



SQL 2019 – Módulo II







SQL 2019 - Módulo II





Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

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

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

SQL 2019 - Módulo II

Coordenação Geral

Henrique Thomaz Bruscagin

Autoria

Daniel Paulo Tamarosi Salvador

Revisão Ortográfica e Gramatical

Marcos Cesar dos Santos Silva

Diagramação

Bruno de Oliveira Santos

Edição nº 1 | 1883_1

Janeiro/ 2020

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

Autoria: Daniel Paulo Tamarosi Salvador

Capítulo 1 - Comandos adicionais	11
1.1. CASE	12
1.2. UNION	13
1.2.1. UNION ALL	14
1.3. EXCEPT e INTERSECT	15
1.4. IIF/CHOOSE	18
1.5. LAG e LEAD	20
1.6. Paginação (FETCH e OFFSET)	21
1.7. Consultas cruzadas	22
1.7.1. PIVOT ()	23
1.7.2. UNPIVOT()	28
1.8. CROSS APPLY e OUTER APPLY	29
Pontos principais	33
Teste seus conhecimentos	35
Mãos à obra!	39
Capítulo 2 - Dados temporários	43
2.1. Tabela temporárias	44
2.2. SELECT INTO	45
2.3. Subconsultas e tabelas temporárias	46
2.4. COMMON TABLE EXPRESSIONS (CTE)	48
2.4.1. CTE Recursiva	50
Pontos principais	54
Teste seus conhecimentos	55
Mãos à obra!	59
Capítulo 3 - Dados espaciais	61
3.1. Introdução	62
3.2. Resultado espacial	62
3.3. Tipos de dados geográficos	63
Pontos principais	67
Teste seus conhecimentos	69
Mãos à obra!	73
Capítulo 4 - Views	75
4.1. Introdução	76
4.2. Tipos de VIEWS	76
4.3. Vantagens	76
4.4. Restrições	77
4.5. Criando uma VIEW	77
4.5.1. WITH ENCRYPTION	79
4.5.2. WITH SCHEMABINDING	80
4.5.3. WITH CHECK OPTION	81
4.5.4. Criando índices	82
4.6. ALTER VIEW	83
4.7. DROP VIEW	84
4.8. Visualizando informações sobre VIEWS	84

4.9.	VIEWS atualizáveis	85
4.10.	Retornando dados tabulares	86
	Pontos principais	89
	Teste seus conhecimentos.....	91
	Mãos à obra!.....	95
Capítulo 5 - Funções nativas		97
5.1.	Funções	98
5.1.1.	Funções determinísticas e não determinísticas	98
5.2.	Funções de texto	99
5.3.	Funções matemáticas	105
5.4.	Funções de data e hora	108
5.5.	Funções de conversão	115
5.6.	Funções de classificação	120
5.6.1.	ROW_NUMBER	120
5.6.2.	RANK	122
5.6.3.	DENSE_RANK	123
5.6.4.	NTILE	125
5.6.5.	ROW_NUMBER, RANK, DENSE_RANK e NTILE	126
	Pontos principais	127
	Teste seus conhecimentos.....	129
	Mãos à obra!.....	133
Capítulo 6 - Programação		135
6.1.	Introdução	136
6.2.	Variáveis	136
6.2.1.	Atribuindo valores às variáveis	136
6.3.	Operadores	137
6.3.1.	Operadores aritméticos	137
6.3.2.	Operadores relacionais	138
6.3.3.	Operadores lógicos	138
6.3.4.	Precedência	139
6.4.	Controle de fluxo	139
6.4.1.	BEGIN/END	140
6.4.2.	IF/ELSE	140
6.5.	WHILE	141
6.5.1.	BREAK	142
6.6.	CONTINUE	142
6.6.1.	Exemplos	143
6.7.	Outros comandos	144
6.7.1.	GOTO	144
6.7.2.	RETURN	145

6.7.3.	WAITFOR.....	145
6.7.4.	EXISTS.....	146
6.7.5.	Atribuição de valor de uma consulta	146
6.8.	Queries dinâmicas	146
6.9.	Tratamento de erros	147
6.9.1.	Severidade de um erro	147
6.9.2.	@@ERROR	148
6.9.3.	TRY...CATCH	149
6.9.4.	Funções para tratamento de erros.....	151
6.10.	Mensagens de erro.....	152
6.10.1.	SP_ADDMESSAGE.....	152
6.10.2.	RAISERROR.....	153
6.10.3.	THROW	155
	Pontos principais	158
	Teste seus conhecimentos.....	159
	Mãos à obra!.....	163
	Capítulo 7 - Stored procedures	169
7.1.	Introdução	170
7.2.	STORED PROCEDURES	170
7.2.1.	Vantagens.....	171
7.2.2.	Considerações	171
7.2.3.	CREATE PROCEDURE.....	173
7.2.4.	Alterando stored procedures.....	175
7.2.5.	Excluindo STORED PROCEDURES	175
7.2.6.	Declarando parâmetros	175
7.2.7.	Exemplos	176
7.2.8.	Passagem de parâmetros posicional.....	177
7.2.9.	Passagem de parâmetros nominal.....	177
7.2.10.	Retornando valores	178
7.2.11.	PRINT	179
7.2.12.	SELECT.....	180
7.2.13.	Parâmetros de saída (OUTPUT)	181
7.3.	CURSOR	182
7.4.	Parâmetros tabulares (TABLE-VALUED).....	185
7.5.	Boas práticas	186
7.6.	Recompilando stored procedures	187
7.7.	XP_CMDSHELL	188
7.8.	CLR STORED PROCEDURE	191
7.9.	SP_EXECUTE_EXTERNAL_SCRIPT	192
7.10.	Compilação Nativa	194
	Pontos principais	195
	Teste seus conhecimentos.....	197
	Mãos à obra!.....	201

Capítulo 8 - Funções	207
8.1. Introdução	208
8.2. Funções e STORED PROCEDURES	208
8.3. Funções definidas pelo usuário	208
8.4. Funções escalares	209
8.5. Funções tabulares	220
8.5.1. Funções tabulares com várias instruções	220
8.5.2. Funções tabulares IN-LINE	222
8.6. Campos computados com funções	223
Pontos principais	225
Teste seus conhecimentos	227
Mãos à obra!	231
Capítulo 9 - Triggers	235
9.1. Introdução	236
9.2. Triggers	236
9.2.1. TRIGGERS e CONSTRAINTS	237
9.2.2. Considerações	238
9.2.3. Visualizando triggers	239
9.2.4. Alterando triggers	239
9.2.5. Desabilitando e excluindo triggers	239
9.2.5.1. DISABLE TRIGGER	239
9.2.5.2. ENABLE TRIGGER	240
9.2.5.3. DROP TRIGGER	241
9.3. Triggers DML	241
9.3.1. Tabelas INSERTED e DELETED	242
9.3.2. Triggers de inclusão	243
9.3.3. Triggers de exclusão	243
9.3.4. Trigger de alteração	244
9.3.5. Trigger INSTEAD OF	244
9.4. Triggers DDL	249
9.4.1. Criando triggers DDL	249
9.5. Triggers de logon	255
9.6. Aninhamento de triggers	257
9.6.1. Habilitando e desabilitando aninhamento	258
9.7. Recursividade de triggers	258
Pontos principais	259
Teste seus conhecimentos	261
Mãos à obra!	265

Capítulo 10 - Acesso a recursos externos.....	269
10.1. Introdução	270
10.2. OPENROWSET	270
10.3. BULK INSERT	276
10.4. XML	279
10.4.1. FOR XML	279
10.4.2. Métodos XML	298
10.4.2.1. Query.....	298
10.4.2.2. Value	300
10.4.2.3. Exists	301
10.4.2.4. Nodes	302
10.4.3. Gravando um arquivo XML	303
10.4.4. Abrindo um arquivo XML.....	304
10.5. JSON	305
10.5.1. FOR JSON	305
10.5.2. OPENJSON	306
10.5.3. JSON_VALUE.....	308
10.5.4. JSON_QUERY	309
10.5.5. ISJSON.....	310
10.5.6. Exportação para arquivo JSON	311
10.5.7. Importação de arquivo JSON.....	312
Pontos principais	314
Teste seus conhecimentos.....	315
Mãos à obra!.....	319



Comandos adicionais

- ◆ CASE;
- ◆ UNION;
- ◆ EXCEPT e INTERSECT;
- ◆ IIF/CHOOSE;
- ◆ LAG e LEAD;
- ◆ Paginação (FETCH e OFFSET);
- ◆ Consultas cruzadas;
- ◆ CROSS APPLY e OUTER APPLY.



1.1.CASE

Os valores pertencentes a uma coluna podem ser testados por meio da cláusula **CASE** em conjunto com o comando **SELECT**. Dessa maneira, é possível aplicar diversas condições de validação em uma consulta.

No exemplo a seguir, **CASE** é utilizado para verificar se os funcionários da tabela **TB_EMPREGADO** são ou não sindicalizados:

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

	NOME	SALARIO	Sindicato?	DATA_ADMISSAO
1	OLAVO	3000.00	Sim	2009-11-20 00:00:00.000
2	JOSE	600.00	Sim	2017-07-24 00:00:00.000
3	MARCELO	2400.00	Sim	2013-06-15 00:00:00.000
4	PAULO	600.00	Sim	2011-07-03 00:00:00.000
5	JOAO	1200.00	Sim	2012-11-21 00:00:00.000
6	CARLOS ALBERTO	4500.00	Sim	2011-05-18 00:00:00.000
7	ELIANE	1200.00	Sim	2015-01-04 00:00:00.000
8	RUDGE	800.00	Não	2015-01-26 00:00:00.000
9	MARIA APARECIDA	1200.00	Não	2011-07-29 00:00:00.000
10	FERNANDO	1200.00	Sim	2018-02-28 00:00:00.000

Já no próximo exemplo, verificamos em qual dia da semana os empregados foram admitidos:

```
SELECT NOME, SALARIO, DATA_ADMISSAO,
      CASE DATEPART(WEEKDAY, DATA_ADMISSAO)
      WHEN 1 THEN 'Domingo'
      WHEN 2 THEN 'Segunda-Feira'
      WHEN 3 THEN 'Terça-Feira'
      WHEN 4 THEN 'Quarta-Feira'
      WHEN 5 THEN 'Quinta-Feira'
      WHEN 6 THEN 'Sexta-Feira'
      WHEN 7 THEN 'Sábado'
      END AS DIA_SEMANA
FROM TB_EMPREGADO;
```

	NOME	SALARIO	DATA_ADMISSAO	DIA_SEMANA
1	OLAVO	3000.00	2009-11-20 00:00:00.000	Sexta-Feira
2	JOSE	600.00	2017-07-24 00:00:00.000	Segunda-Feira
3	MARCELO	2400.00	2013-06-15 00:00:00.000	Sábado
4	PAULO	600.00	2011-07-03 00:00:00.000	Domingo
5	JOAO	1200.00	2012-11-21 00:00:00.000	Quarta-Feira
6	CARLOS ALBERTO	4500.00	2011-05-18 00:00:00.000	Quarta-Feira
7	ELIANE	1200.00	2015-01-04 00:00:00.000	Domingo
8	RUDGE	800.00	2015-01-26 00:00:00.000	Segunda-Feira
9	MARIA APARECIDA	1200.00	2011-07-29 00:00:00.000	Sexta-Feira
10	FERNANDO	1200.00	2018-02-28 00:00:00.000	Quarta-Feira

1.2. UNION

A cláusula **UNION** combina resultados de duas ou mais queries em um conjunto de resultados simples, incluindo todas as linhas de todas as queries combinadas. Ela é utilizada quando é preciso recuperar todos os dados de duas tabelas, sem fazer associação entre elas.

Para utilizar **UNION**, é necessário que o número e a ordem das colunas nas queries sejam iguais, bem como os tipos de dados sejam compatíveis. Se os tipos de dados forem diferentes em precisão, escala ou extensão, as regras para determinar o resultado serão as mesmas das expressões de combinação.

O operador **UNION**, por padrão, elimina linhas duplicadas do conjunto de resultados.

Veja o seguinte exemplo:

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

	NOME	FONE1
1	ADSMAlVQDCMSKMenMOXXwAKXUGMAVW	30417760482
2	AFGPOBICSMGOGIVPWQKTOQWMNLUPME	74613004626
3	AGMRBFSMVXKFNEOUCHIHVWGVRPPGM	26731572424
4	AGUKSBQWFRJEJNFBIRCUQYPIVTEET	83011724547
5	AIBFYQAKWHLTXHIMNRDQPKUSLXICM	34122732671
6	AIVNLERCVTPCUKKJDEJSOSIVWWPXW	72565316137
7	AJDCNKEYLWXPGOWHFKNLOGELBTUYCG	21623282637
8	ANXTUDDIVPWQUVINKNFBVLEOSUODXI	18867421776
9	AOPNWJXRKRBDLTQKHPLRMXCHCMPCX	13311317434
10	ARHDSLQHMFMVLARJVFECTLOVGHMBJ	55011320625
11	ASIUCCWwKNVGIBNxBLUVUCQHHKRTXH	55840528065

Verifique que a consulta retorna 562 linhas.

1.2.1. UNION ALL

A **UNION ALL** é a cláusula responsável por unir informações obtidas a partir de diversos comandos **SELECT**. Para obter esses dados, não há necessidade de que as tabelas que os possuem estejam relacionadas.

Para utilizar a cláusula **UNION ALL**, é necessário considerar as seguintes regras:

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

Enquanto **UNION**, por padrão, elimina linhas duplicadas do conjunto de resultados, **UNION ALL** inclui todas as linhas nos resultados e não remove as linhas duplicadas. A seguir, veja um exemplo da utilização de **UNION ALL**:

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

	NOME	FONE1
1	ADSMIAVQOCMSK MENMOXXWAKXUGMAVW	30417760482
2	ADSMIAVQOCMSK MENMOXXWAKXUGMAVW	30417760482
3	AFGPOBICSMGOGIVPWQKTOQWMNLUPME	74613004626
4	AFGPOBICSMGOGIVPWQKTOQWMNLUPME	74613004626
5	AGMRBFMSMVXKFNEOUCIHIWGXVRPPGM	26731572424
6	AGMRBFMSMVXKFNEOUCIHIWGXVRPPGM	26731572424
7	AGUKSBQWFRJEJNFBIRCUQYPIVTEET	83011724547
8	AGUKSBQWFRJEJNFBIRCUQYPIVTEET	83011724547
9	AIBFYQAKWHLTXHIMNRDQPKUSLXICM	34122732671
10	AIBFYQAKWHLTXHIMNRDQPKUSLXICM	34122732671
11	AIVNLERCVTPCUKJDSEJSOSIVWWPXW	72565316137
12	AIVNLERCVTPCUKJDSEJSOSIVWWPXW	72565316137

Nesse caso a consulta retornará 1124 linhas.

1.3.EXCEPT e INTERSECT

Os resultados de duas instruções **SELECT** podem ser comparados por meio dos operadores **EXCEPT** e **INTERSECT**, resultando, assim, em novos valores.

O operador **INTERSECT** retorna os valores encontrados nas duas consultas, tanto a que está à esquerda quanto a que está à direita do operador na sintaxe a seguir:

```
<instrução_select_1>  
INTERSECT  
<instrução_select_2>
```

O operador **EXCEPT** retorna os valores da consulta à esquerda que não se encontram também na consulta à direita:

```
<instrução_select_1>  
EXCEPT  
<instrução_select_2>
```

Os operadores **EXCEPT** e **INTERSECT** podem ser utilizados juntamente com outros operadores em uma expressão. Neste caso, a avaliação da expressão segue uma ordem específica: em primeiro lugar, as expressões em parênteses; em seguida, o operador **INTERSECT**; e, por fim, o operador **EXCEPT**, avaliado da esquerda para a direita, de acordo com sua posição na expressão.

Também podemos utilizar **EXCEPT** e **INTERSECT** para comparar mais de duas queries. Quando for assim, a conversão dos tipos de dados é feita pela comparação de duas consultas ao mesmo tempo, de acordo com a ordem de avaliação que apresentamos.

Para utilizar **EXCEPT** e **INTERSECT**, é necessário que as colunas estejam em mesmo número e ordem em todas as consultas, e que os tipos de dados sejam compatíveis.

A seguir, temos exemplos que demonstram a utilização dos operadores **EXCEPT** e **INTERSECT**.

Para auxiliar nos exemplos, vamos criar algumas tabelas temporárias (que serão explicadas no próximo capítulo).

