como ocorrerão essas execuções.

O comando **INSERT** registra a operação no Transaction Log e aciona o trigger, criando a tabela **INSERTED** na memória.

1.3.3. Triggers de exclusão

Há um tipo especial de stored procedure executada toda vez que uma instrução **DELETE** exclui dados de uma tabela ou view na qual o trigger está configurado. Estamos nos referindo ao trigger **DELETE**.

Quando um trigger **DELETE** é disparado, as linhas excluídas da tabela afetada são inseridas em uma tabela lógica chamada **DELETED**, constituída por uma cópia das linhas que foram excluídas. Ela permite referenciar dados registrados da instrução **DELETE** inicial.

1.3.4. Trigger de alteração

Um trigger **UPDATE** é um trigger disparado sempre que uma instrução **UPDATE** altera dados em uma tabela ou view na qual o trigger está configurado.

Ao ser executado pelo usuário, o comando **UPDATE** registra a operação no Transaction Log e aciona a execução do trigger, criando as tabelas **INSERTED** e **DELETED** na memória.

1.3.5. Trigger INSTEAD OF

Este trigger é responsável por determinar que o trigger DML seja executado ao invés de o comando SQL ser disparado. Com isso, as ações realizadas por comandos disparadores são sobreescritas. Apenas um trigger **INSTEAD OF** pode ser definido em uma tabela ou em uma view por cada comando **INSERT**, **UPDATE** ou **DELETE**. Apesar desse aspecto, é possível definir views em outras views, cada uma delas possuindo seu próprio trigger **INSTEAD OF**.

Os triggers **INSTEAD OF** não podem ser especificados para triggers DDL e utilizados com views aptas a serem atualizadas e que utilizam o **WITH CHECK OPTION**, pois, caso isso aconteça, o SQL Server gera um erro. A fim de evitar problemas com os triggers **INSTEAD OF**, o usuário deve utilizar a opção **ALTER VIEW** antes de defini-lo.

Quando trabalhamos com os triggers **INSTEAD OF**, não podemos utilizar os comandos **DELETE** e **UPDATE** em tabelas que possuem um relacionamento referencial que determina ações em cascata **ON DELETE** e **ON UPDATE**, respectivamente. Vale destacar que, no mínimo, uma das seguintes opções deve ser especificada: **DELETE**, **INSERT** e **UPDATE**. A função dessas opções é determinar quais comandos de alteração de dados terão a finalidade de ativar um trigger DML no momento em que ele é utilizado com uma tabela ou uma view. Essas opções podem não apenas ser utilizadas isoladamente como em conjunto.

- **Exemplo 1**

No exemplo a seguir, o objetivo é registrar em uma tabela de histórico de salários todas as alterações de salário efetuadas na tabela **TB_EMPREGADO**. Devemos registrar o código do funcionário, a data da alteração, o salário antigo e o salário novo. Vejamos a criação da tabela de histórico de salários:

```
USE PEDIDOS

CREATE TABLE EMPREGADOS_HIST_SALARIO
( NUM_MOVTO          INT IDENTITY,
  CODFUN             INT,
  DATA_ALTERACAO    DATETIME,
  SALARIO_ANTIGO    NUMERIC(12,2),
  SALARIO_NOVO      NUMERIC(12,2),
  CONSTRAINT PK_EMPREGADOS_HIST_SALARIO
  PRIMARY KEY (NUM_MOVTO) )
```

Vejamos o procedimento de criação do trigger:

```
CREATE TRIGGER TRG_EMPREGADOS_HIST_SALARIO ON TB_EMPREGADO
    FOR UPDATE
AS BEGIN
DECLARE @CODFUN INT, @SALARIO_ANTIGO FLOAT, @SALARIO_NOVO FLOAT;

-- Ler os dados antigos
SELECT @SALARIO_ANTIGO = SALARIO FROM DELETED;
-- Ler os dados novos
SELECT @CODFUN = CODFUN, @SALARIO_NOVO = SALARIO FROM INSERTED;
-- Se houver alteração de salário
IF @SALARIO_ANTIGO <> @SALARIO_NOVO
    -- Inserir dados na tabela de histórico
    INSERT INTO EMPREGADOS_HIST_SALARIO
    (CODFUN, DATA_ALTERACAO, SALARIO_ANTIGO, SALARIO_NOVO)
    VALUES
    (@CODFUN, GETDATE(), @SALARIO_ANTIGO, @SALARIO_NOVO)

END
-- Testar
UPDATE TB_EMPREGADO SET SALARIO = SALARIO * 1.2
WHERE CODFUN = 3
--
SELECT * FROM EMPREGADOS_HIST_SALARIO
-- Observar que foi inserido na tabela o registro referente à
alteração efetuada

-- Alterar o salário "em lote"
UPDATE TB_EMPREGADO SET SALARIO = SALARIO * 1.2
WHERE CODFUN IN (4,5,7)
-- Conferir se foram gerados os históricos para os 3 funcionários
SELECT * FROM EMPREGADOS_HIST_SALARIO
```

Como demonstrado nos testes realizados, esse trigger não funciona corretamente quando fazemos uma alteração em lote (vários registros de uma só vez). Isto ocorre porque, neste caso, as tabelas **INSERTED** e **DELETED** possuem três registros cada uma. Vejamos:

- **Tabela DELETED**

	CODFUN	NOME	NU...	DATA_N...	COD_DEPTO	COD_CARGO	DATA_AD...	SALARIO
1	4	PAULO CESAR JUNIOR	2	1952-03-...	8	14	1987-05-0...	600.00
2	5	JOAO LIMA MACHADO DA SILVA	2	1955-10-...	4	3	1993-01-1...	1200.00
3	7	MOHAMED ABDULAH ROSEMBERG	0	1961-07-...	11	11	1987-02-0...	4500.00

- Tabela INSERTED

	CODFUN	NOME	NU...	DATA_N...	COD_DEPTO	COD_CARGO	DATA_AD...	SALARIO
1	4	PAULO CESAR JUNIOR	2	1952-03...	8	14	1987-05-0...	720.00
2	5	JOAO LIMA MACHADO DA SILVA	2	1955-10...	4	3	1993-01-1...	1440.00
3	7	MOHAMED ABDULAH ROSEMBERG	0	1961-07...	11	11	1987-02-0...	5400.00

Ao realizarmos o procedimento a seguir, apenas a primeira linha das tabelas é lida, pois uma variável escalar não consegue armazenar mais de um valor ao mesmo tempo:

```
-- Ler os dados antigos
SELECT @SALARIO_ANTIGO = SALARIO FROM DELETED;
-- Ler os dados novos
SELECT @CODFUN = CODFUN, @SALARIO_NOVO = SALARIO FROM INSERTED;
```

A solução é utilizar um **JOIN** entre as tabelas **DELETED** e **INSERTED**, e substituir o **INSERT...VALUES** por **INSERT...SELECT**.

Vejamos o procedimento de alteração do trigger:

```
ALTER TRIGGER TRG_EMPREGADOS_HIST_SALARIO ON TB_EMPREGADO
FOR UPDATE
AS BEGIN
    INSERT INTO EMPREGADOS_HIST_SALARIO
    (CODFUN, DATA_ALTERACAO, SALARIO_ANTIGO, SALARIO_NOVO)
    SELECT I.CODFUN, GETDATE(), D.SALARIO, I.SALARIO
    FROM INSERTED I JOIN DELETED D ON I.CODFUN = D.CODFUN
    WHERE I.SALARIO <> D.SALARIO
END

-- Testar
DELETE FROM EMPREGADOS_HIST_SALARIO

UPDATE TB_EMPREGADO SET SALARIO = SALARIO * 1.2
WHERE CODFUN IN (4,5,7)

-- 
SELECT * FROM EMPREGADOS_HIST_SALARIO
```

O resultado é o seguinte:

	NUM_MOVTO	CODFUN	DATA_ALTERACAO	SALARIO_ANTIGO	SALARIO_NOVO
1	1	7	2009-03-20 16:55:09.357	4500.00	5400.00
2	2	5	2009-03-20 16:55:09.357	1200.00	1440.00
3	3	4	2009-03-20 16:55:09.357	600.00	720.00

- **Exemplo 2**

O exemplo a seguir faz com que qualquer alteração (**DELETE**, **INSERT** ou **UPDATE**) executada na tabela **TB_ITENSPEDIDO** provoque o recálculo do campo **VLR_TOTAL** de **PEDIDOS**. Vejamos:

É importante observar que, se a tabela **INSERTED** estiver vazia, podemos concluir que o trigger foi executado por um comando **DELETE**, enquanto que, se a tabela **DELETED** estiver vazia, podemos concluir que o trigger foi executado por um comando **INSERT**.

```
CREATE TRIGGER TRG_CORRIGE_VLR_TOTAL ON TB_ITENSPEDIDO
    FOR DELETE, INSERT, UPDATE
AS BEGIN
    -- Se o trigger foi executado por DELETE
    IF NOT EXISTS( SELECT * FROM INSERTED )
        UPDATE TB_PEDIDO
            SET VLR_TOTAL = (SELECT SUM( PR_UNITARIO * QUANTIDADE *
                ( 1 - DESCONTO/100 ) )
                FROM TB_ITENSPEDIDO
                WHERE NUM_PEDIDO = P.NUM_PEDIDO)
            FROM TB_PEDIDO P JOIN DELETED D
                ON P.NUM_PEDIDO = D.NUM_PEDIDO
    -- Se o trigger foi executado por INSERT ou UPDATE
    ELSE
        UPDATE TB_PEDIDO
            SET VLR_TOTAL = (SELECT SUM( PR_UNITARIO * QUANTIDADE *
                ( 1 - DESCONTO/100 ) )
                FROM TB_ITENSPEDIDO
                WHERE NUM_PEDIDO = P.NUM_PEDIDO)
            FROM TB_PEDIDO P JOIN INSERTED I
                ON P.NUM_PEDIDO = I.NUM_PEDIDO
    END

    -- Testar
    SELECT * FROM TB_PEDIDO WHERE NUM_PEDIDO = 1000
    -- Pedido 1000 -> VLR_TOTAL = 380
    SELECT * FROM TB_ITENSPEDIDO WHERE NUM_PEDIDO = 1000
    -- Possui um único item com PR_UNITARIO = 1
    UPDATE TB_ITENSPEDIDO SET PR_UNITARIO = 2
    WHERE NUM_PEDIDO = 1000
    --
    SELECT * FROM TB_PEDIDO WHERE NUM_PEDIDO = 1000
    -- VLR_TOTAL = 760
```

- **Exemplo 3**

Para manter as informações antigas do movimento de vendas, não podemos excluir um cliente, a não ser que sejam excluídos também todos os seus pedidos, o que não é de nosso interesse. Porém, se um cliente deixar de existir, podemos sinalizar que ele foi desativado. O exemplo a seguir substitui o **DELETE** de um cliente por uma sinalização de que ele está desativado.