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO  
INTO #TMP_SALARIO_ACIMA_5000  
FROM TB_EMPREGADO  
WHERE SALARIO > 5000
```

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO  
INTO #TMP_DEPART_1_4_5  
FROM TB_EMPREGADO  
WHERE ID_DEPARTAMENTO IN (1,4,5)
```

• Exemplo 1

Vamos consultar os registros que são idênticos na tabela de empregados e os empregados da tabela #TMP_SALARIO_ACIMA_5000:

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM TB_EMPREGADO
INTERSECT
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM #TMP_SALARIO_ACIMA_5000
```

Verifique o resultado:

	ID_DEPARTAMENTO	NOME	DATA_NASCIMENTO
1	1	CARLOS FERNANDO	1960-02-19 00:00:00.000
2	1	JOSÉ	1990-10-11 00:00:00.000
3	2	SEBASTIÃO	1961-10-27 00:00:00.000

No resultado, são apresentados os registros que aparecem nas duas consultas.

• Exemplo 2

Na próxima consulta, é realizada a comparação da tabela de empregados com a #TMP_DEPART_1_4_5:

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM TB_EMPREGADO
INTERSECT
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM #TMP_DEPART_1_4_5
```

O resultado do código anterior é exibido a seguir:

	ID_DEPARTAMENTO	NOME	DATA_NASCIMENTO
1	1	ARNALDO	1989-11-09 00:00:00.000
2	1	CARLOS FERNANDO	1960-02-19 00:00:00.000
3	1	CASSIANO	1961-11-24 00:00:00.000
4	1	JOÃO	1969-11-09 00:00:00.000
5	1	JORGE	1990-10-11 00:00:00.000
6	1	JOSÉ	1969-03-26 00:00:00.000
7	1	JOSÉ	1990-10-11 00:00:00.000
8	1	PEDRO	1990-05-10 00:00:00.000
9	1	ROBERTO	1996-09-18 00:00:00.000
10	1	ROBERTO ALEXANDRO	1990-10-11 00:00:00.000
11	1	ROGÉRIO	1989-11-09 00:00:00.000
12	1	RONALDO	1989-11-09 00:00:00.000
13	4	CARLOS	2002-09-29 00:00:00.000

- Exemplo 3

O exemplo a seguir lista os registros da tabela de empregados que não estão relacionados com a tabela **#TMP_SALARIO_ACIMA_5000**. (Lembrando que a comparação é somente das colunas **ID_DEPARTAMENTO**, **NOME** e **DATA_NASCIMENTO**):

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM TB_EMPREGADO
EXCEPT
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM #TMP_SALARIO_ACIMA_5000
```

O resultado do código anterior é exibido a seguir:

	ID_DEPARTAMENTO	NOME	DATA_NASCIMENTO
1	NULL	ANTONIO JORGE	1990-05-10 00:00:00.000
2	NULL	JOSE	NULL
3	NULL	SEVERINO	1998-12-20 00:00:00.000
4	1	ARNALDO	1989-11-09 00:00:00.000
5	1	CASSIANO	1961-11-24 00:00:00.000
6	1	JOÃO	1969-11-09 00:00:00.000
7	1	JORGE	1990-10-11 00:00:00.000
8	1	JOSÉ	1969-03-26 00:00:00.000
9	1	PEDRO	1990-05-10 00:00:00.000
10	1	ROBERTO	1996-09-19 00:00:00.000
11	1	ROBERTO ALEXANDRO	1990-10-11 00:00:00.000
12	1	ROGÉRIO	1989-11-09 00:00:00.000
13	1	RONALDO	1989-11-09 00:00:00.000

- Exemplo 4

O código a seguir apresenta os empregados que são do departamento 6 e que não estão na tabela **#TMP_SALARIO_ACIMA_5000**:

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 6
EXCEPT
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM #TMP_SALARIO_ACIMA_5000
```

O resultado do código anterior é exibido a seguir:

	ID_DEPARTAMENTO	NOME	DATA_NASCIMENTO
1	6	ALTAMIR	1973-03-23 00:00:00.000
2	6	ARLINDO	2002-09-19 00:00:00.000
3	6	ELIANE	1964-11-22 00:00:00.000
4	6	OMAR	1985-07-09 00:00:00.000

• Exemplo 5

O código a seguir consulta os empregados com salário maior que 3500 e que não estão na tabela #TMP_DEPART_1_4_5:

```
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM TB_EMPREGADO
WHERE SALARIO>3500
EXCEPT
SELECT ID_DEPARTAMENTO , NOME , DATA_NASCIMENTO
FROM #TMP_DEPART_1_4_5
```

O resultado do código anterior é exibido a seguir:

	ID_DEPARTAMENTO	NOME	DATA_NASCIMENTO
1	2	ROBERTO MARIA	1967-12-22 00:00:00.000
2	2	SEBASTIÃO	1961-10-27 00:00:00.000
3	3	MANOEL ELIAS	2002-11-09 00:00:00.000
4	11	CARLOS ALBERTO	1971-05-15 00:00:00.000

1.4.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**:

```
SELECT IIF (15 > 10, 'VERDADEIRO', 'FALSO') AS Resultado
```

	Resultado
1	VERDADEIRO

```
SELECT
ID_CLIENTE ,
TIPO_CLI,
IF( TIPO_CLI = 'C' , 'CLIENTE' , 'REVENDA') AS TIPO_CLIENTE
FROM TB_CLIENTE
```

	ID_CLIENTE	TIPO_CLI	TIPO_CLIENTE
1	3	C	CLIENTE
2	4	C	CLIENTE
3	5	C	CLIENTE
4	6	C	CLIENTE
5	7	C	CLIENTE
6	8	C	CLIENTE
7	11	C	CLIENTE
8	12	C	CLIENTE
9	13	C	CLIENTE
10	14	R	REVENDA
11	15	C	CLIENTE
12	16	C	CLIENTE
13	17	R	REVENDA

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 **NULL**;
- **<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:

```
SELECT CHOOSE (4, 'A', 'B', 'C', 'D', 'E', 'F', 'G') AS
RESULTADO
```

	RESULTADO
1	D

Exemplo 2

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

```
SELECT
ID_EMPREGADO, 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
```

	ID_EMPREGADO	NOME	DATA_ADMISSAO	SINDICALIZADO	DIA_SEMANA
1	1	OLAVO	2009-11-20 00:00:00.000	SIM	SEXTA
2	2	JOSE	2017-07-24 00:00:00.000	SIM	SEGUNDA
3	3	MARCELO	2013-06-15 00:00:00.000	SIM	SÁBADO
4	4	PAULO	2011-07-03 00:00:00.000	SIM	DOMINGO
5	5	JOAO	2012-11-21 00:00:00.000	SIM	QUARTA
6	7	CARLOS ALBERTO	2011-05-18 00:00:00.000	SIM	QUARTA
7	8	ELIANE	2015-01-04 00:00:00.000	SIM	DOMINGO
8	9	RUDGE	2015-01-26 00:00:00.000	NÃO	SEGUNDA
9	10	MARIA APARECIDA	2011-07-29 00:00:00.000	NÃO	SEXTA
10	11	FERNANDO	2018-02-28 00:00:00.000	SIM	QUARTA
11	12	JOAO	2019-02-22 00:00:00.000	SIM	SEXTA
12	13	OSMAR	2009-07-14 00:00:00.000	SIM	TERÇA

1.5.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:** Quantidade 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 ID_EMPREGADO, NOME, SALARIO,
LAG(SALARIO,1, 0) OVER (ORDER BY ID_EMPREGADO) AS SALARIO_
ANTERIOR,
LEAD(SALARIO,1, 0) OVER (ORDER BY ID_EMPREGADO) AS PROXIMO_
SALARIO
FROM TB_EMPREGADO
ORDER BY ID_EMPREGADO
```

O resultado a seguir será mostrado:

	ID_EMPREGADO	NOME	SALARIO	SALARIO_ANTERIOR	PROXIMO_SALARIO
1	1	OLAVO	3000.00	0.00	600.00
2	2	JOSE	600.00	3000.00	2400.00
3	3	MARCELO	2400.00	600.00	600.00
4	4	PAULO	600.00	2400.00	1200.00
5	5	JOAO	1200.00	600.00	4500.00
6	7	CARLOS ALBERTO	4500.00	1200.00	1200.00
7	8	ELIANE	1200.00	4500.00	800.00
8	9	RUDGE	800.00	1200.00	1200.00
9	10	MARIA APARECI	1200.00	800.00	1200.00
10	11	FERNANDO	1200.00	1200.00	1200.00
11	12	JOAO	1200.00	1200.00	2400.00
12	13	OSMAR	2400.00	1200.00	1200.00
13	14	CASSIANO	1200.00	2400.00	2400.00

1.6. 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 ID_CLIENTE
OFFSET 0 ROWS FETCH NEXT 20 ROWS ONLY;

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

O resultado a seguir será mostrado:

Results		Messages						
	ID_CLIENTE	NOME	FANTASIA	CNPJ	IE	FONE1	FONE2	E_MAIL
1	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	UQDQESKOGMNPVHJ	75441168107640	2366546204042665373	23266266455	56212106888	XIMIKNUJBPGWXX
2	4	QQCQBPNKUNHDSFBMEKESJNMJMJQDDP	CMEKPTSEFHGUGHG	13415416137526	1232174750387123066	12377276844	31630275544	QXMBXDMWXXIVM
3	5	TTOMDCPSXCWFXSQWVNOVURRENQDFKL	LKWDRTTERXHQKMT	47554254311717	3736373517726751140	46116241685	53443550354	DORJNTQNSQWV
4	6	ANKTUDDIVPWQUVINKNFBVLEOSUODXI	CXNEJDNDBNYQHTN	15532554268767	0342873281571252357	18867421776	61751768457	EKTGENKXSSROH
5	7	TCSGVJISYISLFFELSMILLYSBMPRQNK	OSGDKNHGJYFUPCJ	50765211004317	4144613835402321551	34271555417	48567565135	EQWVGJIXURGG
6	8	QXVTRVGAOBVBXCAAQIGMMHNFIFVJH	RWFFYBWCVINNNHM	63337228707746	1450345882181175375	63367513277	41554102380	KXDJFQVCUDVFE
7	11	OVRUUMSLRNTIUVGPCVLHFJQTKRSDKD	OCLKEQHQCUIYRLN	47521518136144	5334443718211822357	16243601251	25433201182	WFPVGRYLRLBU
8	12	RDVNRMHFEIRJCCTAQDFLBJHHMUIO	FNHUFRIWVCGIIP	88112414378743	8173731723766842245	86343273643	74184811870	UKVRFRLUIDIMR
◀								
	ID_CLIENTE	NOME	FANTASIA	CNPJ	IE	FONE1	FONE2	E_MAIL
1	27	MUWCSFWSLXSKJQJULJVSHVKWNUUEU	CLBTHUVSRYUDCNL	25143533660624	8557604400445254233	26024377847	07132627563	IIKLDDPMKHFTUHV
2	28	RRGTFHWUQUAXGPVRDQMHVCOTEBSQTB	SGPRILTMXYLFGG	33634123534580	1613361155356508374	47151645735	46465156456	BDGUFULQOMEVRV
3	29	PNMYFQSEMGVGOBACUJUNFQENCQBGBVU	AILTERWHXJGTUVF	51744657865436	6653784377413316857	46661745323	36156121346	PTGBUWLDIWWGPL

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.

1.7.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
  ID_EMPREGADO,
  MONTH(DATA_EMISSAO) AS MES,
  YEAR(DATA_EMISSAO) AS ANO,
  SUM(VLR_TOTAL) AS TOT_VENDIDO
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO) = 2016
GROUP BY
  ID_EMPREGADO , 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.

Results Messages				
	ID_EMPREGADO	MES	ANO	TOT_VENDIDO
1	1	1	2016	39623.76
2	1	2	2016	49768.82
3	1	3	2016	29116.16
4	1	4	2016	63366.89
5	1	5	2016	39053.13
6	1	6	2016	49946.17
7	1	7	2016	42354.07
8	1	8	2016	35052.00
9	1	9	2016	60375.67
10	1	10	2016	61011.42
11	1	11	2016	39420.38
12	1	12	2016	87522.54
13	2	1	2016	55449.46
14	2	2	2016	72941.98
15	2	3	2016	33951.89
16	2	4	2016	33857.12

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	51993.74	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 gerar 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 ID_EMPREGADO AS VENDEDOR,
[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 ID_EMPREGADO, VLR_TOTAL, MONTH(DATA_EMISSAO) AS MES
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO) = 2016
) AS 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
```

	VENDEDOR	MES1	MES2	MES3	MES4	MES5	MES6	MES7	MES8	MES9	MES10	MES11	MES12
1	1	39623.76	49768.82	29116.16	63366.89	39053.13	49946.17	42354.07	35052.00	60375.67	61011.42	39420.38	87522.54
2	2	55449.46	72941.98	33951.89	33857.12	22629.57	20048.78	35467.58	66604.37	79143.15	45829.15	61978.99	72452.60
3	3	34652.20	42349.27	36252.01	32675.80	39268.78	53027.09	41302.07	36459.56	47453.34	22647.59	40407.23	138744.56
4	4	42232.22	41688.07	61579.00	34702.44	31720.85	33586.12	55899.19	63995.04	33395.40	30245.16	16615.03	72365.73
5	5	37484.06	41995.01	25435.24	38865.68	21725.20	21328.10	25210.47	29182.92	34116.45	44795.97	51808.42	41252.38
6	7	23383.82	32546.88	38620.09	43833.03	36093.78	36430.83	40599.72	42028.10	26461.63	58041.05	54321.99	29711.42
7	8	32060.65	56117.53	60820.66	41600.42	37967.30	70340.25	33954.49	30384.93	28425.74	46385.63	14205.96	50650.01
8	69	42013.00	60570.42	70766.02	37209.50	51673.74	29540.55	62209.36	18942.88	13704.78	45239.25	49980.68	31342.88

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

```
SELECT ID_EMPREGADO AS ID_VENDEDOR, VENDEDOR ,
[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.ID_EMPREGADO, E.NOME AS VENDEDOR, VLR_TOTAL,
MONTH(DATA_EMISSAO) AS MES
FROM TB_PEDIDO AS P
JOIN TB_EMPREGADO AS E ON E.ID_EMPREGADO = P.ID_EMPREGADO
WHERE YEAR(DATA_EMISSAO) = 2016
) AS 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 2016:

```
SELECT ID_CLIENTE, [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 ID_CLIENTE, VLR_TOTAL, MONTH(DATA_EMISSAO) AS MES
      FROM TB_PEDIDO
      WHERE YEAR(DATA_EMISSAO) = 2016) 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
```

	ID_CLIENTE	MES1	MES2	MES3	MES4	MES5	MES6	MES7	MES8	MES9	MES10	MES11	MES12
1	3	NULL	NULL	NULL	NULL	NULL	NULL	7964.30	NULL	NULL	NULL	NULL	NULL
2	4	NULL	4367.10	NULL	NULL	NULL	2412.42	NULL	NULL	NULL	7616.81	NULL	NULL
3	5	NULL	NULL	2.00	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	5715.53
4	6	NULL	NULL	212.78	NULL	NULL	NULL	NULL	NULL	NULL	9340.39	NULL	NULL
5	7	NULL	7880.06	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
6	11	NULL	NULL	NULL	6393.24	NULL	NULL	NULL	NULL	NULL	12292.40	NULL	NULL
7	12	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	2178.01	NULL
8	13	NULL	2419.26	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1264.19	NULL
9	14	NULL	NULL	NULL	3996.61	NULL	NULL	NULL	NULL	NULL	6684.11	NULL	NULL
10	15	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	2120.10	NULL	NULL

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

```
SELECT ID_CLIENTE, NOME,
[2016] AS '2016',
[2017] AS '2017',
[2018] AS '2018',
[2019] AS '2019'
FROM (SELECT P.ID_CLIENTE, C.NOME, P.VLR_TOTAL,
           YEAR(P.DATA_EMISSAO) AS ANO
      FROM TB_PEDIDO P
      JOIN TB_CLIENTE C ON P.ID_CLIENTE = C.ID_CLIENTE) AS X
      PIVOT( SUM(VLR_TOTAL)
FOR ANO IN ([2016],[2017],[2018],[2019])) AS Y
```

	ID_CLIENTE	NOME	2016	2017	2018	2019
1	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	7964.30	21712.48	7874.64	17979.51
2	4	QCQQBPNPUKNHDSFBMEKESJNMJMQDDP	14396.33	12168.78	18316.23	10999.87
3	5	TTOMDCPSXCWXRXSQWWNOVURRENQDFKL	5717.53	34636.31	20394.61	12148.41
4	6	ANXTUDDIVPWQUVINKNFBVLEQSUODXI	9553.17	5494.54	17837.36	17102.01
5	7	TCSGVJISYISLFFELSMMLYSBMPRQNK	7880.06	16026.41	12566.86	5176.15
6	8	QXVTRVGAOBVBXCAAQIGMMHNFIFVJH	NULL	11165.43	26980.85	30758.87
7	11	OVRUUSLRLNTIJVGPCVLHFJQTKRSDKD	18685.64	15354.34	5537.83	25281.45
8	12	RDVNRMHFEIRJCATAQDFLBHMHMUIO	2178.01	2452.38	12562.54	48305.45
9	13	EAPFLIPYEWEXBAKTVOHLPVXWKOCBVR	3683.45	2496.08	11985.51	12563.51
10	14	BPBQCCYFGMVGGIUOXJHNMTRTDGGAMQ	10680.72	9442.46	29302.98	2206.94

No exemplo seguinte, temos o total vendido de cada produto em cada um dos meses de 2016:

```
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.id_PEDIDO = I.id_PEDIDO  
WHERE YEAR(P.DATA_EMISSAO) = 2016) I  
PIVOT( SUM(VLOR) 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,  
[2016] AS ANO_2016,  
[2017] AS ANO_2017,  
[2018] AS ANO_2018,  
[2019] AS ANO_2019  
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 ([2016],[2017],[2018],[2019])  
) AS PVT  
ORDER BY 1
```

Results		Messages			
	MES	ANO_2016	ANO_2017	ANO_2018	ANO_2019
1	1	306899.17	501962.17	719317.70	3362097.41
2	2	397977.98	388374.75	615493.97	654849.98
3	3	356541.07	413728.09	721816.59	632680.37
4	4	326110.96	510549.79	616429.74	644567.85
5	5	280132.35	436959.73	702276.96	684837.14
6	6	314247.89	545082.90	661344.87	619744.50
7	7	336996.95	674456.79	669380.14	745572.13
8	8	322649.80	730188.07	685813.94	639566.11
9	9	323076.16	661445.75	552194.67	704857.93

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 ID_DEPARTAMENTO, ['A'], ['B'], ['C'], ['D'], ['E']
FROM (SELECT ID_EMPREGADO, ID_DEPARTAMENTO, LEFT(NOME,1) AS
LETRA FROM TB_EMPREGADO) AS P
PIVOT (COUNT(ID_EMPREGADO) 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 ID_DEPARTAMENTO, [65], [66], [67], [68], [69]
FROM (SELECT ID_EMPREGADO, ID_DEPARTAMENTO, ASCII(
UPPER(LEFT(NOME,1))) AS LETRA FROM TB_EMPREGADO) AS P
PIVOT (COUNT(ID_EMPREGADO) FOR LETRA IN ([65], [66], [67],
[68], [69])) AS PVT
```

Results		Messages				
	ID_DEPARTAMENTO	65	66	67	68	69
1	NULL	1	0	0	0	0
2	1	1	0	2	0	0
3	2	0	0	0	0	1
4	3	4	0	0	0	0
5	4	0	0	2	0	0
6	5	3	0	0	0	0
7	6	2	0	0	0	1
8	7	0	0	1	0	0
9	8	0	0	0	0	0
10	9	1	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.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 CÓDIGO E NOME DO CLIENTE, VALOR TOTAL E DATA DA EMISSÃO.
```

```
SELECT C.ID_CLIENTE, C.NOME, P.ID_PEDIDO, P.VLR_TOTAL, P.DATA_EMISSAO
FROM TB_CLIENTE AS C
JOIN TB_PEDIDO AS P ON P.ID_CLIENTE = C.ID_CLIENTE
ORDER BY C.ID_CLIENTE, P.ID_PEDIDO
```

	ID_CLIENTE	NOME	ID_PEDIDO	VLR_TOTAL	DATA_EMISSAO
1	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	1841	1364.12	2017-09-09 00:00:00.000
2	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	1904	2042.52	2017-09-15 00:00:00.000
3	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	2570	48.44	2017-11-21 00:00:00.000
4	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	4455	762.19	2018-06-24 00:00:00.000
5	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	5398	5894.05	2018-12-24 00:00:00.000
6	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	5737	6203.62	2019-04-18 00:00:00.000
7	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	5850	7230.31	2019-05-29 00:00:00.000
8	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	7119	7964.30	2016-07-23 00:00:00.000
9	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	7764	682.81	2017-02-23 00:00:00.000
10	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE	8384	5002.41	2017-09-24 00:00:00.000

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.ID_CLIENTE, C.NOME, P.ID_PEDIDO, P.VLR_TOTAL, P.DATA_
EMISSAO,
(SELECT COUNT(*) FROM TB_PEDIDO
WHERE ID_CLIENTE = C.ID_CLIENTE) AS QTD_PED,
(SELECT MAX(VLR_TOTAL) FROM TB_PEDIDO
WHERE ID_CLIENTE = C.ID_CLIENTE) AS MAIOR_VALOR,
(SELECT MIN(VLR_TOTAL) FROM TB_PEDIDO
WHERE ID_CLIENTE = C.ID_CLIENTE) AS MENOR_VALOR,
(SELECT MAX(DATA_EMISSAO) FROM TB_PEDIDO
WHERE ID_CLIENTE = C.ID_CLIENTE) AS ULTIMA_COMPRA
FROM TB_CLIENTE AS C
JOIN TB_PEDIDO AS P ON P.ID_CLIENTE = C.ID_CLIENTE
ORDER BY C.ID_CLIENTE, P.ID_PEDIDO
```

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

```
---Utilizando o CROSS APPLY

SELECT C.ID_CLIENTE, C.NOME, P.ID_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.ID_CLIENTE = C.ID_CLIENTE

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.ID_CLIENTE = P.ID_CLIENTE ) AS CR
ORDER BY C.ID_CLIENTE, P.ID_PEDIDO
```

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

```
-- Consulta com filtro dos pedidos de 2016.

SELECT  C.ID_CLIENTE, C.NOME, P.ID_PEDIDO, P.VLR_TOTAL, P.DATA_
EMISSAO,
(SELECT COUNT(*) FROM TB_PEDIDO
  WHERE YEAR(DATA_EMISSAO) = 2014
    AND ID_CLIENTE = C.ID_CLIENTE) AS QTD_PED,
(SELECT MAX(VLR_TOTAL) FROM TB_PEDIDO
  WHERE YEAR(DATA_EMISSAO) = 2014
    AND ID_CLIENTE = C.ID_CLIENTE) AS MAIOR_VALOR,
(SELECT MIN(VLR_TOTAL) FROM TB_PEDIDO
  WHERE YEAR(DATA_EMISSAO) = 2014
    AND ID_CLIENTE = C.ID_CLIENTE) AS MENOR_VALOR,
(SELECT MAX(DATA_EMISSAO) FROM TB_PEDIDO
  WHERE YEAR(DATA_EMISSAO) = 2014
    AND ID_CLIENTE = C.ID_CLIENTE) AS ULTIMA_COMPRA
FROM TB_CLIENTE AS C
JOIN TB_PEDIDO AS P ON P.ID_CLIENTE = C.ID_CLIENTE
WHERE YEAR(DATA_EMISSAO) = 2016
```

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

```
--UTILIZANDO O CROSS APPLY

SELECT  C.ID_CLIENTE, C.NOME, P.ID_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.ID_CLIENTE = C.ID_CLIENTE
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.ID_CLIENTE = PC.ID_CLIENTE AND YEAR(PC.DATA_EMISSAO)
= 2014 ) AS CR
WHERE YEAR(P.DATA_EMISSAO) = 2016
ORDER BY C.ID_CLIENTE, P.ID_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.DEPARTAMENTO , E.NOME
FROM TB_DEPARTAMENTO AS D
LEFT JOIN TB_EMPREGADO AS E ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
ORDER BY 2
```

	DEPARTAMENTO	NOME
1	TREINAMENTO	NULL
2	CONTROLADORIA	NULL
3	RECURSOS HUMANOS	ALBERTO
4	DIRETORIA	ALTAMIR
5	PRODUCAO	ANA
6	CONTROLE DE ESTOQUE	ANA DO CARMO
7	CONTROLE DE ESTOQUE	ANA MARIA
8	PRODUCAO	ANTONIO
9	DIRETORIA	ARLINDO
10	CONTROLE DE ESTOQUE	ARMANDO

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

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

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

```
SELECT D.DEPARTAMENTO , E.NOME
FROM TB_DEPARTAMENTO AS D
LEFT JOIN TB_EMPREGADO AS E ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
ORDER BY 2
```

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

```
SELECT D.DEPARTAMENTO , CA.NOME
FROM TB_DEPARTAMENTO AS D
OUTER APPLY
(SELECT E.NOME FROM TB_EMPREGADO AS E WHERE E.ID_DEPARTAMENTO
= D.ID_DEPARTAMENTO ) 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 do capítulo.

- **CASE**, **IIF** e **CHOOSE** são recursos para condicionar o resultado de um determinado campo ou valor;
- Com **LAG** e **LEAD** é possível analisar as linhas anteriores e posteriores de uma determinada consulta;
- 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 gerar 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;
- **CROSS APPLY** e **OUTER APPLY** possuem funcionalidades parecidas com **INNER JOIN** e **LEFT JOIN**. Em determinados casos, possuem uma melhor performance.



Comandos adicionais

Teste seus conhecimentos



1. Verifique o comando a seguir e responda: Qual a alternativa correta?

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

- ☐ a) No campo SINDICATO, será mostrado S, N ou N/C.
- ☐ b) Quando o campo Sindicalizado for nulo, o retorno é nulo.
- ☐ c) A sintaxe está errada e retornará um erro.
- ☐ d) Quando o campo Sindicalizado for nulo, o retorno é N/C.
- ☐ e) Nenhuma das alternativas anteriores está correta.

2. Qual comando adiante não condiciona um valor?

- ☐ a) UNION
- ☐ b) CASE
- ☐ c) CHOOSE
- ☐ d) IIF
- ☐ e) As respostas B, C e D estão corretas.

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 T-SQL.

4. Verifique o comando adiante e responda: Qual a alternativa correta?

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

- ☐ a) O comando retorna a união das duas consultas.
- ☐ b) Une as duas consultas retirando os valores duplicados.
- ☐ c) O comando retorna a união das duas consultas mesmo com valores duplicados.
- ☐ d) Gera um erro de sintaxe.
- ☐ e) Não podemos usar a cláusula ALL.

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

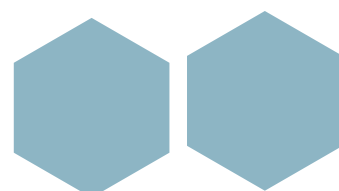
- ☐ a) Utilizando o UNPIVOT.
- ☐ b) Por meio do comando PIVOT ou UNPIVOT.
- ☐ c) Utilizando o PIVOT.
- ☐ d) Utilizando tabela temporária com UNION.
- ☐ e) O melhor é utilizar outra ferramenta, como o EXCEL.



Comandos adicionais



Mãos à obra!



Laboratório 1

A – Trabalhando com comandos adicionais

1. Coloque em uso o banco de dados **db_Ecommerce**;
2. Realize uma consulta na tabela de clientes, apresentando o nome do cliente, o código e o tipo de cliente (C para cliente ou R para revenda);
3. Faça uma consulta: ID do pedido, Nome do cliente, Valor total e um campo informando 'Vendas de 2016' para as vendas do ano de 2016;
4. Faça uma consulta: ID do pedido, Nome do cliente, Valor total e um campo informando 'Vendas de 2017' para as vendas do ano de 2017;
5. Realize uma união das consultas dos passos 3 e 4.

Laboratório 2

A – Trabalhando com comandos adicionais II

1. Coloque em uso o banco de dados **db_Ecommerce**;
2. Desenvolva uma consulta que retorne o estado do cliente, valor total de compra e o mês de todas as compras do ano de 2018;
3. Realize uma rotação da consulta para apresentar a soma dos meses por estado;
4. Utilizando a consulta anterior, acrescente a cidade do cliente.

Laboratório 3

A – Trabalhando com comandos adicionais III

1. Coloque em uso o banco de dados **db_Ecommerce**;
2. Realize uma consulta que apresente as informações adiante:
 - Código, nome, número do pedido, valor total e estado do cliente;
 - Quantos pedidos o 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 o 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 2018;
 - Ordene por nome do cliente e número de pedido.

2

Dados temporários

- Tabela temporárias;
- SELECT INTO;
- Subconsultas e tabelas temporárias;
- COMMON TABLE EXPRESSIONS (CTE).



2.1. Tabela temporárias

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

A seguir, um exemplo de criação de tabela temporária local (lembrando que a criação de tabelas será vista no módulo II):

```
CREATE TABLE #MATRICULA
(
    NUM_MATRICULA INTEGER NOT NULL ,
    DAT_MATRICULA DATETIME NOT NULL ,
    VAL_MATRICULA DECIMAL (12,2) NOT NULL ,
    COD_ALUNO INTEGER NOT NULL,
    CONSTRAINT MATRICULA_PK PRIMARY KEY CLUSTERED (NUM_
MATRICULA)
)
```

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

Vejamos um exemplo de criação de tabela temporária global:

```
CREATE TABLE ##MATRICULA
(
    NUM_MATRICULA INTEGER NOT NULL ,
    DAT_MATRICULA DATETIME NOT NULL ,
    VAL_MATRICULA DECIMAL (12,2) NOT NULL ,
    COD_ALUNO INTEGER NOT NULL,
    CONSTRAINT MATRICULA_PK PRIMARY KEY CLUSTERED (NUM_
MATRICULA)
)
```

2.2.SELECT INTO

A criação de uma tabela é realizada através do comando **CREATE TABLE**, porém também é possível a criação de uma tabela através da cláusula **INTO**. Verifique o exemplo adiante em que é criada uma tabela temporária local com as informações **Nome do Cliente**, **Data do pedido**, **Número do pedido** e **Valor total da compra**:

- Consulta base:

```
SELECT C.NOME, P.DATA_EMISSAO, P.ID_PEDIDO, P.VLR_TOTAL
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

- Introduzindo a cláusula **INTO**:

```
--Utilizando a cláusula INTO
SELECT C.NOME, P.DATA_EMISSAO, P.ID_PEDIDO, P.VLR_TOTAL
INTO #Pedidos_CLIENTE
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

O resultado não é apresentado, somente a quantidade de registros inseridos:

Messages
(11322 rows affected)

Para visualizar as informações, é necessária consulta na tabela temporária:

```
SELECT * FROM #Pedidos_CLIENTE
```

	NOME	DATA_EMISSAO	ID_PEDIDO	VLR_TOTAL
1	TICMQXMWLDDWLHBBHDQPUQQXUGFRV	2019-07-29 00:00:00.000	6049	2030.31
2	EKVRSGEANLSJMPREVLNCOHYHNPDI	2019-07-29 00:00:00.000	6050	135.36
3	RWMJEENTDJDURDICIWRWVDTMUCMBE	2019-07-30 00:00:00.000	6051	1376.19
4	FKFOLKXCRYFELGMCBJTMOBHEMWHJB	2019-07-31 00:00:00.000	6052	6357.35
5	RFWBVEWSOGAVSFRTRPTTKXLHLMNEG	2019-07-31 00:00:00.000	6053	3467.52
6	BWEJBMDLFPNFMONXSVNHVWQBBWAULK	2019-08-01 00:00:00.000	6054	901.06
7	HYQFPWJTFQWKHEVLSELATXOASMKGSC	2019-08-01 00:00:00.000	6055	8178.28
8	LNXYNOHPSRBJYTBBDFYNCWPQFTMQJL	2019-08-01 00:00:00.000	6056	3481.60
9	NVOSWXQBWDCEXRYUBCMWFEJBIUCBU	2019-08-01 00:00:00.000	6057	2552.56

É importante ter em mente que é necessário possuir o nome das colunas distintas para não ter erro de execução:

```
SELECT C.NOME, C.NOME, P.DATA_EMISSAO , P.ID_PEDIDO , P.VLR_
TOTAL
INTO #Pedidos_CLIENTE2
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

Messages

Msg 2705, Level 16, State 3, Line 1246

Column names in each table must be unique. Column name 'NOME' in table '#Pedidos_CLIENTE2' is specified more than once.

2.3. Subconsultas e tabelas temporárias

Embora as tabelas temporárias sejam parecidas com as permanentes, elas são armazenadas em **TEMPDB** e excluídas automaticamente após terem sido utilizadas.

As tabelas temporárias locais apresentam, antes do nome, o símbolo # e são visíveis somente durante a conexão atual. Quando o usuário se desconecta da instância do SQL Server, ela é excluída.

Já as tabelas temporárias globais apresentam, antes do nome, dois símbolos ## e são visíveis para todos os usuários. Uma tabela desse tipo será excluída apenas quando todos os usuários que a referenciam se desconectarem da instância do SQL Server.

A escolha da utilização de tabelas temporárias ou de subconsultas dependerá de cada situação e de aspectos como desempenho do sistema e até mesmo das preferências pessoais de cada usuário. O fato é que, por conta das diferenças existentes entre elas, o uso de uma, para uma situação específica, acaba sendo mais indicado do que o emprego de outra.

Assim, quando temos bastante memória no servidor, as subconsultas são preferíveis, pois ocorrem na memória. Já as tabelas temporárias, como necessitam dos recursos disponibilizados pelo disco rígido para serem executadas, são indicadas nas situações em que o(s) servidor(es) do banco de dados apresenta(m) bastante espaço no disco rígido.

Há, ainda, uma importante diferença entre tabela temporária e subconsulta: normalmente, esta última é mais fácil de manter. No entanto, se a subconsulta for muito complexa, a melhor medida a ser tomada pode ser fragmentá-la em diversas tabelas temporárias, criando, assim, blocos de dados de tamanho menor.

Veja o exemplo a seguir, que realiza a consulta com subconsulta e com tabelas temporárias. O resultado esperado é o retorno dos clientes que realizaram em 2017 a última compra, valor e data de emissão:

- Usando subconsulta:

```
SELECT C.NOME, P.ID_PEDIDO , P.DATA_EMISSAO, P.VLR_TOTAL
FROM TB_PEDIDO AS P
JOIN
(SELECT MAX(ID_PEDIDO) AS ULT_PEDIDO, ID_CLIENTE
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO)= 2017
GROUP BY ID_CLIENTE) AS PP ON P.ID_PEDIDO = PP.ULT_PEDIDO
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

- Usando tabela temporária:

```
--Criação de tabela temporária
SELECT MAX(ID_PEDIDO) AS ULT_PEDIDO, ID_CLIENTE
INTO #PEDIDO
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO)= 2017
GROUP BY ID_CLIENTE
```

```
--Consulta realizando JOIN com a tabela temporária.
SELECT C.NOME, P.ID_PEDIDO , P.DATA_EMISSAO, P.VLR_TOTAL
FROM TB_PEDIDO AS P
JOIN #PEDIDO AS PP ON P.ID_PEDIDO = PP.ULT_PEDIDO
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

	NOME	ID_PEDIDO	DATA_EMISSAO	VLR_TOTAL
1	SQSKRGYHTBLNWIAREAKTKDQPWQFQLE	8626	2017-12-10 00:00:00.000	275.32
2	QCQQBPNPKNHDSFBMEKESJNMJMJDOP	8118	2017-06-26 00:00:00.000	4079.47
3	TTOMDCPSXCWRXSQWVNOVURRENQDFKL	8490	2017-10-26 00:00:00.000	6071.25
4	ANXTUDDIVPWQUVINKNFBVLEOSUODXI	8538	2017-11-30 00:00:00.000	1648.82
5	TCSGVJISYSISLFFELSMILYSBMPRQNK	8540	2017-11-12 00:00:00.000	7170.78
6	QXVTRVGAQBVBCAAQIGMMHNFIFVJH	2298	2017-10-24 00:00:00.000	3123.66
7	OVRUUMSLRNTIUVGPCVLHFJQTKRSDKD	8052	2017-05-31 00:00:00.000	6790.93
8	RDVNRMHFEIRJCATCTAQDFLBHMHMUIQ	1306	2017-09-16 00:00:00.000	1270.36
9	EAPFLIPYEWEXBAKTVOHLPVXWKOCBVR	7836	2017-03-20 00:00:00.000	2494.58

2.4.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 uma subconsulta 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 2018:

```
SELECT
MONTH(DATA_EMISSAO) AS MES ,
YEAR(DATA_EMISSAO) AS ANO,
MAX(VLR_TOTAL) AS MAIORVALOR
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO)= 2018
GROUP BY
MONTH(DATA_EMISSAO),
YEAR(DATA_EMISSAO)
ORDER BY 1
```


	MES	ANO	MAIORVALOR
1	1	2018	7297.47
2	2	2018	12340.67
3	3	2018	8787.02
4	4	2018	9103.67
5	5	2018	9138.45
6	6	2018	10018.13
7	7	2018	11307.32
8	8	2018	9237.65
9	9	2018	9977.20
10	10	2018	10647.69
11	11	2018	11206.01
12	12	2018	9259.84

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 2018. 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) = 2018
  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.ID_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;
```

2.4.1. 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:

```
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	1	5
2	2	25
3	3	125
4	4	625
5	5	3125
6	6	15625
7	7	78125
8	8	390625
9	9	1953125
10	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	144	233
13	144	233	377
14	233	377	610

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:

```
WITH CTE( CODFUN, NOME, ID_DEPARTAMENTO, CODSUP, NOME_SUP )
AS
(
    -- Membro âncora
    SELECT ID_EMPREGADO, NOME, ID_DEPARTAMENTO, COD_
SUPERVISOR, NOME
    FROM TB_Empregado WHERE COD_SUPERVISOR = 0
    UNION ALL
    -- Membro recursivo
    SELECT E.ID_CARGO, E.NOME, E.ID_DEPARTAMENTO, 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.DEPARTAMENTO AS [Departamento], CTE.NOME_SUP AS [Nome
Supervisor]
FROM CTE JOIN TB_EMPREGADO E ON CTE.CODFUN = E.ID_CARGO
JOIN TB_DEPARTAMENTO D ON E.ID_DEPARTAMENTO = D.ID_
DEPARTAMENTO
ORDER BY CTE.ID_DEPARTAMENTO;
```

	Código	Funcionário	Departamento	Nome Supervisor
1	6	PEDRO	PRODUCAO	CARLOS ALBERTO
2	6	PEDRO	PESSOAL	CARLOS ALBERTO
3	6	PEDRO	COMPRAS	CARLOS ALBERTO
4	6	PEDRO	PESSOAL	CARLOS ALBERTO
5	1	SEBASTIÃO	PESSOAL	CARLOS ALBERTO
6	1	SEBASTIÃO	TI	CARLOS ALBERTO
7	1	SEBASTIÃO	PESSOAL	CARLOS ALBERTO
8	16	MARIANA	CONTROLE DE ESTOQUE	CARLOS ALBERTO
9	6	LÚCIO	PRODUCAO	CARLOS ALBERTO
10	6	LÚCIO	PESSOAL	CARLOS ALBERTO
11	6	LÚCIO	COMPRAS	CARLOS ALBERTO
12	6	LÚCIO	PESSOAL	CARLOS ALBERTO
13	10	ANA	PRODUCAO	CARLOS ALBERTO
14	10	ANA	PRODUCAO	CARLOS ALBERTO



Pontos principais

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

- Tabelas temporárias são úteis para armazenar dados que serão utilizados posteriormente sem a necessidade de criar tabelas físicas;
- Podemos criar tabelas temporárias a partir do comando **CREATE TABLE** e **SELECT INTO**;
- Tabelas temporárias podem ser locais e globais;
- **Common table expression (CTE)** é 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 esses motivos, parecidas com as tabelas derivadas.

Bruna C
397.642.208-108



Dados temporários

Teste seus conhecimentos



1. Como podemos criar uma tabela temporária?

- ☐ a) Através da cláusula SELECT INTO.
- ☐ b) Não podemos criar uma tabela temporária.
- ☐ c) Tabelas temporárias somente podem ser criadas pelo administrador.
- ☐ d) Através do comando CREATE TABLE TEMP.
- ☐ e) Com o SELECT.