Vejamos como criar o campo na tabela **TB_CLIENTE** para sinalizar se está ativo ou não:

```
ALTER TABLE TB_CLIENTE
ADD SN_ATIVO CHAR(1) NOT NULL DEFAULT 'S'
```

Todos os registros já foram preenchidos com **S** devido ao uso de **NOT NULL DEFAULT 'S'**.

Vejamos o procedimento para a criação do trigger:

```
CREATE TRIGGER TRG_CLIENTES_DESATIVA ON TB_CLIENTE
    INSTEAD OF DELETE
AS BEGIN
    UPDATE TB_CLIENTE SET SN_ATIVO = 'N'
    FROM TB_CLIENTE C JOIN DELETED D ON C.CODCLI = D.CODCLI
END

-- TESTAR
SELECT CODCLI, NOME, SN_ATIVO FROM TB_CLIENTE
--
DELETE FROM TB_CLIENTE WHERE CODCLI IN (4,7,11)
--
SELECT CODCLI, NOME, SN_ATIVO FROM TB_CLIENTE
```

Na figura a seguir, podemos conferir o resultado:

	CODCLI	NOME	SN_ATIVO
1	3	AUGUSTO'S FOLHINHAS LTDA	S
2	4	ASSIS BRINDES COMERCIO INDUSTRIA LTDA	N
3	5	AVEL APOLIMARIO VEICULOS S.A	S
4	6	ANTONIO M.DE SOUZA	S
5	7	ARISTON SERIGRAFIA ESTRUT.ART.PLAST.LTDA.-ME	N
6	8	APLICE SERIGRAFIA LTDA	S
7	11	ANDRE DE CASTRO A.MALAQUIAS	N
8	12	ANDRADE E SILVA BRINDES PROMOCIONAIS	S

1.4. Triggers DDL

Os **triggers DDL**, assim como os triggers regulares, executam stored procedures em resposta a um evento. Entretanto, diferentemente dos triggers DML, não são executados em resposta a uma instrução **UPDATE**, **INSERT** ou **DELETE** em uma tabela ou view. Os triggers DDL são executados em resposta a eventos DDL, que correspondem às instruções Transact-SQL, que começam com as seguintes palavras-chave: **CREATE**, **ALTER** e **DROP**.

Utilizamos os triggers DDL para as seguintes ações:

- Prevenir certas alterações no esquema do banco de dados ou determinar que algo ocorra no banco de dados conforme a alteração sofrida por esse esquema;
- Registrar alterações ou eventos realizados no esquema do banco de dados;
- Iniciar, parar, pausar, modificar e repetir os resultados de trace;
- Regular operações de banco de dados.

Os triggers DDL operam nas instruções **CREATE**, **ALTER** e **DROP**, nas stored procedures que executam operações similares às DDL ou mesmo em outras instruções DDL.

Os triggers DDL são disparados somente após uma instrução Transact-SQL ter sido executada.

Diferentemente dos triggers comuns, que respondem somente às alterações nos dados, os triggers DDL podem ser utilizados para alterações de objetos em bancos de dados. Nesse sentido, esses triggers são uma ferramenta importante para registrar as ações administrativas do sistema.

Podemos citar como exemplo a criação de um trigger na instrução **CREATE TABLE** para registrar os detalhes a respeito de uma tabela criada.

É importante considerar que não há equivalência entre as operações dos triggers DDL e dos triggers **INSTEAD OF**. Podemos utilizar a instrução **ROLLBACK TRANSACTION** para interromper a transação atual e desfazer qualquer ação que tenha sido executada, incluindo a operação DDL que dispara o trigger.

Uma única operação DDL pode executar múltiplos triggers DDL, entretanto, a ordem na qual serão executados não é documentada, por isso não haverá uma sequência específica.

1.4.1. Criando triggers DDL

Para criar um trigger DDL, utilizamos a instrução **CREATE TRIGGER**, cuja sintaxe é exibida a seguir:

```
CREATE TRIGGER nome_trigger ON {DATABASE | ALL SERVER}
FOR lista_de_eventos
AS
<corpo_do_trigger>
```

Em que:

- **DATABASE**: O escopo de execução do trigger se restringe ao banco de dados em uso no momento;
- **ALL SERVER**: O escopo de execução do trigger é no servidor onde ele foi criado, portanto, funciona para todos os bancos de dados existentes;
- **lista_de_eventos**: Eventos que irão disparar o trigger.