2. Uma tabela temporária pode ser local ou global. Verifique as afirmações adiante e assinale a correta:

- ☐ a) Temporárias globais iniciam com #.
- ☐ b) Temporárias locais iniciam com ##.
- ☐ c) Temporárias locais podem ser iniciadas com TMP.
- ☐ d) É necessário realizar um comando DROP TABLE para apagar um tabela temporária.
- ☐ e) Tabelas temporárias globais iniciam com ##.

3. Verifique o comando a seguir:

```
WITH CTE( MES, ANO, MAIOR_PEDIDO )
AS
(
SELECT MONTH( DATA_EMISSAO ) AS MES,
       YEAR( DATA_EMISSAO ) AS ANO,
       MAX( VLR_TOTAL ) AS MAIOR_PEDIDO
FROM TB_PEDIDO
WHERE YEAR(DATA_EMISSAO) = 2018
GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
)
SELECT CTE.MES, CTE.ANO, CTE.MAIOR_PEDIDO, P.ID_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;
```

Qual a função do comando da CTE?

- ☐ a) Armazenar informações de pedidos.
- ☐ b) Armazenar as informações do maior pedido por ano e mês de 2018.
- ☐ c) Uma CTE tem a função de armazenar pedidos.
- ☐ d) Os dados do maior pedido.
- ☐ e) Realizar um JOIN com a tabela TB_PEDIDO.

4. O que o comando adiante realiza?

```
SELECT      C.NOME, P.DATA_EMISSAO , P.ID_PEDIDO , P.VLR_TOTAL
INTO #Pedidos_CLIENTE
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

- ☐ a) Cria uma tabela temporária local.
- ☐ b) Cria uma tabela temporária global.
- ☐ c) Realiza uma consulta na tabela TB_PEDIDO, apresenta os dados e cria uma tabela temporária.
- ☐ d) Executa a criação de uma tabela.
- ☐ e) Apresenta o nome, data de emissão, id do pedido e valor total do pedido.

5. Verifique o comando a seguir:

```
SELECT C.NOME, P.ID_PEDIDO , P.DATA_EMISSAO, P.VLR_TOTAL  
FROM TB_PEDIDO AS P  
JOIN  
(SELECT MAX(ID_PEDIDO) AS ULT_PEDIDO, ID_CLIENTE  
FROM TB_PEDIDO  
WHERE YEAR(DATA_EMISSAO)= 2017  
GROUP BY ID_CLIENTE) AS PP ON P.ID_PEDIDO = PP.ULT_PEDIDO  
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
```

Assinale a afirmação correta:

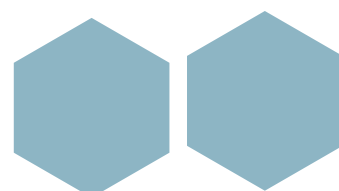
- ☐ a) O correto é usar uma CTE.
- ☐ b) O correto é usar uma tabela ou view.
- ☐ c) Crie uma tabela temporária para ganhar performance.
- ☐ d) A cláusula FROM com a subconsulta pode ser substituída por uma CTE ou tabela temporária.
- ☐ e) A sintaxe está errada.



Dados temporários



Mãos à obra!





Laboratório 1

A - Criando tabelas temporárias

1. Coloque o banco **db_Ecommerce** em uso;
2. Crie uma tabela temporária com as vendas de julho de 2018;
3. Consulte a tabela temporária e conte quantos pedidos foram realizados;
4. Crie uma tabela temporária com as informações: Número do pedido, Nome do cliente, Nome do vendedor, Data da emissão e Valor total;
5. Filtre as informações da tabela temporária criada no passo 4, mostrando apenas os dados de setembro de 2019;
6. Realize um **JOIN** com a tabela de pedidos para apresentar a observação.

Laboratório 2

A - Utilizando resultados temporários

1. Utilizando uma CTE, realize uma consulta que apresente os campos: ID_PEDIDO, Nome do cliente, Valor total e o último número do pedido do cliente referente ao período de março de 2017;
2. Utilizando uma CTE, realize uma consulta que apresente a soma dos valores por produtos vendidos de cor AZUL que foram vendidos em 2018. Apresente o produto e o valor.

3

Dados espaciais

- Resultado espacial;
- Tipos de dados geográficos.



3.1. Introdução

Dados espaciais são muito utilizados em armazenamento de informações para geoprocessamento, onde podemos usar as informações estruturadas das tabelas com informações e visualizar em mapas e formas geométricas.

Utiliza-se um sistema de referência espacial (**SRS**) ou sistema de coordenadas (**CRS**), que é uma coordenada local, regional ou global.

O SQL possui dois tipos de dados:

- **GEOMETRY** (Geometria): Coordenadas euclidianas (dados planos). O SQL utiliza o recurso de **OGC** (Open Geospatial Consortium);
- **GEOGRAPHY** (Geografia): Coordenadas esféricas.

3.2. Resultado espacial

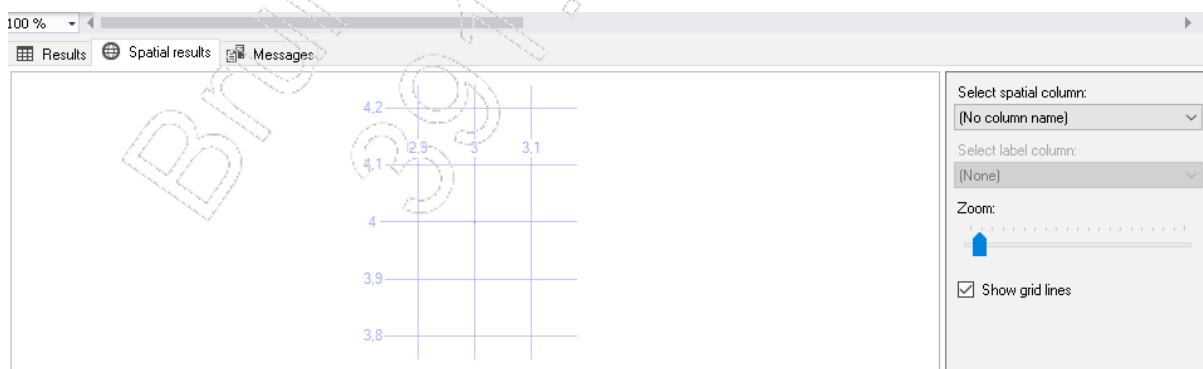
Ao executar um comando espacial, o SSMS retorna o resultado de forma binária e abre um resultado gráfico espacial. Veja o exemplo:

```
select geometry::STGeomFromText('POINT (3 4)', 0)
```

- Resultado binário:



- Resultado gráfico:



O resultado de formas espaciais é apresentado na aba **Results** e **Spatial results**.

3.3. Tipos de dados geográficos

Vejamos, a seguir, os tipos de dados geográficos:

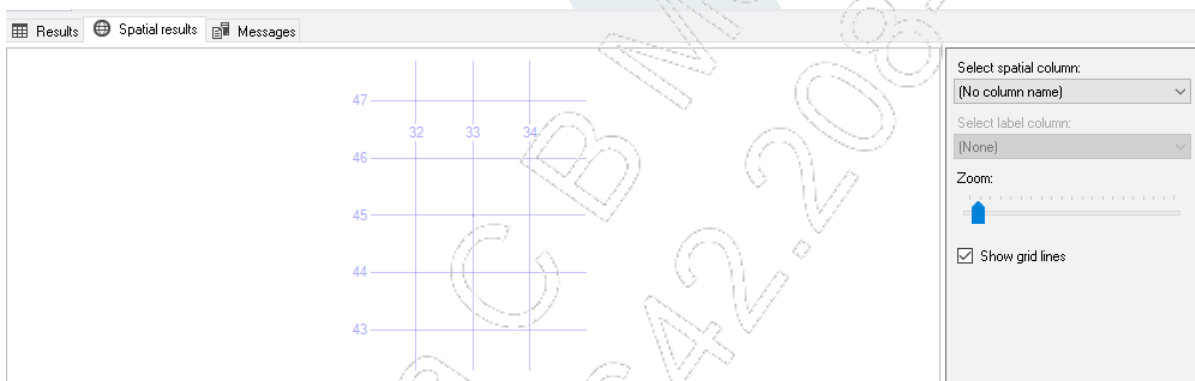
- **Point**

Representa um local único de uma longitude.

Veja alguns exemplos:

```
SELECT geometry::STGeomFromText('POINT (33 45)', 0);
```

	Results	Spatial results	Messages
	(No column name)		
1	0x00000000010C00000000008040400000000000804640		



No próximo exemplo, será armazenado um valor em uma variável e depois extraídas as posições X, Y, Z e M:

```
declare @g geometry
SET @g = geometry::Parse('POINT(4 7 3 1)');

SELECT @g.STX;
SELECT @g.STY;
SELECT @g.Z;
SELECT @g.M;
```

Results		Messages
(No column name)		
1	4	
(No column name)		
1	7	
(No column name)		
1	3	
(No column name)		
1	1	

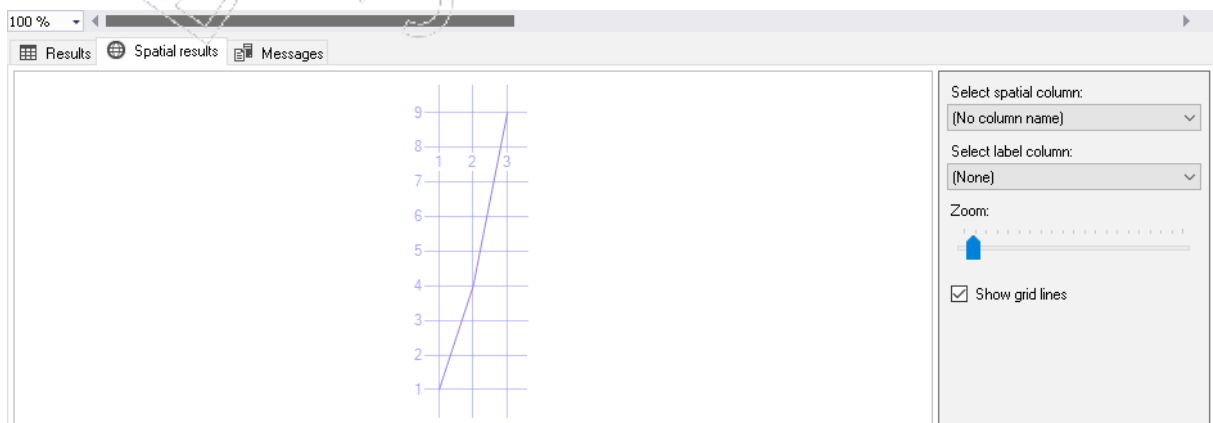
- **LineString**

Apresenta uma figura linear. Pode ser:

- Simples não fechada;
- Não simples não fechada;
- Fechada simples;
- Fechada não simples.

```
Select geometry::STGeomFromText('LINESTRING(1 1, 2 4, 3 9)',
```

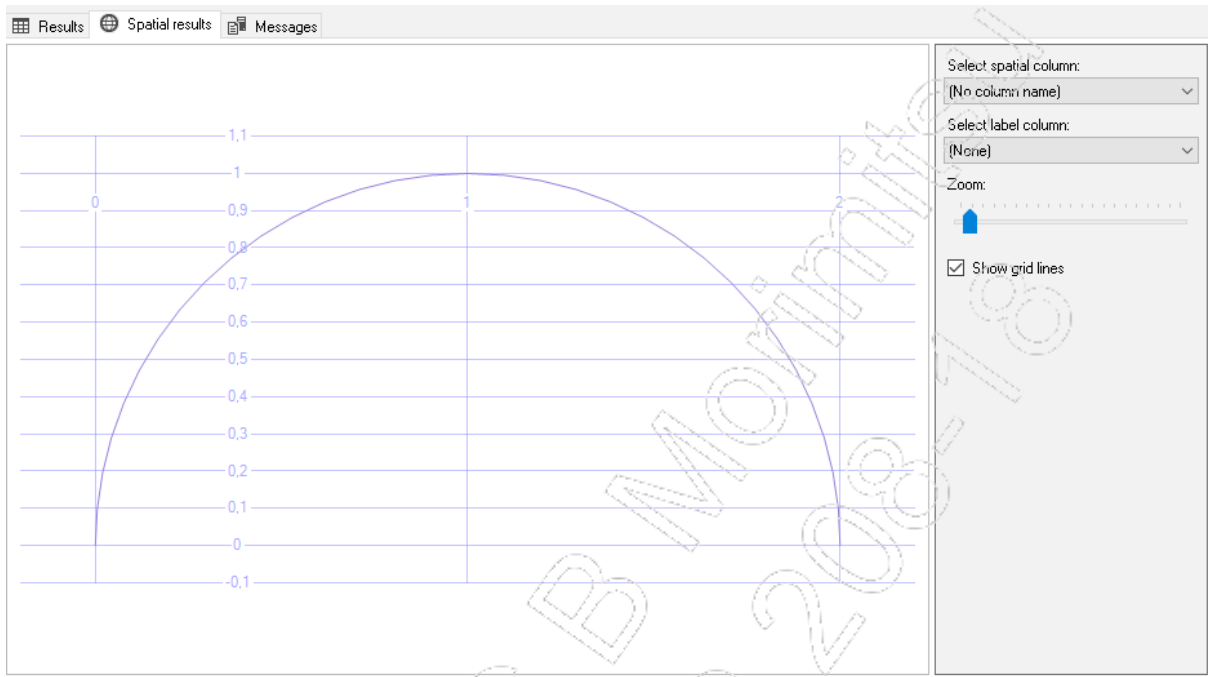
```
0);
```



- **CircularString**

Representa uma figura circular.

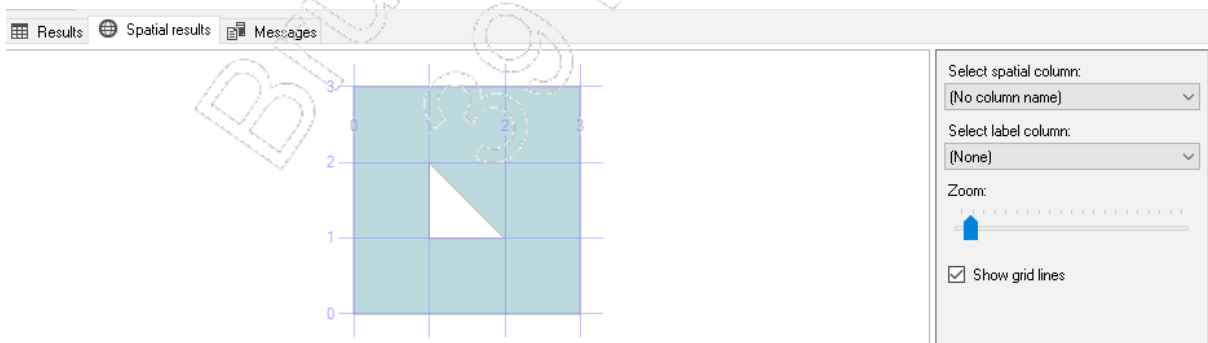
```
select geometry:: STGeomFromText('CIRCULARSTRING(2 0, 1 1, 0  
0)', 0);
```



- **Polígono**

Representa uma figura de um polígono.

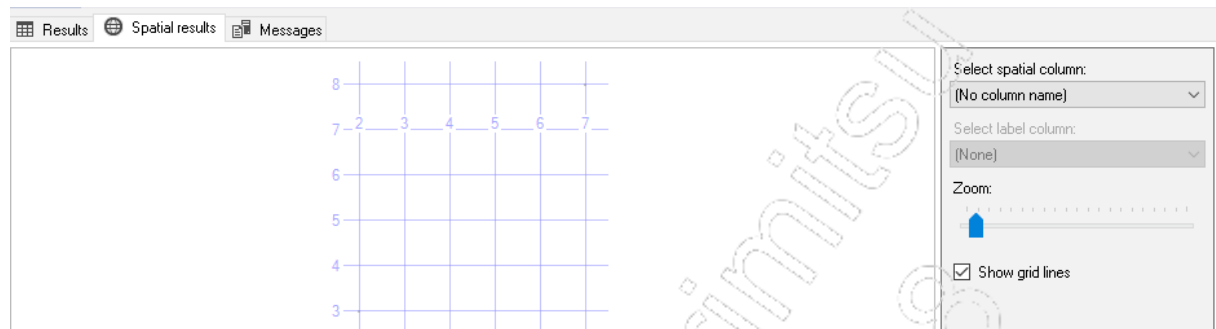
```
SELECT geometry::STPolyFromText('POLYGON((0 0, 0 3, 3 3, 3 0,  
0 0), (1 1, 1 2, 2 1, 1 1))', 10);
```



- **MultiPoint**

É um conjunto de vários pontos.

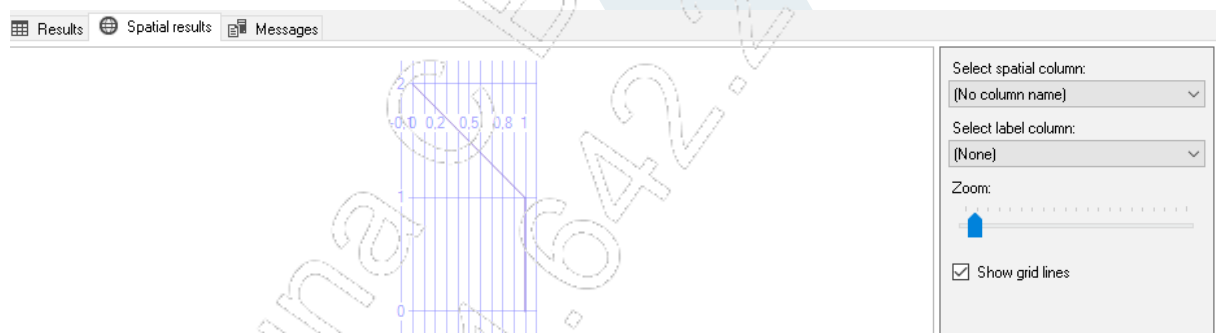
```
select geometry::STPointFromText('MULTIPOINT((2 3), (7 8 9.5))', 23);
```



- **MultiLineString**

Representa várias linhas de uma figura.

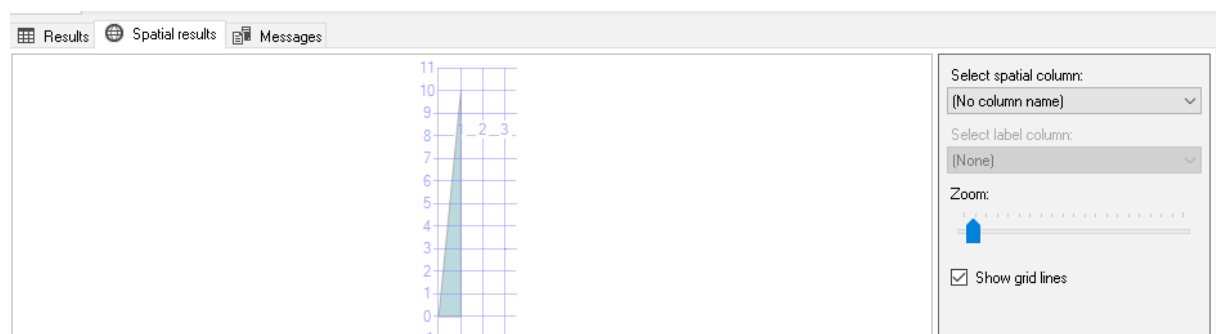
```
select geometry::Parse('MULTILINESTRING((0 2, 1 1), (1 0, 1 1))');
```



- **GeometryCollection**

É uma coleção de figuras geométricas ou geográficas.

```
select geometry::STGeomCollFromText('GEOMETRYCOLLECTION(POINT(3 3 1), POLYGON((0 0 2, 1 0 3, 1 0 4, 0 0 2)))', 1);
```



Pontos principais

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

- O SQL Server permite a utilização de dados espaciais. Dados espaciais podem ser geométricos e geográficos;
- Utiliza-se um sistema de referência espacial (**SRS**) ou sistema de coordenadas (**CRS**), que é uma coordenada local, regional ou global;
- Existem várias funções que permitem a utilização de formas geográficas;
- O resultado de formas espaciais é apresentado na aba **Results** e **Spatial results**.



Dados espaciais

Teste seus conhecimentos



1. Onde podemos usar dados espaciais?

- ☐ a) Em qualquer consulta.
- ☐ b) Somente em consultas com dados XML.
- ☐ c) Para armazenamento de dados para geoprocessamento.
- ☐ d) Em consultas com JSON.
- ☐ e) Nenhuma das alternativas anteriores está correta.

2. Qual tipo de dado é espacial?

- ☐ a) GEOMETRY
- ☐ b) CARTESIANO
- ☐ c) COORDENADA
- ☐ d) ESPATIAL
- ☐ e) GEOCENTRIC

3. Verifique o comando a seguir:

```
SELECT geometry::STGeomFromText('POINT (33 45)', 0);
```

Como podemos verificar o retorno da consulta?

- ☐ a) Exportando para o Excel.
- ☐ b) Através de software próprio.
- ☐ c) O SQL somente armazena os dados.
- ☐ d) Usando o Spatial graphics.
- ☐ e) Através da aba Spatial results.

4. Qual a função do CircularString?

- ☐ a) Retorna uma figura.
- ☐ b) Retorna uma figura circular.
- ☐ c) Gera dados para uma figura.
- ☐ d) Plota valores para o Spatial results.
- ☐ e) Não existe essa função.

5. O que é o OGC?

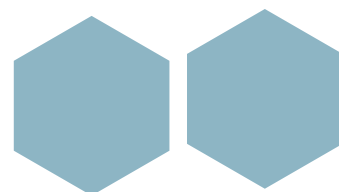
- ☐ a) Mecanismo de busca de dados espaciais.
- ☐ b) Função de conversão de dados espaciais.
- ☐ c) Função de conversão de dados geométricos para geográficos.
- ☐ d) Sigla para Open Geospatial Consortium.
- ☐ e) Função de conversão de dados geográficos para geométricos.



Dados espaciais



Mãos à obra!





Laboratório 1

A - Realizando uma consulta geométrica

1. Faça uma consulta geométrica marcando o ponto: 22, 72. Utilize a função **POINT**;
2. Faça uma consulta geométrica triangular com os valores: 1 1, 2 4, 4 2, 1 1. Utilize a função **LINESTRING**;
3. Apresente os pontos de X, Y, Z e M para o ponto: 10, 15,2 ,1;
4. Apresente uma figura circular para os pontos: 3 0, 2 2 , 1 1.

Bruna C
391.642.208-18
B Morimatsu

4

Views

- Tipos de views;
- Vantagens;
- Restrições;
- Criando uma VIEW;
- ALTER VIEW;
- DROP VIEW;
- Visualização de informações sobre VIEWS;
- VIEWS atualizáveis;
- Retorno de dados tabulares.

4.1. Introdução

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

A consulta 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 consultas distribuídas.

É permitida a realização de quaisquer consultas e até mesmo a alteração de dados por meio de VIEWS.

4.2. Tipos de 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.

4.3. Vantagens

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 consulta 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 poupá-los do trabalho de escrever várias consultas **SELECT** complexas.

4.4. 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.

4.5. Criando uma 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 DB_ECOMMERCE
GO
-- CRIANDO A VIEW
CREATE VIEW VW_EMPREGADO AS
SELECT ID_EMPREGADO, NOME, DATA_ADMISSAO,
       ID_DEPARTAMENTO, ID_CARGO, SALARIO
FROM TB_EMPREGADO
GO

-- Testando a VIEW
SELECT * FROM VW_EMPREGADO
--
SELECT ID_EMPREGADO, NOME FROM VW_EMPREGADO
```

4.5.1.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 VW_EMPREGADO_ENCRY WITH ENCRYPTION
AS
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO,
       ID_CARGO, SALARIO
FROM TB_EMPREGADO
GO

SELECT * FROM VW_EMPREGADO_ENCRY
```

Ao realizar a consulta na **SYSCOMMENTS**, o campo **TEXT** está nulo:

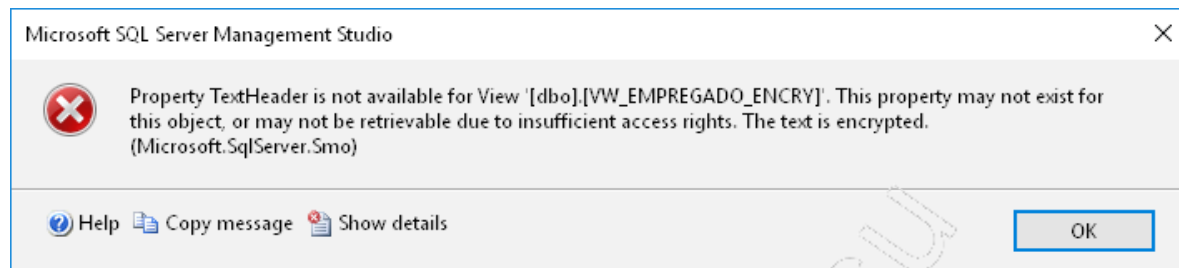
```
SELECT * FROM SYSCOMMENTS
WHERE ID= OBJECT_ID('VW_EMPREGADO_ENCRY')
```

Results		Messages								
	id	number	colid	status	ctext	texttype	language	encrypted	compressed	text
1	1451152215	0	1	1	NULL	6	0	1	0	NULL

Podemos observar no **Object Explorer** que não é possível ter acesso ao código fonte da VIEW:



Também não é possível gerar o script:



4.5.2. 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 VW_EMPREGADO_SCHEMA
WITH ENCRYPTION, SCHEMABINDING
AS
SELECT ID_EMPREGADO, NOME, DATA_ADMISSAO,
       ID_DEPARTAMENTO, ID_CARGO, SALARIO, NUM_DEPEND
FROM DBO.TB_EMPREGADO
GO

-- Testando a VIEW
SELECT * FROM VW_EMPREGADO_SCHEMA
```


Tentando realizar uma alteração na tabela **TB_EMPREGADO**:

```
ALTER TABLE TB_EMPREGADO DROP COLUMN NUM_DEPEND
```

Messages

```
Msg 5074, Level 16, State 1, Line 158
The object 'VW_EMPREGADO_SCHEMA' is dependent on column 'NUM_DEPEND'.
Msg 4922, Level 16, State 9, Line 158
ALTER TABLE DROP COLUMN NUM_DEPEND failed because one or more objects access this column.
```

4.5.3. 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:

```
CREATE VIEW VW_EMPREGADO_OPTION WITH ENCRYPTION
AS
SELECT ID_EMPREGADO, NOME, DATA_ADMISSAO,
       ID_DEPARTAMENTO, ID_CARGO, SALARIO
FROM TB_EMPREGADO
WHERE ID_DEPARTAMENTO = 2
GO
-- Testando
SELECT * FROM VW_EMPREGADO_OPTION
```

A view apresentada mostrará apenas funcionários que trabalham no departamento de código 2, como podemos visualizar adiante:

	ID_EMPREGADO	NOME	DATA_ADMISSAO	ID_DEPARTAMENTO	ID_CARGO	SALARIO
1	2	JOSE	2017-07-24 00:00:00.000	2	5	660.00
2	9	RUDGE	2015-01-26 00:00:00.000	2	4	880.00
3	19	SEBASTIÃO	2016-12-23 00:00:00.000	2	1	9130.00
4	20	EURICO	2010-07-20 00:00:00.000	2	4	880.00
5	25	ROBERTO MARIA	2010-04-15 00:00:00.000	2	11	4950.00
6	28	MARIANO	2016-04-18 00:00:00.000	2	9	3663.00
7	38	LUIS	2018-07-13 00:00:00.000	2	14	660.00
8	40	JOAQUIM	2018-02-03 00:00:00.000	2	5	550.00

No SQL Server, podemos alterar uma tabela através de uma view, mediante algumas condições vistas a seguir:

```
INSERT INTO VW_EMPREGADO_OPTION  
( NOME, DATA_ADMISSAO, ID_DEPARTAMENTO, id_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 satisfaçam a condição de filtro. Isso é feito com a instrução **WITH CHECK OPTION**:

```
ALTER VIEW VW_EMPREGADO_OPTION WITH ENCRYPTION  
AS  
SELECT ID_EMPREGADO, NOME, DATA_ADMISSAO,  
       ID_DEPARTAMENTO, ID_CARGO, SALARIO  
FROM TB_EMPREGADO  
WHERE ID_DEPARTAMENTO = 2  
WITH CHECK OPTION
```

Ao executarmos o comando novamente, o SQL retornará um erro:

```
INSERT INTO VW_EMPREGADO_OPTION  
( NOME, DATA_ADMISSAO, ID_DEPARTAMENTO, id_CARGO, SALARIO)  
VALUES ( 'TESTE INCLUSÃO', GETDATE(), 1, 1, 1000)
```

```
Messages  
Msg 550, Level 16, State 1, Line 193  
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION.  
The statement has been terminated.
```

4.5.4. Criando índices

Uma VIEW não ocupa espaço em disco, porém os índices vão consumir espaço em disco.

Para criarmos um índice, é obrigatória a utilização da cláusula **SCHEMABINDING**. Também é necessário que o primeiro índice seja do tipo **clusterizado**.

```
-- Criando índices para a view  
CREATE UNIQUE CLUSTERED INDEX IX_VIE_EMP3_CODFUN  
ON VW_EMPREGADO_SCHEMA(ID_EMPREGADO)  
GO  
--  
CREATE INDEX IX_VIE_EMP3_NOME  
ON VW_EMPREGADO_SCHEMA(NOME)
```

4.6.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**.

```
ALTER VIEW VW_EMPREGADO_ENCRY WITH ENCRYPTION  
AS  
SELECT ID_DEPARTAMENTO, NOME, DATA_ADMISSAO,  
       ID_CARGO, SALARIO  
FROM TB_EMPREGADO  
GO
```

4.7. 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.

```
DROP VIEW VW_EMPREGADO_ENCRY
```

4.8. 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.

4.9. 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.

4.10. 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
-----

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) = 2016
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.ID_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.ID_CLIENTE = C.ID_CLIENTE

```

Uma view não pode receber parâmetros e não utiliza variáveis e comandos de programação, portanto sempre retornará todos os registros da consulta.

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 2016  
EXEC STP_MAIOR_PEDIDO 2017  
EXEC STP_MAIOR_PEDIDO 2018  
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( '2017.1.1', '2017.12.31' )  
ORDER BY ANO, MES  
-- Aceita filtro  
SELECT * FROM DBO.FN_MAIOR_PEDIDO( '2017.1.1', '2017.12.31' )  
WHERE MES > 6  
ORDER BY ANO, MES  
-- Aceita JOIN  
SELECT F.MES, F.ANO, F.MAIOR_VALOR, P.NUM_PEDIDO  
FROM FN_MAIOR_PEDIDO( '2017.1.1', '2017.12.31' ) 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 do capítulo.

- Uma **view** é uma tabela virtual formada por linhas e colunas de dados provenientes de tabelas referenciadas em uma consulta;
- 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





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?

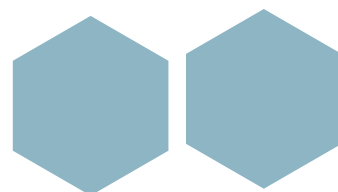
- ☐ a) Não é possível atualizar registros a partir de uma view.
- ☐ b) As colunas devem pertencer a uma mesma tabela.
- ☐ c) As colunas não podem ser computadas.
- ☐ d) Colunas que não são especificadas na view devem permitir valores nulos ou possuir valores DEFAULT.
- ☐ e) Colunas não serão alteradas se forem utilizadas as cláusulas GROUP BY, HAVING ou DISTINCT.



Views



Mãos à obra!



Laboratório 1

A – Criando views e consultando as tabelas virtuais criadas pela view

1. Coloque em uso o banco de dados **DB_ECOMMERCE**;
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 2018. 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 2018. 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 2018 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_ITENSPEDIDO**, mais a data de emissão do pedido (**DATA_EMISSAO**), a descrição do produto (**DESCRIÇÃO**), 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 2018;
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**.

5

Funções nativas

- Funções;
- Funções de texto;
- Funções matemáticas;
- Funções de data e hora;
- Funções de conversão;
- Funções de classificação.



5.1. Funções

Uma **função (function)** é, basicamente, um objeto que contém um bloco de comandos T-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.

5.1.1. Funções determinísticas e não determinísticas

As funções, tanto nativas 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('IMPACTA TREINAMENTO')

-- 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()
```

5.2. Funções de texto

Estas funções executam operações em um valor de entrada de cadeia de caracteres e retornam o mesmo dado trabalhado. Por exemplo, podemos concatenar, replicar ou inverter os dados de entrada. A seguir, vamos apresentar as funções de cadeia de caracteres principais fornecidas pelo SQL Server:

Função	Descrição	Argumento
ASCII	Retorna o código ASC de um caractere.	Expressão texto
CHAR	Retorna o caractere correspondente a um determinado código.	Código ASCII
CHARINDEX	Retorna a posição de uma string dentro de outra.	Expressão pesquisa, expressão pesquisada
CONCAT	Concatena as expressões retornando uma string. As expressões podem ser de qualquer tipo.	expr1, expr2, exprN
CONCAT_WS	Concatena expressões com a opção de uso de separador.	Separador, expr1, expr2, exprN
DIFFERENCE	Compara valores diferentes e retorna um valor inteiro correspondente à semelhança entre valores.	Expressão texto, expressão texto
FORMAT	Formata um valor conforme um argumento de cultura.	Valor, Formato, Cultura
LEFT	Retorna N caracteres a partir da esquerda de um texto.	Expressão texto, quantidade de caracteres
LEN	Retorna a quantidade de caracteres de um texto.	Expressão texto
LOWER	Converte para minúsculo.	Expressão texto
LTRIM	Retira espaços em branco à esquerda.	Expressão texto
NCHAR	Retorna o caractere correspondente a um determinado código.	Valor inteiro
PATINDEX	Retorna a posição da primeira ocorrência do texto pesquisado.	Texto de pesquisa, Texto pesquisado

Função	Descrição	Argumento
QUOTENAME	Substitui uma expressão texto por um caractere: ',"', [], {}, ><.	Expressão texto, caractere quota
REPLACE	Realiza busca e substituição.	Expressão texto, texto para substituir, Texto substituído
REPLICATE	Repete um texto N vezes.	Expressão texto, quantidade de repetições
REVERSE	Retorna o reverso de um texto.	Expressão texto
RIGHT	Retorna N caracteres a partir da direita de um texto.	Expressão texto, quantidade de caracteres
RTRIM	Elimina espaços à direita de um texto.	Expressão texto
SOUNDEX	Retorna o código do som do texto.	Expressão texto
SPACE	Retorna uma quantidade de espaços.	Quantidade de espaços
STR	Esta função retorna dados do tipo texto a partir de dados numéricos.	número, tamanho, decimal
STRING_AGG	Agrega valores retornando uma única linha.	Expressão texto, separador WITHIN GROUP (ORDER BY <expressão de ordenação)
STRING_ESCAPE	Retorna um texto com argumento de ESCAPE.	Expressão texto, tipo
STRING_SPLIT	Retorna um conjunto de dados a partir de um separador.	Expressão texto, separador
STUFF	Insere uma cadeia de caracteres em outra.	Expressão texto, posição inicial, tamanho, expressão para substituir
SUBSTRING	Retorna N caracteres a partir de determinada posição de um texto.	Expressão, início, tamanho
TRANSLATE	Atualiza um texto a partir de outro.	Expressão texto, caracteres a serem substituídos, caracteres que vão substituir o texto
TRIM	Remove espaços em branco à esquerda e à direita.	Expressão texto
UNICODE	Retorna o valor unicode de código.	Expressão texto
UPPER	Converte para maiúsculo.	Expressão texto

Exemplos:

- ASCII

```
--- Código de um dado caractere
SELECT ASCII( 'A' )
SELECT ASCII( 'B' )
SELECT ASCII( '0' )
```

- CHAR

```
--- 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

```
--- Posição de uma string dentro de outra
SELECT CHARINDEX( 'PA', 'IMPACTA' )
SELECT CHARINDEX( 'Antonio', 'Antonio da Silva' )
SELECT CHARINDEX( 'SILVA', 'CARLOS Alberto' )
```

Results		Messages
(No column name)		
1	3	
(No column name)		
1	1	
(No column name)		
1	0	

- **CONCAT**

```
SELECT CONCAT ( 'SQL ', 'módulo ', 'I' ) as CONCATENAR_TEXTO;
```

Results Messages	
CONCATENAR_TEXTO	
1	SQL módulo I

- **CONCAT_WS**

```
SELECT CONCAT_WS( '-', 'A', 'B', 'C' )
```

Results Messages	
(No column name)	
1	A-B-C

- **LEFT**

```
--- Pegar uma parte de uma string  
SELECT LEFT( 'IMPACTA TECNOLOGIA', 5 ) -- Retorna IMPAC
```

- **LEN**

```
SELECT LEN ( 'Brasil' ) as QTD_CARACTERES
```

Results Messages	
QTD_CARACTERES	
1	6

- **LOWER**

```
--- Minúsculo  
PRINT LOWER( 'tECnOLOGIA 123 @@ !!' )
```

- **LTRIM**

```
--- Eliminar espaços em branco à esquerda  
SELECT 'IMPACTA' + '          TECNOLOGIA'  
SELECT 'IMPACTA' + LTRIM('          TECNOLOGIA')
```

- **RIGHT**

```
SELECT RIGHT( 'IMPACTA TECNOLOGIA', 5 ) -- Retorna LOGIA
```

- **RTRIM**

```
--- Eliminar espaços em branco à direita
SELECT 'IMPACTA' + 'TECNOLOGIA'
SELECT RTRIM('IMPACTA') + 'TECNOLOGIA'
```

- **REVERSE**

```
--- Reverso de uma string
SELECT REVERSE('IMPACTA')
SELECT REVERSE(ZÉ)
SELECT REVERSE('TECNOLOGIA')
SELECT REVERSE('COMPUTADOR')
SELECT REVERSE('REVER')
SELECT REVERSE('ANILINA')
SELECT REVERSE('MIRIM')
SELECT REVERSE('RADAR')
SELECT REVERSE('REVIVER')
```

- **REPLACE**

```
-- Substituição de texto
PRINT REPLACE('JOSÉ,CARLOS,ROBERTO,FERNANDO,MARCO',' ',' - ')
```

- **REPLICATE**

```
SELECT REPLICATE('TESTE',4) AS REPLICA_A_PALAVRA
```

Results		Messages
	(No column name)	
1	TesteTesteTesteTeste	

- **REPLICATE**

```
--- Replicar string várias vezes
PRINT REPLICATE('IMPACTA', 10)
```

- **STR**

```
SELECT STR (-1) AS CONVERTE_EM_NUMERO;
SELECT STR (0) AS CONVERTE_EM_NUMERO;
SELECT STR (215) AS CONVERTE_EM_NUMERO;
```

	Converte_em_numero
1	-1

	Converte_em_numero
1	0

	Converte_em_numero
1	215

- **SUBSTRING**

```
SELECT SUBSTRING ('IMPACTA',3,2) AS RETORNA_PARTE_TEXTO;
```

	RETORNA_PARTE_TEXTO
1	PA

- **STRING_SPLIT**

```
SELECT * FROM STRING_SPLIT('A,T,C,D' , ',') AS A
```

	value
1	A
2	T
3	C
4	D

- **STRING_AGG**

```
SELECT STRING_AGG (E_MAIL, ';' ) FROM TB_CLIENTE
WHERE ID_CLIENTE<10
```

	(No column name)
1	XIMIKNUJBPGWXXVEEGQSDQBOJL;OXIBXDMWXXIVNCKVHSEFWK...