Vejamos quais os eventos que podem disparar o trigger:

	Server Scope	Database Scope
DDL_SERVER_LEVEL_EVENTS (CREATE DATABASE, ALTER DATABASE, DROP DATABASE)	X	
DDL_ENDPOINT_EVENTS (CREATE ENDPOINT, ALTER ENDPOINT, DROP ENDPOINT)	X	
DDL_SERVER_SECURITY_EVENTS	X	
DDL_LOGIN_EVENTS (CREATE LOGIN, ALTER LOGIN, DROP LOGIN)	X	
DDL_GDR_SERVER_EVENTS (GRANT SERVER, DENY SERVER, REVOKE SERVER)	X	
DDL_AUTHORIZATION_SERVER_EVENTS (ALTER AUTHORIZATION SERVER)	X	
DDL_DATABASE_LEVEL_EVENTS		X
DDL_TABLE_VIEW_EVENTS		X
DDL_TABLE_EVENTS (CREATE TABLE, ALTER TABLE, DROP TABLE)		X
DDL_VIEW_EVENTS (CREATE VIEW, ALTER VIEW, DROP VIEW)		X
DDL_INDEX_EVENTS (CREATE INDEX, ALTER INDEX, DROP INDEX, CREATE XML INDEX)		X
DDL_STATISTICS_EVENTS (CREATE STATISTICS, UPDATE STATISTICS, DROP STATISTICS)		X
DDL_SYNONYM_EVENTS (CREATE SYNONYM, DROP SYNONYM)		X
DDL_FUNCTION_EVENTS (CREATE FUNCTION, ALTER FUNCTION, DROP FUNCTION)		X
DDL_PROCEDURE_EVENTS (CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE)		X
DDL_TRIGGER_EVENTS (CREATE TRIGGER, ALTER TRIGGER, DROP TRIGGER)		X
DDL_EVENT_NOTIFICATION_EVENTS (CREATE EVENT NOTIFICATION, DROP EVENT NOTIFICATION)		X
DDL_ASSEMBLY_EVENTS (CREATE ASSEMBLY, ALTER ASSEMBLY, DROP ASSEMBLY)		X
DDL_TYPE_EVENTS (CREATE TYPE, DROP TYPE)		X
DDL_DATABASE_SECURITY_EVENTS		X
DDL_CERTIFICATE_EVENTS (CREATE CERTIFICATE, ALTER CERTIFICATE, DROP CERTIFICATE)		X
DDL_USER_EVENTS (CREATE USER, ALTER USER, DROP USER)		X
DDL_ROLE_EVENTS (CREATE ROLE, ALTER ROLE, DROP ROLE)		X
DDL_APPLICATION_ROLE_EVENTS (CREATE APPROLE, ALTER APPROLE, DROP APPROLE)		X
DDL_SCHEMA_EVENTS (CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA)		X
DDL_GDR_DATABASE_EVENTS (GRANT DATABASE, DENY DATABASE, REVOKE DATABASE)		X
DDL_AUTHORIZATION_DATABASE_EVENTS (ALTER AUTHORIZATION DATABASE)		X
DDL_SS EVENTS		X
DDL_MESSAGE_TYPE_EVENTS (CREATE MSGTYPE, ALTER MSGTYPE, DROP MSGTYPE)		X
DDL_CONTRACT_EVENTS (CREATE CONTRACT, DROP CONTRACT)		X
DDL_QUEUE_EVENTS (CREATE QUEUE, ALTER QUEUE, DROP QUEUE)		X
DDL_SERVICE_EVENTS (CREATE SERVICE, ALTER SERVICE, DROP SERVICE)		X
DDL_ROUTE_EVENTS (CREATE ROUTE, ALTER ROUTE, DROP ROUTE)		X
DDL_REMOTE_SERVICE_BINDING_EVENTS (CREATE REMOTE SERVICE BINDING, ALTER REMOTE SERVICE BINDING, DROP REMOTE SERVICE BINDING)		X
DDL_XML_SCHEMA_COLLECTION_EVENTS (CREATE XML SCHEMA COLLECTION, ALTER XML SCHEMA COLLECTION, DROP XML SCHEMA COLLECTION)		X
DDL_PARTITION_EVENTS		X
DDL_PARTITION_FUNCTION_EVENTS (CREATE PARTITION FUNCTION, ALTER PARTITION FUNCTION, DROP PARTITION FUNCTION)		X
DDL_PARTITION_SCHEME_EVENTS (CREATE PARTITION SCHEME, ALTER PARTITION SCHEME, DROP PARTITION SCHEME)		X

10.1. Fonte: TechNet – Microsoft SQL Server. Event Groups for Use with DDL Triggers. <[https://technet.microsoft.com/en-us/library/ms191441\(v=sql.90\).aspx](https://technet.microsoft.com/en-us/library/ms191441(v=sql.90).aspx)> (Acesso em 11 ago. 2016)

Na utilização de **CREATE TRIGGER**, devemos definir o nome do trigger, o escopo e o evento (tipo de operação DDL). Para especificar o escopo, utilizamos as cláusulas **ON DATABASE**, para banco de dados, e **ON ALL SERVER**, para o servidor.

Também, devemos utilizar a função **EventData()** para retornar dados. Os triggers DML criam as tabelas **INSERTED** e **DELETED**, permitindo que o desenvolvedor possa examinar os dados originais e os novos valores. Já os triggers DDL não criam tais tabelas; por isso, para obter as informações com relação ao evento que executa um trigger DDL, devemos utilizar **EventData()**.