- UPPER

```
--- Maiúsculo
PRINT UPPER('impacta 123 @@ !!')
```

5.3. Funções matemáticas

O SQL possui diversas funções matemáticas. Segue algumas delas:

Função	Descrição	Argumento
ABS	Valor absoluto positivo.	Expressão numérica
DEGREES	Retorna o ângulo correspondente.	Expressão numérica
RAND	Retorna um valor (float) randômico entre 0 e 1.	Expressão numérica
ACOS	Retorna o ângulo correspondente em radianos.	Expressão numérica
EXP	Retorna o valor exponencial.	Expressão numérica
ROUND	Realiza o arredondamento de um valor numérico.	Expressão numérica, tamanho, função
ASIN	Retorna o ângulo, em radianos, do seno.	Expressão numérica
FLOOR	Retorna o valor inteiro de um número.	Expressão numérica
SIGN	Retorna o valor -1, 0 ou +1 de um valor.	Expressão numérica
ATAN	Retorna a tangente de um número.	Expressão numérica
LOG	Retorna o logaritmo de um número.	Expressão numérica, base
SIN	Retorna o seno de um número.	Expressão numérica
ATN2	Retorna o radiano de um número.	Expressão 1, Expressão 2
LOG10	Retorna o logaritmo base 10.	Expressão numérica
SQRT	Retorna raiz quadrada.	Expressão numérica
CEILING	Retorna o menor inteiro maior ou igual à expressão.	Expressão numérica

Função	Descrição	Argumento
PI	Retorna o valor de PI.	
SQUARE	Retorna o quadrado do valor.	Expressão numérica
COS	Retorna o cosseno.	Expressão numérica
POWER	Retorna a potência de um valor.	Expressão numérica, potência
TAN	Retorna a tangente.	Expressão numérica
COT	Retorna a cotangente de um valor.	Expressão numérica
RADIANS	Retorna o radiano de um valor.	Expressão numérica

Alguns exemplos:

- CEILING**

```
SELECT CEILING(150.54) AS PROXIMO_NUMERO_INTEIRO
SELECT CEILING(149) AS PROXIMO_NUMERO_INTEIRO
SELECT CEILING(149.99) AS PROXIMO_NUMERO_INTEIRO
```

- FLOOR**

```
SELECT FLOOR(1.3012) AS VALOR_INTERIRO
SELECT FLOOR(1.999) AS VALOR_INTERIRO
```

Results Messages

VALOR_INTERIRO
1

VALOR_INTERIRO
1

- PI**

```
SELECT PI() AS VALOR_DE_PI
```

- RAND**

Esta função retorna um valor randômico (float) entre 0 e 1.

```
SELECT RAND() AS NUMERO_RANDOMICO
```

Caso coloquemos um valor dentro dos parênteses, o valor será fixo:

```
SELECT RAND(1) AS NUMERO_RANDOMICO
SELECT RAND(2) AS NUMERO_RANDOMICO
```

Results Messages	
	NUMERO_RANDOMICO
1	0,713591993212924
	NUMERO_RANDOMICO
1	0,713610626184182

- **ROUND**

```
SELECT ROUND(10/3., 5) AS VALOR_ARREDONDADO
SELECT ROUND(10., 5) AS VALOR_ARREDONDADO
SELECT ROUND(3.333333333, 2) AS VALOR_ARREDONDADO
```

Results Messages	
	VALOR_ARREDONDADO
1	3.333330
	VALOR_ARREDONDADO
1	10
	VALOR_ARREDONDADO
1	3.3300000000

- **SQRT**

```
SELECT SQRT(144) AS RAIZ_QUADRADA
```

Results Messages	
	RAIZ_QUADRADA
1	12

- **SQUARE**

```
SELECT SQUARE(10) AS QUADRADO_DO_VALOR
```

Results Messages	
	QUADRADO_DO_VALOR
1	100

- POWER

```
SELECT POWER(10,3) AS EXPONENCIAL
```

Results		Messages
EXPONENCIAL		
1	1000	

5.4. Funções de data e hora

Função	Descrição
DATEADD	Adiciona um valor de data ou hora para uma data.
DATEDIFF	Realiza a diferença entre datas.
DATENAME	Retorna o nome da data. Pode ser dia da semana ou dia do mês.
DATEPART	Retorna parte da data.
DAY	Retorna o dia da data.
GETDATE()	Retorna data e hora do servidor.
GETUTCDATE()	Retorna data e hora do servidor com UTC.
MONTH	Retorna o mês da data especificada.
YEAR	Retorna o ano da data especificada.
SYSDATETIME()	Retorna data e hora do servidor com precisão de sete dígitos.
SYSDATETIMEOFFSET()	Retorna DateTimeOffset com uma precisão de sete.
FORMAT	Formata uma data.
DATEFROMPARTS	Monta uma data a partir de argumentos inteiros.
TIMEFROMPARTS	Monta uma hora a partir de argumentos inteiros.
DATETIMEFROMPARTS	Monta data e hora no padrão datetime.
DATETIME2FROMPARTS	Monta data e hora no padrão datetime2.
SMALLDATETIMEFROMPARTS	Monta data e hora no padrão smalldatetime.
DATETIMEOFFSETFROMPARTS	Monta data e hora no padrão datetime offset.
EOMonth	Apresenta o último dia do mês.

- **FORMAT** (expressão, formato)

Formata uma expressão numérica ou date/time no formato definido por uma string de formatação.

A seguir, temos alguns exemplos de caracteres usados na formatação de strings:

- **Caracteres para formatação de números:**

0 (zero)	Define uma posição numérica. Se não existir número na posição, aparece o zero.
#	Define uma posição numérica. Se não existir número na posição, fica vazio.
. (ponto)	Separador de decimal.
, (vírgula)	Separador de milhar.
%	Mostra o sinal e o número multiplicado por 100.

Qualquer outro caractere inserido na máscara de formatação será exibido normalmente na posição em que foi colocado.

- **Caracteres para formatação de data:**

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

Veja um exemplo:

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

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

- **DATEADD()**

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

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

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

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

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

O próximo código retorna o dia de hoje mais seis meses:

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

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

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

- **DATEDIFF()**

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

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

O argumento **parte** representa a porção de **data_inicio** e **data_final** que especificará o limite cruzado.

O exemplo a seguir retorna a quantidade de dias vividos até hoje por uma pessoa nascida em 12 de julho de 1989:

```
SELECT DATEDIFF( DAY, '1989.07.12', GETDATE());
```

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

```
SELECT DATEDIFF( MONTH, '1989.07.12', GETDATE());
```

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

```
SELECT DATEDIFF( YEAR, '1989.07.12', GETDATE());
```

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

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

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

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

- **DATEFROMPARTS()**

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

```
DATEFROMPARTS (ano, mês, dia)
```

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

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

O resultado é mostrado a seguir:

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

- **TIMEFROMPARTS()**

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

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

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

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

O resultado é mostrado a seguir:

Results Messages	
(No column name)	
1	10:25:15

- **DATETIMEFROMPARTS()**

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

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

Caso algum valor esteja incorreto, a função retornará um erro. Agora, se o parâmetro for nulo, a função retornará valor nulo. No exemplo adiante, vamos retornar o valor 15 de setembro de 2013 às 14:00:15.0000:

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

O resultado é mostrado a seguir:

Results Messages	
(No column name)	
1	2013-09-15 14:00:15.0000

- **DATETIME2FROMPARTS()**

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

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

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

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


O resultado é mostrado a seguir:

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

- **SMALLDATETIMEFROMPARTS()**

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

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

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

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

O resultado é mostrado a seguir:

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

- **DATETIMEOFFSETFROMPARTS()**

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

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

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

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

O resultado é mostrado a seguir:

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

- **EOMonth()**

Esta função retorna o último dia do mês a partir dos parâmetros **data_início** e **adicionar_mês** (opcional) da seguinte sintaxe:

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

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

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

O resultado desse exemplo é mostrado a seguir:

Results		Messages
(No column name)		
1	2019-05-31	

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

```
SELECT EOMONTH (GETDATE(),1);
```

Results		Messages
(No column name)		
1	2019-06-30	

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

```
SELECT EOMONTH (GETDATE(),-1);
```

Results		Messages
(No column name)		
1	2019-04-30	

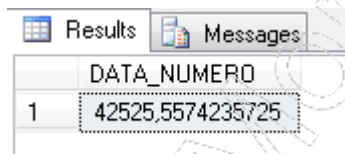
5.5. Funções de conversão

- **CAST**: Converte um tipo de dados em outro;

Exemplos:

- Convertendo uma data em número:

```
SELECT CAST(GETDATE() AS FLOAT) AS DATA_NUMERO
```

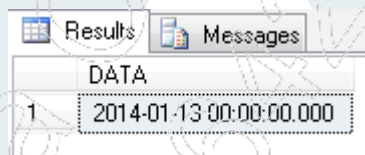


The screenshot shows a SQL Server Results window with a single column header 'DATA_NUMERO'. The first row contains the value '42525,5574235725'.

	DATA_NUMERO
1	42525,5574235725

- Convertendo um texto em data:

```
SELECT CAST('2014.1.13' AS DATETIME) AS DATA
```

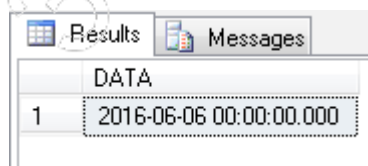


The screenshot shows a SQL Server Results window with a single column header 'DATA'. The first row contains the value '2014-01-13 00:00:00.000'.

	DATA
1	2014-01-13 00:00:00.000

- Convertendo um número em data:

```
SELECT CAST(42525 AS DATETIME) AS DATA
```



The screenshot shows a SQL Server Results window with a single column header 'DATA'. The first row contains the value '2016-06-06 00:00:00.000'.

	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;

Veamos alguns argumentos:

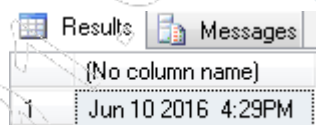
Código	País	Formato
101	EUA	1 = mm/dd/aa
103	Britânico/francês	3 = dd/mm/aa
		103 = dd/mm/aaaa
111	JAPÃO	11 = aa/mm/dd
		111 = aaaa/mm/dd
13 ou 113	Padrão Europa + milissegundos	dd mês aaaa hh:mi:ss:mmm (24h)
114	-	hh:mi:ss:mmm(24h)
21 ou 121		aaaa-mm-dd hh:mi:ss. mmm (24h)

Exemplos:

- Convertendo uma data para texto com **CAST**:

```
SELECT CAST (GETDATE() AS VARCHAR(20))
```

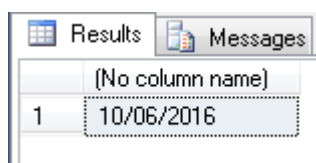
No exemplo anterior, o formato é o de data completa.



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	Messages
(No column name)	
1	10/06/2016

- Retornando a data no formato AAAA/MM/DD:

```
SELECT CONVERT(VARCHAR(10), GETDATE() , 111)
```

Results		Messages
(No column name)		
1	2016/06/10	

- Retornando a hora:

```
SELECT CONVERT(VARCHAR(10), GETDATE() , 114)
```

Results		Messages
(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.

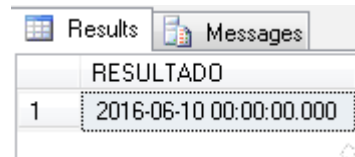
Vejamos uma listagem com alguns códigos de cultura:

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:

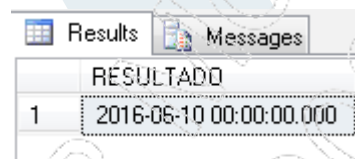
```
SELECT PARSE('2016.6.10' AS datetime) AS RESULTADO;
```



RESULTS	
Messages	
RESULTADO	
1	2016-06-10 00:00:00.000

- Convertendo um texto em data utilizando o formato americano:

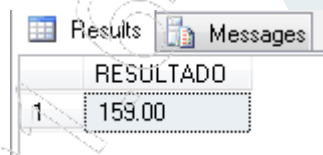
```
SELECT PARSE('2016.6.10' AS datetime USING 'en-US') AS  
RESULTADO;
```



RESULTS	
Messages	
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:

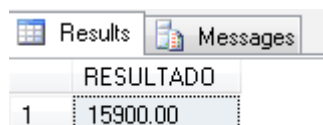
```
SELECT PARSE('159.00' AS DECIMAL(10,2)) AS RESULTADO
```



RESULTS	
Messages	
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	
RESULTS	
1	RESULTADO
	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	
RESULTS	
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	
RESULTS	
1	RESULTADO
	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)
```

RESULTS	
RESULTS	
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
```

Results Messages	
RESULTADO	
1	NULL

5.6. Funções de classificação

Vejamos, nos subtópicos a seguir, as funções de classificação.

5.6.1. 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:

```
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.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME
) AS A
```


Vejamos o resultado da consulta:

Results Messages			
	NOME	TOTAL	LINHA
1	WTUKESLWUWIECMGFDBETXWNUAKQBK	112510.02	1
2	MJJQNSJUEHRGCHGQWOFKQCWMJYXTYT	111033.06	2
3	XRFNMSMWVSJPWWTKVKHPBKKEGKFAD	108433.24	3
4	TOPQJEASUONMWDSGAQTPLXWDJLLMMC	101155.53	4
5	NNBNLWMTDNEGWBULDOKXMYMWXPTXL	100375.40	5
6	VDRYTHUPMYGFNEDENXQEFFDJRFVHAR	99976.04	6
7	PNYWVAEXLFDLVKSMKKGHLJOSIDWJHJ	99271.43	7
8	YKWCXOIXPTFXLRKJMNVDUNDGPKNLYX	99226.09	8
9	DFHUSFGFBVPBPNEVOMSWDROXXRJUM	98535.92	9
10	PHLGMHQQVPCLXHLDUASOHJAQIEXJL	96733.36	10

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 , E.ESTADO, COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
JOIN TB_ENDERECO AS E ON E.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME, E.ESTADO
) AS A
```

Verifique que a cada novo estado é inicializada a contagem de cada linha, gerando uma nova consulta:

Results Messages				
	NOME	QTD_PEDIDOS	ESTADO	LINHA
1	NLJLSDJFVKLNAFNIACVEIFUUBHMTKV	59	NULL	1
2	WTTFOQKGBYVWXDNBQXNETJRJTGLNH	54	NULL	2
3	YSWTKPUFFNKBFBKIBUQBXHRVINNLWDC	54	NULL	3
4	BXWDJFGFCLUSFCAIOPMGTSRCETPKLJ	42	NULL	4
5	JVXTKMMMPUDACKERDEJMSVTHMDSNCHQ	39	NULL	5
6	LELOTXDTFJDQLKSQFHCWFKJHROHUNA	38	NULL	6
7	MDNKLWMFROMTDBKCLFCABDSIKNHWXT	36	NULL	7
8	MBNYGNLHWXFOOKKWGXCBXELBRSAM	36	NULL	8
9	UJQGOQJTDKTQARGLDUGQOQBQUGWKLK	35	NULL	9
10	MJPRTSGDLOYNDRMQXJGHWJHANSUFO	35	NULL	10

5.6.2. 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**:

```
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.ID_CLIENTE = P.ID_CLIENTE
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
1	XRFNMSMVVSJPWWTKVKHPBKKEGKFAD	83	1
2	NLJLSDJFVKLNAFNIACVEIFUJBHMTKV	59	2
3	VDRYTHUPMYGFNEDEXQEFFDJRFVHAR	55	3
4	WTTFOQKGBYVWXDNBQXNETJRJTGCLNH	54	4
5	PEGBNYVFGPHOWDLIGOVCTPYKRECJGG	42	5
6	DFHUSFGFGBVPBPNEVOMSWDROXXRJUM	42	5
7	SXRNWHKYLFGMRPXQTAJJQJJQUNQRL	41	7
8	JVXTKMPUDACKERDEJMSVTHMDSNCHQ	39	8
9	MBNYGNLHVXFOOKKWGXCHBXLBR SAM	36	9
10	MJPRTSGDLOYNDRMQXJGHWJHANSDUFO	35	10

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 , E.ESTADO, COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
JOIN TB_ENDERECO AS E ON E.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME, E.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
1	NLJLSDJFVKLNAFNIACVEIFUUBHMTKV	NULL	59	1
2	WTTFOQKGBYVWXDNBQXNETJRJTGCLNH	NULL	54	2
3	YSWTKPUFFNKBFKIBUQBXFVINNLWDC	NULL	54	2
4	BXWJDJFGCLUSFCAIOPMGTSRCETPKLJ	NULL	42	4
5	JVXTKMMPUACKERDEJMSVTHMDSNCHQ	NULL	39	5
6	LELOTXDTFJDQLKSQFHCWFKJHROHUNA	NULL	38	6
7	MDNKLWMFROMTDBKCLFCABDSIKNHWXT	NULL	36	7
8	MBNYGNLHVXFOOKKWGXCHBXLBRSAM	NULL	36	7
9	UJQGOQJTDKTQARGLDUGQOQBQUGWKLK	NULL	35	9
10	MJPRTSGLDYNDRMQXJGHWJHANSUFO	NULL	35	9
11	YRHLJAJPSOWTJAPUOBSCQHUGBJGOH	NULL	34	11

5.6.3.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**:

```
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.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME
) AS A
```

Observe que o resultado da classificação continuará o valor da sequência:

	NOME	QTD_PEDIDOS	CLASSIFICAÇÃO
1	XRFNMSMVVSJPWWTKVKHPBKKEGIKFAD	83	1
2	NLJLSDJFVKLNAFNIACVEIFUUBHMTKV	59	2
3	VDRYTHUPMYGFNEDENXQEFFDJRFVHAR	55	3
4	WTTFOQKGBYVWXDNBQXNETJRJTGCLNH	54	4
5	PEGBNYVFGPHOWDLIGOVCTPYKRECJGG	42	5
6	DFHUSFGGBVPBPNEVOMSWDROXXRJUM	42	5
7	SXRNWHKYLFGMRPXQTAJJQQJQUNQRL	41	6
8	JVXTKMMPUDACKERDEJMSVTHMDSNCHQ	39	7
9	MBNYGNLHVXFOOKKWGXCHBXLBR SAM	36	8
10	MJPRTSGDLOYNDRMQXJGHWWJHANSUFO	35	9
11	UJQGOQJTDKTQARGLDUGQOQBQUGWKLK	35	9

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 , E.ESTADO, COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
JOIN TB_ENDERECO AS E ON E.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME, E.ESTADO
) AS A
Order by Estado
```

Verifique que o resultado não apresenta saltos de valores:

	NOME	ESTADO	QTD_PEDIDOS	CLASSIFICAÇÃO
1	NLJLSDJFVKLNAFNIACVEIFUUBHMTKV	NULL	59	1
2	WTTFOQKGBYVWxDNBQXNETJRJTGLNH	NULL	54	2
3	YSWTKPUFFNKBKIBUQBXHFVINNLWDC	NULL	54	2
4	BXWDJFGFCLUSFCAIOPMGTSRCETPKLJ	NULL	42	3
5	JVXTKMPUDACKERDEJMSVTHMDSNCHQ	NULL	39	4
6	LELOTXDTFJDQLKSQFHCWFKJHROHNA	NULL	38	5
7	MDNKLWMFROMTDBKCLFCABDSIKNHwXT	NULL	36	6
8	MBNYGNLHVXFOOKKWGXCHBXLBR SAM	NULL	36	6
9	UJQGOQJTDKTQARGLDUGQOQBQUGWKLK	NULL	35	7
10	MJPRTSGLDYNDRMQXJGHWJHANSUFO	NULL	35	7
11	YRHLJAJPSOWTJAPUOBSCQHUOGBJGOH	NULL	34	8

5.6.4. NTILE

Cria uma coluna de agrupamento sobre o total de linhas.