EventData() retorna uma string XML com a seguinte estrutura:

```
<EVENT_INSTANCE>
    <EventType>CREATE_TABLE</EventType>
    <PostTime>2009-02-11T22:33:56.920</PostTime>
    <SPID>53</SPID>
    <ServerName>INSTRUTOR15</ServerName>
    <LoginName>PAULISTA15\Administrator</LoginName>
    <UserName>dbo</UserName>
    <DatabaseName>PEDIDOS</DatabaseName>
    <SchemaName>dbo</SchemaName>
    <ObjectName>TESTE</ObjectName>
    <ObjectType>TABLE</ObjectType>
    <TSQLCommand>
        <SetOptions ANSI_NULLS="ON"
                    ANSI_NULL_DEFAULT="ON"
                    ANSI_PADDING="ON"
                    QUOTED_IDENTIFIER="ON"
                    ENCRYPTED="FALSE"/>
        <CommandText>
            CREATE TABLE TESTE
                ( COD INT, NOME VARCHAR(30) )
        </CommandText>
    </TSQLCommand>
</EVENT_INSTANCE>
```

Para extrair o dado necessário do evento, utiliza-se **Xquery**.

- **Exemplo 1**

Vejamos um trigger com escopo no banco de dados atual:

```
CREATE TRIGGER TRG_LOG_BANCO
    ON DATABASE
    FOR DDL_DATABASE_LEVEL_EVENTS
AS BEGIN
    DECLARE @DATA XML, @MSG VARCHAR(5000);
    -- Recupera todas as informações sobre o motivo da
    -- execução do trigger
    SET @DATA = EVENTDATA();
```

```
-- Extrai uma informação específica da variável XML
SET @MSG = @DATA.value('(/EVENT_INSTANCE/EventType)[1]',
                      'Varchar(100)');
PRINT @MSG;

SET @MSG = @DATA.value('(/EVENT_INSTANCE/ObjectType)[1]',
                      'Varchar(100)');
PRINT @MSG;

SET @MSG = @DATA.value('(/EVENT_INSTANCE/ObjectName)[1]',
                      'Varchar(100)');
PRINT @MSG;

SET @MSG = @DATA.value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
                      'Varchar(4000)');
PRINT @MSG;
END

----- TESTAR
CREATE TABLE TESTE ( COD INT, NOME VARCHAR(30) )
ALTER TABLE TESTE ADD E_MAIL VARCHAR(100)
DROP TABLE TESTE
```

- **Exemplo 2**

Vejamos um trigger com escopo no servidor:

```
ALTER TRIGGER TRG_LOG_SERVER
    ON ALL SERVER
    FOR CREATE_DATABASE, DROP_DATABASE, ALTER_DATABASE, DDL_DATABASE_
LEVEL_EVENTS
AS BEGIN
DECLARE @DATA XML;
-- Recupera todas as informações sobre o motivo da
-- execução do trigger
SET @DATA = EVENTDATA();
-- Extrai uma informação específica da variável XML
SET @MSG = @DATA.value('(/EVENT_INSTANCE/EventType)[1]',
                      'Varchar(100)');
PRINT @MSG;

SET @MSG = @DATA.value('(/EVENT_INSTANCE/ObjectType)[1]',
                      'Varchar(100)');
PRINT @MSG;

SET @MSG = @DATA.value('(/EVENT_INSTANCE/ObjectName)[1]',
                      'Varchar(100)');
PRINT @MSG;
```

```
SET @MSG = @DATA.value('(/EVENT_INSTANCE/TSQLCommand/CommandText)  
[1]',  
                           'Varchar(4000)');  
PRINT @MSG;  
END  
  
-- TESTAR  
CREATE DATABASE TESTE_TRIGGER  
  
USE TESTE_TRIGGER  
  
CREATE TABLE TESTE (C1 INT, C2 VARCHAR(30))  
  
DROP DATABASE TESTE_TRIGGER
```

1.5. Triggers de logon

Um tipo especial de trigger são aqueles relacionados ao logon. Este tipo permite tanto auditar o acesso quanto bloquear acessos indevidos.

No exemplo adiante, criaremos um trigger que bloqueia qualquer acesso com um usuário de aplicação iniciado por **Adm**:

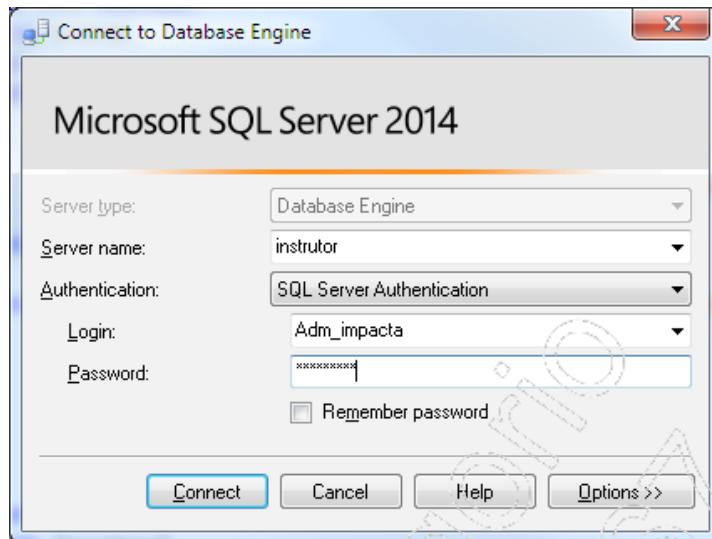
- Criação do usuário **Adm_Impacta**:

```
CREATE LOGIN Adm_Impacta WITH PASSWORD = 'Imp@ct@12'
```

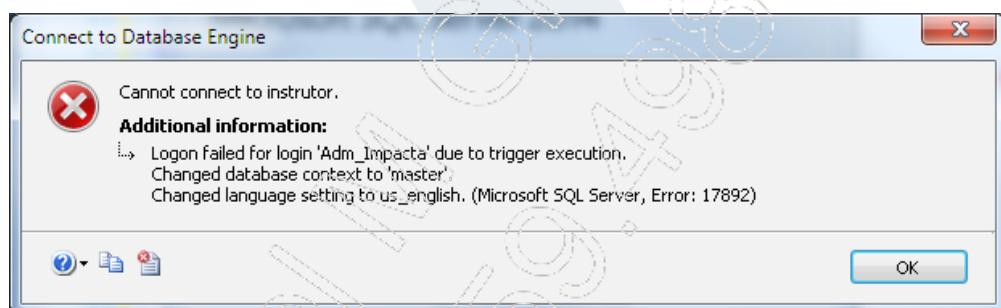
- Criação do trigger que bloqueia qualquer acesso através do SSMS com usuários de aplicação:

```
CREATE TRIGGER TRG_Bloqueio_SSMS  
ON ALL SERVER  
FOR LOGON  
AS  
BEGIN  
  
    IF (ORIGINAL_LOGIN() LIKE 'Adm_%') AND  
        APP_NAME() LIKE 'Microsoft SQL Server Management Studio%'  
        ROLLBACK  
END
```

Ao tentar efetuar o acesso, o usuário recebe um erro de execução de trigger:



- Bloqueio de acesso para usuários administrativos:



Neste outro exemplo, é criada uma tabela que armazena as informações de acesso no Servidor:

- Criação de uma tabela para receber os registros de auditoria:

```
CREATE TABLE dbo.DBA_AuditLogin(
    idPK                int          IDENTITY,
    Data                 datetime,
    ProcID               int,
    LoginID              varchar(128),
    NomeHost             varchar(128),
    App                  varchar(128),
    SchemaAutenticacao varchar(128),
    Protocolo            varchar(128),
    IPcliente            varchar(30),
    IPServidor           varchar(30),
    xmlConectInfo        xml
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```

```
GO
```

- Criação de um trigger de logon:

```
CREATE TRIGGER DBA_AuditLogin on all server
for logon
as
insert master.dbo.DBA_AutitLogin
select getdate(),@@spid,s.login_name,s.[host_name],
s.program_name,c.auth_scheme,c.net_transport,
c.client_net_address,c.local_net_address, eventdata()
from sys.dm_exec_sessions s join sys.dm_exec_connections c
on s.session_id = c.session_id
where s.session_id = @@spid
GO
```

Após o logon no SQL Server, faça uma consulta na tabela **DBA_AU**:

```
select * from dbo.DBA_AuditLogin
```

Resultado:

	Results	Messages							
idPK	Data	ProclD	LoginID	App	SchemaAutenticacao	Protocolo	IPcliente	IPservidor	
80	80	2016-06-07 14:58:29.093	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
81	81	2016-06-07 14:58:29.560	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
82	82	2016-06-07 14:58:29.570	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
83	83	2016-06-07 14:58:29.580	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
84	84	2016-06-07 14:58:29.593	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
85	85	2016-06-07 14:58:29.630	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
86	86	2016-06-07 14:58:29.640	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
87	87	2016-06-07 14:58:29.650	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL
88	88	2016-06-07 14:58:29.663	54	Instrutor	Microsoft SQL Server Management Studio - Transa...	SQL	Named pipe	<named pipe>	NULL

1.6. Aninhamento de triggers

Qualquer trigger pode conter uma instrução **UPDATE**, **INSERT** ou **DELETE** que afeta outra tabela. Os triggers são **aninhados** quando um trigger executa uma ação que inicia outro trigger.

Um trigger aninhado não será disparado duas vezes na mesma transação de trigger e não pode chamar a si mesmo em resposta a uma segunda alteração na mesma tabela dentro do trigger. Entretanto, se um trigger modificar uma tabela que, por consequência, dispara outro trigger, e este segundo trigger, por sua vez, modificar a tabela original, o trigger original irá disparar recursivamente. Para prevenir a recursão indireta deste tipo, devemos desabilitar o aninhamento de triggers.

Por padrão, a opção de configuração de um trigger aninhado é **ON** no nível do servidor.

Quando o aninhamento está habilitado, um trigger que realiza alterações sobre uma tabela pode ativar um segundo trigger, que, por sua vez, pode ativar um terceiro e assim sucessivamente.

Caso o aninhamento esteja desabilitado, os triggers deixam de ser recursivos, seja qual for a configuração determinada para **RECURSIVE_TRIGGERS** por meio de **ALTER DATABASE**.

Podemos aninhar os triggers em até 32 níveis. Caso o trigger entre em um loop infinito e, por conta disso, exceda o limite de aninhamento determinado, o processamento é interrompido e as transações são desfeitas. Visto que os triggers fazem parte de uma transação, caso haja uma falha em qualquer um dos níveis de aninhamento dos triggers, a transação inteira será desfeita e todas as alterações realizadas sobre os dados serão canceladas. Para verificar e evitar possíveis falhas, podemos incluir instruções **PRINT** quando testamos os triggers.

Os triggers aninhados permitem controlar a possibilidade de utilizar um trigger **AFTER** em cascata. Para que seja possível utilizá-lo em cascata, é preciso configurar o trigger aninhado com o valor 1, que é o padrão, pois quando ele está configurado com o valor zero (0), os triggers **AFTER** não podem ser utilizados em cascata. A quantidade máxima de triggers que podem ser colocados em cascata é 32. Seja qual for o valor de configuração, os triggers **INSTEAD OF** podem ser aninhados.

1.6.1. Habilitando e desabilitando aninhamento

Para desabilitar o aninhamento de triggers, devemos ajustar a opção **nested triggers** da stored procedure de sistema **sp_configure** para 0. Para reabilitá-lo, devemos aplicar o valor 1.