```
NTILE (integer_expression) OVER ( [ <partition_by_clause> ] <
order_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 , E.ESTADO, SUM(P.VLR_TOTAL) AS TOTAL
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
JOIN TB_ENDERECO AS E ON E.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME, E.ESTADO
) AS A
```

O resultado apresenta as linhas agrupadas em 10 grupos:

	NOME	TOTAL	ESTADO	GRUPO
1	WTUKESLWUWIECMGFPDBETXWNUAKQBK	225020.04	DF	1
2	XRFNMSMVVSJPWWTkVXHPBKKEGIKAD	216866.48	SP	1
3	NNBNLWMTDNEGWBULDOKXMYMwXPTXL	200750.80	SP	1
4	VDRYTHUPMYGFNEDENXQEFFDJRFVHAR	199952.08	SP	1
5	PNYwVAEXLFDLVKSMKKGHLJOSIDWJHJ	198542.86	SP	1
6	YKWCXOIXPTPLRKJMNVDUNDGPKNLYX	198452.18	SP	1
7	DFHUSFGGBVPBPNEVOMSWDROXXRJUM	197071.84	SP	1
8	PHLGMHQVPLCXHLDUASOHJAQIEIJL	193466.72	GO	1
9	JLJSEFXGFFJMPOFXBEJMSLGJLFGW	189184.36	SC	1
10	AIVNLERCVTPOKUKJDSEJSOSIVwWPXW	184236.04	SP	1
11	FORLQETEXJBWMJUMCMNVQCOTFUJNR	182254.10	RJ	1

5.6.5. 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 ,E.ESTADO, COUNT(*) AS QTD_PEDIDOS
FROM TB_PEDIDO AS P
JOIN TB_CLIENTE AS C ON C.ID_CLIENTE = P.ID_CLIENTE
JOIN TB_ENDERECO AS E ON E.ID_CLIENTE = P.ID_CLIENTE
GROUP BY C.NOME,E.ESTADO
) AS A
```

Resultado:

	NOME	ESTADO	QTD_PEDIDOS	ROW_NUMBER	RANK	DENSE_RANK	NTILE
1	NLJLSDJFVKLNAFNIACVEIFUUBHMTKV	NULL	59	1	1	1	1
2	WTTFOQKGBYVWXDNBQXNETJRJTGCNLH	NULL	54	2	2	2	1
3	YSWTKPUFFNKBFKIBUQBXHRVNNLWDC	NULL	54	3	2	2	1
4	BXWDJFGFCLUSFCAIOPMGTSRCETPKLI	NULL	42	4	4	3	1
5	JVXTKMMPUDACKERDEJMSVTHMDSNCHQ	NULL	39	5	5	4	1
6	LELOTXDTFJDQLKSQFHCWFKJHROHNA	NULL	38	6	6	5	1
7	MDNKLWMFROMTDBKCLFCABDSIKNHWXT	NULL	36	7	7	6	1
8	MBNYGNLHVXFOOKKWGXCHBXELBRSAM	NULL	36	8	7	6	1
9	UJQGOQJTDKTQARGLDUGQOQBQUGWKLK	NULL	35	9	9	7	1
10	MJPRTSGDLOYNDRMQXJGHVJHANSUFD	NULL	35	10	9	7	1
11	YRHLJAJPSOWTJAPUOBSCQUHUGBJGOH	NULL	34	11	11	8	1

Pontos principais

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

- Existem diversas funções que auxiliam na construção de consultas. Essas funções podem ser: texto, matemática, data e hora, conversão e classificação;
- Funções de cadeia de caractere podem retornar a quantidade de caracteres, concatenar texto, extrair parte de um texto etc.;
- vFunções de classificação retornam valores a partir de uma ordenação independente da cláusula final do **ORDER BY**.



Funções nativas

Teste seus conhecimentos





1. O que são funções determinísticas?

- ☐ a) São aquelas cujo resultado podemos prever.
- ☐ b) São aquelas cujo resultado não podemos determinar.
- ☐ c) Determinam o retorno de uma função tabular.
- ☐ d) Não determinam o resultado de uma consulta.
- ☐ e) Nenhuma das alternativas anteriores está correta.

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

Brúna 391.642.200700

4. Qual função adiante não é de texto?

- ☐ a) LEN
- ☐ b) REPLACE
- ☐ c) SQRT
- ☐ d) UPPER
- ☐ e) STRING_AGG

5. De que tipo é a função ROW_NUMBER?

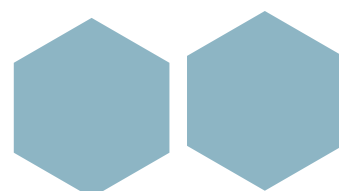
- ☐ a) Texto
- ☐ b) Matemática
- ☐ c) Data
- ☐ d) Classificação
- ☐ e) Comparação



Funções nativas



Mãos à obra!





Laboratório 1

A – Trabalhando com funções nativas

1. Coloque em uso o banco de dados **db_Ecommerce**;
2. Retorne data e hora do SQL Server;
3. Concatene o texto a seguir:

Curso
Impacta
SQL
Server
2019
Módulo
I

4. Extraia 10 caracteres à esquerda do texto **CURSO DE SQL SERVER – IMPACTA**;
5. Extraia os dois primeiros caracteres do nome do empregado;
6. Coloque o nome do empregado em letras maiúsculas;
7. Apresente as vendas de toda sexta-feira de 2018;
8. Apresente as vendas de 2017, formatando a data para DD/MM/YYYY;
9. Selecione o nome do empregado, a data de admissão e um campo calculado com a data de admissão mais três meses;
10. Calcule há quantos anos o funcionário trabalha na empresa.

Laboratório 2

A – Trabalhando com funções nativas II

1. Coloque em uso o banco de dados **db_Ecommerce**;
2. 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;
3. Utilizando a mesma consulta, adicione uma coluna que apresente o ranking das vendas dos clientes.

6

Programação

- Variáveis;
- Operadores;
- Controle de fluxo;
- WHILE;
- Outros comandos;
- Query dinâmicas;
- Tratamento de erros;
- Mensagens de erro.

6.1. Introdução

O presente capítulo 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.

6.2. Variáveis

Uma **variável local** do T-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.

6.2.1. 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
```

6.3. 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 2019, os quais são divididos em categorias. A seguir, veremos os operadores aritméticos, relacionais e lógicos.

6.3.1. 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

6.3.2. 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

6.3.3. 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 .

6.3.4. 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 2019:

Ordem de execução	Operadores
1	~ (Bitwise NOT)
2	*, /, %
3	+ (Positivo), - (Negativo), + (Adição), (+ Concatenação), - (Subtração), & (Bitwise AND), ^ (Bitwise exclusivo OR), (Bitwise OR)
4	=, >, <, >=, <=, <>, !=, !>, !<
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.

6.4. Controle de fluxo

O SQL Server 2019 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.

6.4.1. 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**.

6.4.2. 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 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'
```

6.5.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**.

6.5.1. 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**.

6.6. 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.

6.6.1.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
    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;

```

6.7. Outros comandos

Vejamos, nos subtópicos seguintes, outros comandos de controle de fluxo.

6.7.1. 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**:

```

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.

6.7.2. 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"'
```

6.7.3. 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'
```

6.7.4.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:

```
EXISTS (SELECT * FROM PEDIDOS.DBO.TB_CLIENTE WHERE CODCLI=43)
BEGIN
    PRINT 'Código existe'
END
ELSE
    BEGIN
        PRINT 'Código não existe'
    END
```

6.7.5.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 tb_PEDIDO WHERE VLR_
TOTAL IS NOT NULL)

PRINT @SOMA
```

6.8.Queries 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 ID_CLIENTE=' +
CAST(@CODCLI AS VARCHAR(5))
EXEC( @SQL )
```

6.9.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.

6.9.1.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.
15	Indica erros de sintaxe em um comando.

Nível de Severidade	Descrição
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.

6.9.2. @@ERROR

A função de @@ERROR é retornar um número referente a um erro ocorrido na última instrução T-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.

6.9.3. 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 T-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**.

6.9.4. 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**.

6.10. 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.

6.10.1. 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;
- **[@msgtext =] '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 **SYS.MESSAGES**:

```
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
```

6.10.2. 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 opção [ ,...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**;

- **opção:** 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.

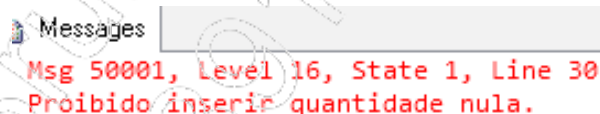
Quando o **RAISERROR** é chamado, a partir de uma **STORED PROCEDURE** remota, com o valor de severidade inferior a 20, ocorre um erro que aborta o comando no servidor remoto. Visto que apenas os erros que abortam batches remotos são manipulados por uma construção **TRY...CATCH** no servidor local, caso 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.

6.10.3.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:

- Teste com **ID_PRODUTO = 780**

```

DECLARE @ID_PRODUTO INT = 780

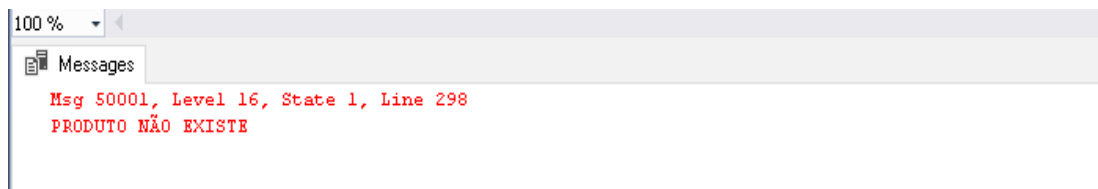
BEGIN TRY
--Realiza o teste para verificar se um produto existe
    IF NOT EXISTS(SELECT * FROM TB_PRODUTO_ERRO WHERE ID_
PRODUTO = @ID_PRODUTO)
        -- gera um erro, o que provoca a execução do bloco
    CATCH
        THROW 50001, 'PRODUTO NÃO EXISTE',1;

        INSERT INTO TB_PRODUTO_ERRO ( ID_PRODUTO, DESCRICAO,
ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
        PRECO_VENDA, QTD_REAL, QTD_MINIMA, CLAS_FISC, IPI,
PESO_LIQ)
        SELECT ID_PRODUTO, DESCRICAO, ID_UNIDADE, ID_TIPO,
PRECO_CUSTO, PRECO_VENDA,
        QTD_REAL, QTD_MINIMA, CLAS_FISC, IPI, PESO_LIQ
        FROM TB_PRODUTO WHERE ID_PRODUTO = @ID_PRODUTO

END TRY
BEGIN CATCH
    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

```

Como o produto não existe, será apresentada a mensagem:



Pontos principais

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

- 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;
- A função de **@@ERROR** é retornar um número referente a um erro ocorrido na última instrução T-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.



Programação

Teste seus conhecimentos



1. 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 utiliza o comando TRY...CATCH.
- ☐ e) Este comando não gera erro.

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 comando a seguir e selecione a afirmação correta:

```
BEGIN TRY  
  
    UPDATE CLIENTES SET ESTADO = 'SP' WHERE CODCLI=13  
  
BEGIN CATCH  
    PRINT 'OCORREU UM ERRO'  
  
END CATCH
```

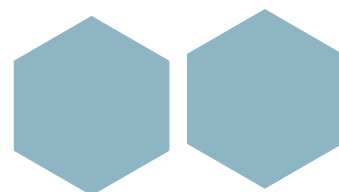
- ☐ a) Caso ocorra erro, será mostrada a mensagem “OCORREU UM ERRO”.
- ☐ b) Gera um erro de sintaxe.
- ☐ c) Caso ocorra erro, será mostrada a mensagem “OCORREU UM ERRO” e realizado um ROLLBACK.
- ☐ d) O correto é utilizar o comando @@ERROR.
- ☐ e) Este comando não gera erro.



Programação



Mãos à obra!



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
```

3. 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));
```

4. 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));
```

5. 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 (inteiras) 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)
```

6. Para o código adiante, realize o tratamento de erro e, caso ocorra erro, informe que terminou devido a um erro;

```
DECLARE @CONT INT = 100

WHILE @CONT >=0
BEGIN

    PRINT 100. / @CONT
    SET @CONT-=1

END
```

7. Utilizando o conceito de **Query Dinâmica**, execute os comandos a seguir:

- Declare as variáveis:
 - @CODCLI INT;
 - @SQL VARCHAR(700).
- Atribua os valores para as variáveis:
 - 1 para @CODCLI;
 - Branco para @SQL.
- Crie um LOOP (WHILE) do valor 1 até 700;
- Crie uma Query Dinâmica com a consulta:

```
'SELECT ID_CLIENTE, COUNT(*) AS QTD_PEDIDO FROM TB_PEDIDO
WHERE ID_CLIENTE=' + CAST(@CODCLI AS VARCHAR(10)) +
' GROUP BY ID_CLIENTE'
```

- Apresente o código do cliente e a quantidade de pedidos.

7

Stored procedures

- STORED PROCEDURES;
- CURSOR;
- Parâmetros tabulares (TABLE-VALUED);
- Boas práticas;
- Recompilando STORED PROCEDURES;
- XP_CMDSHELL;
- CLR STORED PROCEDURE;
- SP_EXECUTE_EXTERNAL_SCRIPT;
- Compilação Nativa.



7.1.Introdução

Uma coleção de comandos T-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.

7.2.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.

7.2.1. 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.

7.2.2. 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:

Vejamos um exemplo:

```
CREATE PROCEDURE SP_LISTA_CLIENTE AS
BEGIN

    SELECT      ID_CLIENTE , NOME
    FROM  TB_CLIENTE

END
GO

EXEC SP_LISTA_CLIENTE
```

Results Messages

	ID_CLIENTE	NOME
1	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE
2	4	QCQQBNPUKNHDSFBMEKESJNMJMJQDDP
3	5	TTOMDCPSXCWRXSQWWNOVURRENQDFKL
4	6	ANXTUDDIVPWQUVINKNFBVLEOSUODXI
5	7	TCSGVJISYSISLFFELSMMLYSBMPRQNK
6	8	QXVTRVGAOBVBXCAAQIGMMHNFIFVJH
7	11	OVRUUMSLRNTIJVGPCVLHFJQTKRSDKD
8	12	RDVNRMHFEIRJCATCTAQDFLBHHMUID
9	13	EAPFLIPYEWEXBAKTVOHLPVXWKOCBVR
10	14	BPBQCCYFGMVGGIUOXJHNMTRTDGGAMQ
11	15	FJHYSGMGXIJYGTURGEGXOWQKRBTRE

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.

```
ALTER PROCEDURE SP_LISTA_CLIENTE AS
BEGIN

    SELECT      ID_CLIENTE , NOME
    FROM  TB_CLIENTE

    SELECT 'PROCEDURE: SP_LISTA_CLIENTE - ' + 'NÍVEL DA
PROCEDURE: ' + CAST(@@NESTLEVEL AS VARCHAR(10))

END
GO

CREATE PROCEDURE SP_LISTA_CLIENTE2 AS
BEGIN

    EXEC SP_LISTA_CLIENTE

    SELECT 'PROCEDURE: SP_LISTA_CLIENTE2 - ' + 'NÍVEL DA
PROCEDURE: ' + CAST(@@NESTLEVEL AS VARCHAR(10))

END
```

Results		Messages
	ID_CLIENTE	NOME
1	3	SQSKRGYHTBLNWIAREAKTKDQPWQFOLE
2	4	QCQQBNPUKNHDSFBMEKESJNMJMJQDDP
3	5	TTOMDCPSXCWRXSQWWNOVURRENQDFKL
4	6	ANXTIUDIVPWQUVINKNFBVLEOSUODXI
5	7	TCSGVJISYSISLFFELSMLLYSBMPRONK
6	8	QXVTRVGAQBVBXCAAQIGMMHNFIFVJH
7	11	QVRUUMSLRNTIJVGPCVLHFJQTKRSDKD
8	12	RDVNRMHFEIRJCATCTAQDFLBHJHMUIO
(No column name)		
1	PROCEDURE: SP_LISTA_CLIENTE - NÍVEL DA PROCEDURE: 2	
(No column name)		
1	PROCEDURE: SP_LISTA_CLIENTE2 - NÍVEL DA PROCEDURE: 1	

7.2.3. 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.

7.2.4. 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**.