```
Exec SP_Configure 'Nested Triggers', 0  
Exec SP_Configure 'Nested Triggers', 1
```

A configuração da opção de aninhamento de triggers é feita da seguinte forma:

1. Primeiramente, abra o **Object Explorer**;
2. Com o botão direito do mouse, clique sobre o servidor;
3. Selecione a opção **Properties**;
4. Em seguida, clique sobre o nó **Misc server settings**;
5. Habilite ou desabilite a caixa de verificação **Allow triggers to be fired by other triggers**, que se encontra sob **General**.

1.7. Recursividade de triggers

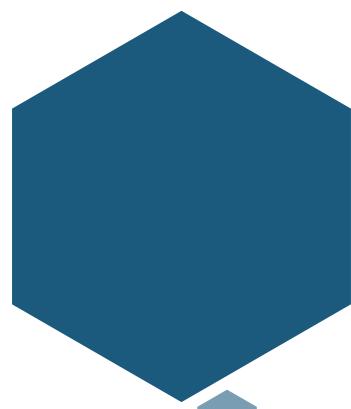
Um **trigger recursivo** é um trigger que executa uma ação que faz com que o mesmo trigger seja disparado novamente, direta ou indiretamente. Vejamos os dois tipos de recursão:

- **Recursão direta:** Ocorre quando um trigger é disparado e executa uma ação na mesma tabela que faz com que ele seja disparado novamente;
- **Recursão indireta:** Ocorre quando um trigger é disparado e executa uma ação que faz com que outro trigger seja disparado na mesma tabela ou em outra e, consequentemente, esse trigger causa uma alteração na tabela original. Essa ação acaba disparando novamente o trigger original.

Pontos principais

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

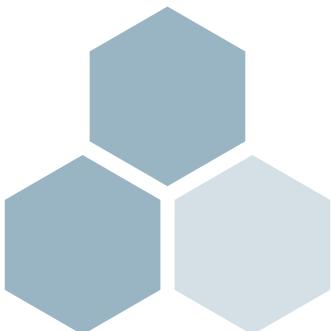
- Um **trigger**, também chamado de **gatilho**, é um tipo especial de stored procedure que é automaticamente disparado quando existe um evento de linguagem, isto é, quando é realizada alguma alteração nos dados de uma tabela. A principal vantagem dos triggers é automatizar processos;
- O SQL Server inclui três tipos genéricos de triggers: **triggers DML (Data Manipulation Language)**, **triggers DDL (Data Definition Language)** e **triggers de logon**;
- Um **trigger DML** é uma ação programada para executar quando um evento DML ocorre em um servidor de banco de dados. As instruções **UPDATE**, **INSERT** ou **DELETE** executadas em uma tabela ou view são exemplos de eventos DML;
- Os **triggers DDL** também executam stored procedures em resposta a um evento. Entretanto, diferentemente dos triggers DML, são executados em resposta a eventos DDL, que correspondem às instruções Transact-SQL que começam com as seguintes palavras-chave: **CREATE**, **ALTER** e **DROP**;
- Podemos obter uma lista de todos os triggers disponíveis em um banco de dados. Para isso, devemos efetuar uma consulta na view de catálogo **sys.triggers**. A visualização dos triggers de um banco de dados também pode ser feita a partir do **Object Explorer**, no SQL Server Management Studio;
- Qualquer trigger pode conter uma instrução **UPDATE**, **INSERT** ou **DELETE** que afeta outra tabela. Os triggers são **aninhados** quando um trigger executa uma ação que inicia outro trigger;
- Um **trigger recursivo** é um trigger que executa uma ação que faz com que o mesmo trigger seja disparado novamente, direta ou indiretamente;
- Para alterar o código interno de um trigger, utiliza-se a instrução **ALTER TRIGGER**;
- Quando um trigger não é mais necessário, podemos excluí-lo definitivamente ou apenas desabilitá-lo. Para excluir um trigger, utiliza-se a instrução **DROP TRIGGER**. As instruções **DISABLE TRIGGER** e **ENABLE TRIGGER** são utilizadas, respectivamente, para desabilitar e habilitar um trigger.



Triggers

Teste seus conhecimentos

Estes testes referem-se ao conteúdo da Aula 25.



1. Qual a funcionalidade de um trigger?

- a) Executar uma stored procedure.
- b) Gravar auditoria em comandos.
- c) Gerar uma tabela de auditoria em resposta a comandos DDL e DML.
- d) Executar uma stored procedure de forma automática.
- e) Executar uma stored procedure de forma automática para comandos DTL.

2. Qual comando dispara um trigger DDL?

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

3. Qual comando dispara um trigger DML?

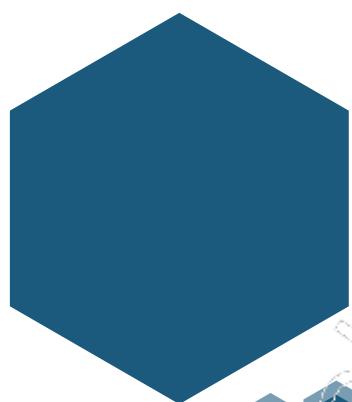
- a) CREATE
- b) ALTER
- c) DROP
- d) UPDATE
- e) TRUNCATE

4. Qual a função das tabelas INSERTED e DELETED?

- a) Armazenar as informações dos dados inseridos.
- b) Armazenar as informações dos dados excluídos.
- c) Armazenar as informações dos dados alterados.
- d) Armazenar os dados da tabela para os comandos INSERT, UPDATE e DELETE.
- e) Estas tabelas só funcionam com a cláusula OUTPUT.

5. Qual afirmação está correta sobre triggers de atualização?

- a) Podemos utilizar a tabela UPDATED para verificar as informações dos registros.
- b) A tabela UPDATED está disponível somente em triggers.
- c) Ao efetuarmos uma atualização do SQL, é disponibilizada a tabela INSERTED com as informações da atualização.
- d) As tabelas INSERTED e DELETED estão disponíveis somente nos triggers.
- e) Podemos verificar o estado dos registros antes e depois da atualização com as tabelas DELETED e INSERTED.



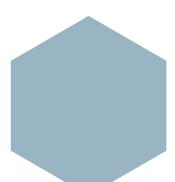
Triggers



Mãos à obra!

Gabriel M. Grigorio
497.459.678-67

Este laboratório refere-se ao conteúdo da Aula 25.



Laboratório 1

A – Trabalhando com triggers

1. Crie um trigger que registre, na tabela de histórico de preços criada pelo comando adiante, todos os reajustes de preço (**PRECO_VENDA**) da tabela **TB_PRODUTO** do banco de dados **PEDIDOS**:

```
CREATE TABLE PRODUTOS_HIST_PRECO
( NUM_MOVTO          INT IDENTITY,
  ID_PRODUTO        INT,
  DATA_ALTERACAO    DATETIME,
  PRECO_ANTIGO      NUMERIC(12,4),
  PRECO_NOVO        NUMERIC(12,4),
  CONSTRAINT PK_PRODUTOS_HIST_PRECO PRIMARY KEY (NUM_MOVTO) )
```

2. Crie um trigger que corrija o estoque (campo **QTD_REAL** da tabela **TB_PRODUTO**) toda vez que um item de pedido for incluído, alterado ou excluído. As seguintes operações devem ser executadas:

- Se o trigger for executado por causa de **DELETE**, somar em **TB_PRODUTO.QTD_REAL** a **QUANTIDADE** do item que foi deletado;
- Se o trigger for executado por causa de **INSERT**, subtrair de **TB_PRODUTO.QTD_REAL** a **QUANTIDADE** do item que foi deletado;
- Se o trigger for executado por causa de **UPDATE**, somar em **TB_PRODUTO.QTD_REAL** o valor resultante de **(DELETED.QUANTIDADE - INSERTED.QUANTIDADE)**.

3. Crie um trigger de DDL para o banco de dados **PEDIDOS** que registre, na tabela criada pelo código a seguir, todos os eventos **CREATE**, **ALTER** e **DROP** executados no nível do banco de dados:

```
CREATE TABLE TAB_LOG_BANCO
(
  ID           INT IDENTITY PRIMARY KEY,
  EventType   VARCHAR(100),
  PostTime    VARCHAR(50),
  UserName    VARCHAR(100),
  ObjectType  VARCHAR(100),
  ObjectName  VARCHAR(300),
  CommandText VARCHAR(max)
)
```

4. Crie um trigger de DDL para o banco de dados **PEDIDOS** que registre, na tabela criada a seguir, todos os eventos **CREATE**, **ALTER** e **DROP** executados no nível do banco de dados:

```
CREATE TABLE TAB_LOG_SERVER
(
    ID          INT IDENTITY PRIMARY KEY,
    DatabaseName VARCHAR(100),
    EventType   VARCHAR(100),
    PostTime    VARCHAR(50),
    UserName    VARCHAR(100),
    ObjectType  VARCHAR(100),
    ObjectName  VARCHAR(300),
    CommandText VARCHAR(max)
)
```

5. Crie um trigger de logon para bloqueio de acesso de usuários não administrativos e gravação de auditoria para acesso ao servidor. Utilize a tabela a seguir para armazenar as informações e lembre-se de criar no banco **MASTER**:

```
CREATE TABLE DBA_AuditLogin(
    idPK int IDENTITY(1,1),
    Data datetime ,
    ProcID int ,
    LoginID varchar(128) ,
    NomeHost varchar(128) ,
    App varchar(128) ,
    SchemaAutenticacao varchar(128) ,
    Protocolo varchar(128) ,
    IPcliente varchar(30) ,
    IPServidor varchar(30) ,
    xmlConectInfo xml
)
```