7.2.5. 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
```

7.2.6. 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**.

7.2.7.Exemplos

O exemplo a seguir retorna o total vendido em cada um dos meses de um determinado ano:

```
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 2016
EXEC STP_TOT_VENDIDO 2017
```

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.ID_PEDIDO, I.ITEM, I.ID_PRODUTO, I.ID_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.ID_CLIENTE = C.ID_CLIENTE
    JOIN TB_VENDEDOR V ON PE.ID_EMPREGADO = V.CODVEN
    JOIN TB_ITENSPEDIDO I ON PE.ID_PEDIDO = I.ID_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.ID_PEDIDO
END
```


7.2.8. 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 '2016.1.1', '2016.1.31', '%A%', 'LEIA'
```

Podemos também omitir o nome do vendedor, que é o último parâmetro:

```
-- Omitindo o nome do vendedor  
EXEC STP_ITENS_PEDIDO '2016.1.1', '2016.1.31', '%A%'
```

Podemos, ainda, omitir os nomes do vendedor e do cliente:

```
-- Omitindo o nome do vendedor e do cliente  
EXEC STP_ITENS_PEDIDO '2016.1.1', '2016.1.31'
```

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 '2016.1.1', '2016.1.31', '%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.

7.2.9. Passagem de parâmetros nominal

Os parâmetros também podem ser passados nominalmente, conforme vemos a seguir:

```
EXEC STP_ITENS_PEDIDO @DT1 = '2016.1.1',  
@DT2 = '2016.1.31',  
@VENDEDOR = 'LEIA'
```

7.2.10. 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 ID_CLIENTE = @CODCLI )
    RETURN -1;

SELECT MAX(DATA_EMISSAO) AS ULT_DATA_COMPRA
FROM TB_PEDIDO WHERE ID_CLIENTE = @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'
```

Results		Messages	
		ULT_DATA_COMPRA	
1		2019-05-29 00:00:00.000	

- **Teste 2**

```
DECLARE @RET INT;
EXEC @RET = STP_ULT_DATA_COMPRA 1
IF @RET < 0 PRINT 'NÃO EXISTE PEDIDO DESTE CLIENTE'
```

Messages	
NÃO EXISTE PEDIDO DESTE CLIENTE	

7.2.11.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.

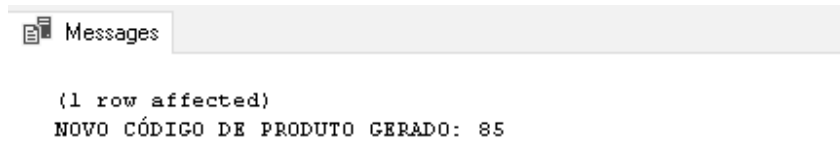
Apesar de existir a possibilidade de uma aplicação capturar a mensagem retornada por um comando **PRINT**, esse procedimento é muito complexo, por isso, costumamos utilizar o **PRINT** apenas em testes e rotinas administrativas utilizadas dentro do SQL Server Management Studio.

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
    ( DESCRICAO, ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
      PRECO_VENDA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
      IPI, PESO_LIQ )
    SELECT
      DESCRICAO, ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
      PRECO_VENDA, 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 'NOVO CÓDIGO DE PRODUTO GERADO: ' + CAST( @ID_
    PRODUTO_NOVO AS VARCHAR(10));
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:



7.2.12.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
    ( DESCRICAO, ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
      PRECO_VENDA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
      IPI, PESO_LIQ )
    SELECT
      DESCRICAO, ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
      PRECO_VENDA, 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 'NOVO CÓDIGO DE PRODUTO
    GERADO'
END
GO

-- Testando
EXEC STP_COPIA_PRODUTO 10
```

O retorno no SSMS será o seguinte:

The screenshot shows the 'Results' window in SQL Server Enterprise Manager. It displays a single row with the value 87 in the column 'NOVO CÓDIGO DE PRODUTO GERADO'.

	NOVO CÓDIGO DE PRODUTO GERADO
1	87

7.2.13. 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**.

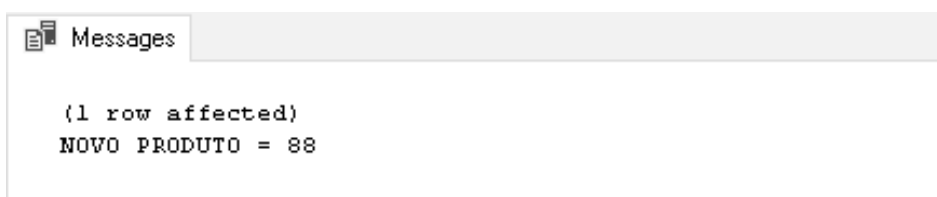
Quando utilizamos **OUTPUT**, podemos manter qualquer valor atribuído ao parâmetro enquanto a procedure é executada, mesmo depois que ela tenha sido finalizada.

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
    ( DESCRICAO, ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
      PRECO_VENDA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
      IPI, PESO_LIQ )
    SELECT
      DESCRICAO, ID_UNIDADE, ID_TIPO, PRECO_CUSTO,
      PRECO_VENDA, QTD_REAL, QTD_MINIMA, CLAS_FISC,
      IPI, PESO_LIQ
    FROM TB_PRODUTO
    WHERE ID_PRODUTO = @ID_PRODUTO;
    -- Descubra qual foi o ID_PRODUTO gerado
    SET @ID_PRODUTO_NOVO = SCOPE_IDENTITY();
END

-- 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**.

7.3.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 ID_EMPREGADO = @COD_SUP
    -- Busca a quantidade de subordinados do supervisor
    SELECT @QTD = COUNT(*) FROM TB_EMPREGADO
    WHERE COD_SUPERVISOR = @COD_SUP
    AND ID_EMPREGADO <> @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
```

O exemplo a seguir cria a procedure:

```
CREATE PROCEDURE SP_LISTA_QTD_EMPREGADOS AS
BEGIN
    --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 ID_EMPREGADO = @COD_SUP
        -- Busca a quantidade de subordinados do
        supervisor
        SELECT @QTD = COUNT(*) FROM TB_
        EMPREGADO
        WHERE COD_SUPERVISOR = @COD_SUP
        AND ID_EMPREGADO <> @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
END
GO

EXEC SP_LISTA_QTD_EMPREGADOS
```


Ao executar a procedure, o resultado é apresentado:

Messages

```
JOSE - 2 Subordinados
PAULO - 2 Subordinados
CARLOS ALBERTO - 11 Subordinados
CASSIANO - 1 Subordinados
ANA MARIA - 3 Subordinados
SEBASTIÃO - 2 Subordinados
EURICO - 3 Subordinados
ANA - 7 Subordinados
MARIANA - 3 Subordinados
MARIA - 2 Subordinados
ROBERTO ALEXANDRO - 3 Subordinados
ARNALDO - 2 Subordinados
ROGÉRIO - 2 Subordinados
ARLINDO - 3 Subordinados
LÚCIO - 8 Subordinados
```

7.4. 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. Seleccione 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:

	ID_CARGO	CARGO	SALARIO_INIC
9	9	COZINHEIRO	3330.00
10	10	AUXILIAR LABORATORIO	5000.00
11	11	GERENTE COMPRAS	4500.00
12	12	AUXILIAR DE MANUTENÇÃO	3300.00
13	13	GERENTE ADMINISTRATIVO	6000.00
14	14	TORNEIRO MECÂNICO	600.00
15	15	SUPERVISOR	2300.00
16	16	GERENTE DE PRODUÇÃO	500.00
17	17	QUÍMICO	2100.00
18	18	TESTANDO 1	500.00
19	19	TESTANDO 2	600.00
20	20	TESTANDO 3	700.00
21	21	TESTANDO 4	800.00

7.5. Boas práticas

Em alguns casos, a customização de uma procedure é importante para garantir a execução e transferência das informações corretamente.

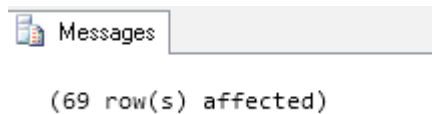
Vejamos, a seguir, algumas boas práticas para melhorar o desenvolvimento de Procedures:

- **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 quantidades de linhas afetadas:

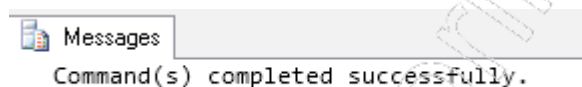


Utilizando o **SET NOCOUNT ON**, essa mensagem é descartada:

```
SET NOCOUNT ON
```

```
UPDATE TB_PRODUTO SET PRECO_VENDA = PRECO_VENDA * 1.2
```

Resultado:



- 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:



- 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.

7.6.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.

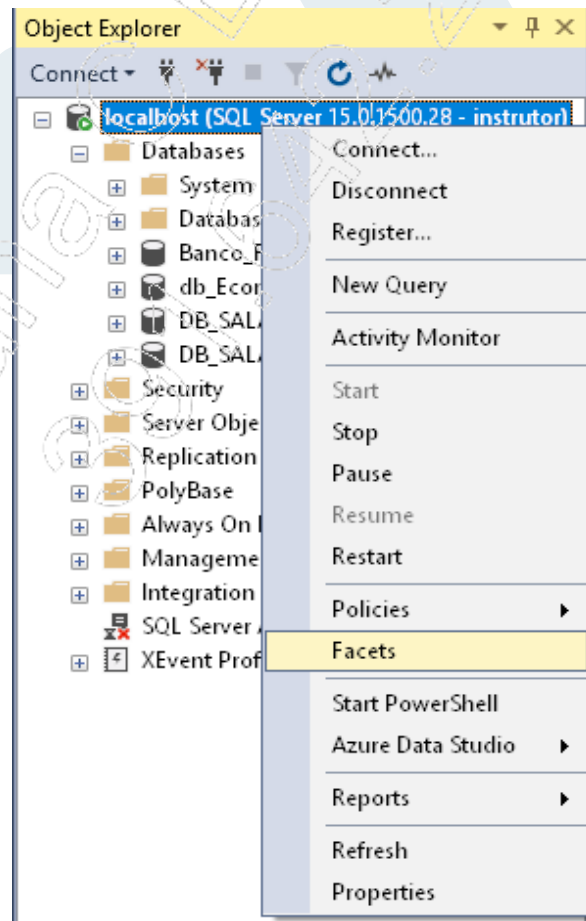
No entanto, podemos provocar a recompilação de uma stored procedure por meio da stored procedure de sistema **SP_RECOMPILE**, ou por meio da inclusão da opção **WITH RECOMPILE** à stored procedure.

7.7.XP_CMDSHELL

O comando **XP_CMDSHELL** executa um comando SHELL (linha de comando) a partir do SQL Server.

Esse tipo de comando aumenta as possibilidades de execução de comandos no nível de sistema operacional. Lembrando que todos os comandos são executados no servidor.

Esse comando não está habilitado por padrão. Clique com o botão direito do mouse sobre o nome da INSTANCE e selecione **Facets**:



Selecione **Server Configuration** em Facet:

Facet:	Server Configuration
Description:	Exposes properties of the Server regarding the configuration settings of the server.

Selecione **True** em **XPCmdShellEnabled**:

WebAssistantEnabled	Property value 'WebAssistantEnabled' is not available.
XPCmdShellEnabled	True

XPCmdShellEnabled
Gets or sets the Boolean property value that specifies whether the XP cmd shell enabled configuration option is enabled.

- Para utilizar o comando:

Execute o comando **DIR** no **C::**:

```
EXEC XP_CMDSHELL 'DIR C:'
```

Results		Messages
output		
1	O volume na unidade C não tem nome.	
2	O Número de Série do Volume é 1AC5-57C8	
3	NULL	
4	Pasta de C:\Windows\System32	
5	NULL	
6	21/05/2019 10:37 <DIR> .	
7	21/05/2019 10:37 <DIR> ..	
8	12/04/2018 13:37 <DIR> 0409	
9	29/04/2019 16:32 <DIR> 1033	
10	11/04/2018 20:34 308 @AudioToastico...	
11	11/04/2018 20:34 450 @BackgroundA...	
12	11/04/2018 20:34 199 @bitlockertoasti...	
13	11/04/2018 20:34 14.791 @edptostimag...	

- Criando um diretório:

```
EXEC XP_CMDSHELL 'MD C:\teste'
```

```
EXEC XP_CMDSHELL 'DIR C:\T*'
```

Results		Messages
		output
1		O volume na unidade C não tem nome.
2		O Número de Série do Volume é 1AC5-57C8
3		NULL
4		Pasta de C:\
5		NULL
6	27/05/2019 11:23	<DIR> temp
7	27/05/2019 12:54	<DIR> teste
8	0 arquivo(s)	0 bytes
9	2 pasta(s)	398.177.484.800 bytes disponíveis
10		NULL

Crie um subdiretório, caso não exista:

```
DECLARE @result int
EXEC @result = xp_cmdshell 'dir C:\TESTE\BANCO'
IF (@result <> 0)
    Exec master.dbo.xp_cmdshell 'MD C:\TESTE\BANCO'
```

Results		Messages
		output
1		O volume na unidade C não tem nome.
2		O Número de Série do Volume é 1AC5-57C8
3		NULL
4		Pasta de C:\TESTE
5		NULL
6		Arquivo não encontrado
7		NULL

		output
1		NULL

7.8. CLR STORED PROCEDURE

No SQL Server, é possível executar programas desenvolvidos em linguagem C# ou Visual Basic. O SQL utiliza as bibliotecas do .NET Framework **Common Language Runtime (CLR)**.

Para isso, é necessário que seja desenvolvido um Programa C# ou Visual:

```
using System;
using System.Data;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;

public class ProgramaC
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void RetornoC(out string text)
    {
        SqlContext.Pipe.Send("Programa C#" + Environment.
        NewLine);
        text = "Programa C#";
    }
}
```

No SQL Server, é necessário habilitar a utilização de códigos CLR:

```
SP_CONFIGURE 'clr enabled',1
```

Também é necessário anexar o novo ASSEMBLY. Para isso, expanda **DATABASES / PROGRAMMABILITY / ASSEMBLIES**. Com o botão direito, clique em **New Assemblies**.

New Assembly

Select a page

- General
- Permissions
- Extended Properties

Script ? Help

Assembly name:

Assembly owner:

Permission set:

Path to assembly: [Browse...](#)

Additional properties:

Misc

Creation Date	27/05/2019 13:15
Strong Name	False
Version	0.0

Connection

Server: localhost

Connection: instructor

[View connection properties](#)

Progress

Ready

Creation Date
When the assembly was created.

OK Cancel

Após a criação da PROCEDURE:

```
CREATE PROCEDURE PROGRAMA_DLL_C  
AS EXTERNAL NAME SQLCLR.StoredProcedures.RetornoC
```

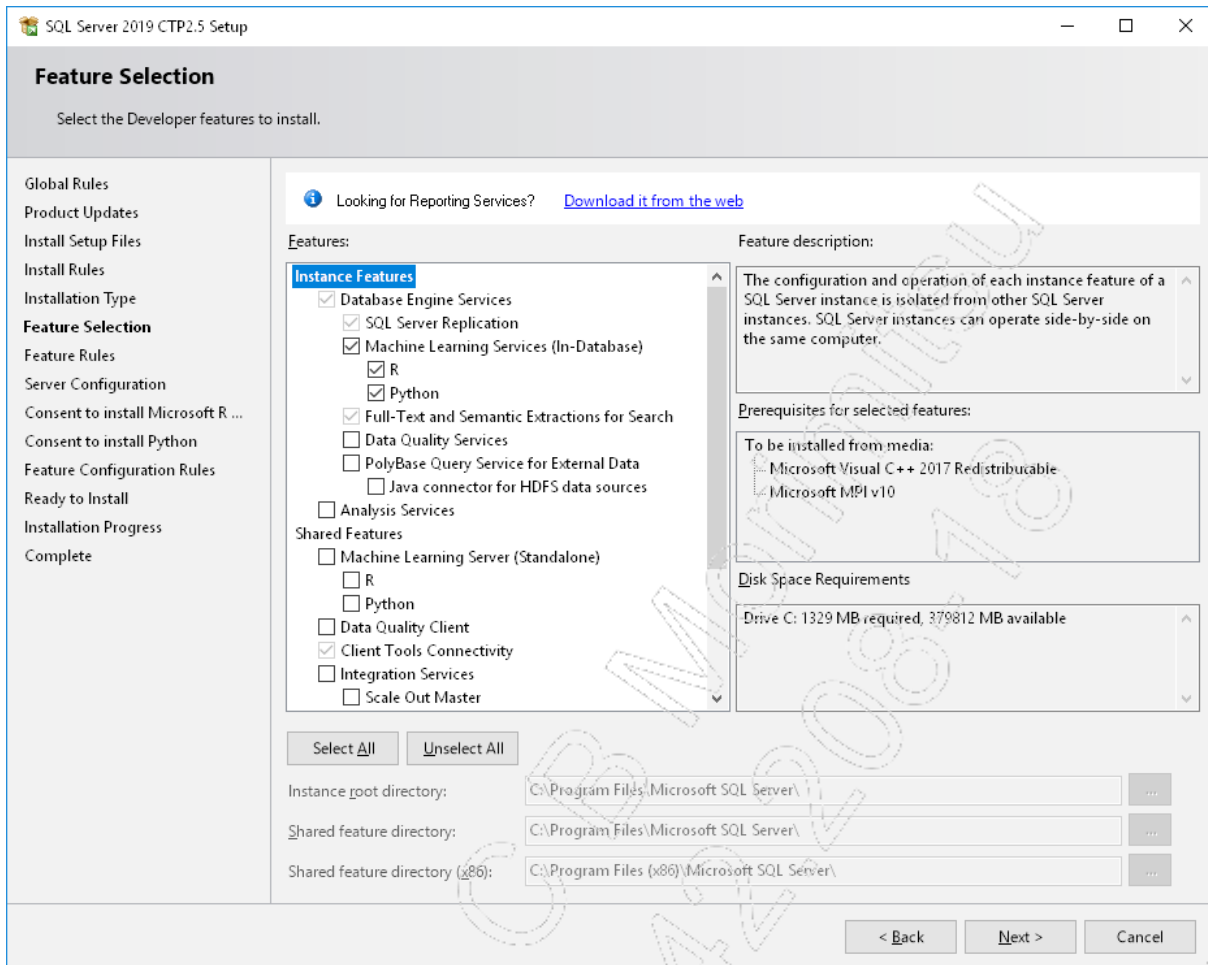
7.9.SP_EXECUTE_EXTERNAL_SCRIPT

O comando **SP_EXECUTE_EXTERNAL_SCRIPT** executa um programa: R, Python ou Java.

Para executar o comando, é necessário habilitá-lo no banco de dados:

```
exec sp_configure 'external scripts enabled', 1;
```


Para utilizar esse recurso, é necessário que sejam instalados os serviços de **R** e **Python**.



- Realizando um teste:

```
execute sp_execute_external_script
@language = N'Python',
@script = N'
print(1)
'
```

7.10. Compilação Nativa

Ao preparar um procedimento armazenado para que seja compilado nativamente, o código não será mais interpretado, mas sim compilado. A vantagem está na performance, pois esse processo diminui a preparação do comando.

Para que a procedure seja compilada nativamente:

- O banco deve possuir um FILEGROUP otimizado em memória;
- STORED PROCEDURES normais são compiladas na primeira execução, enquanto as STORED PROCEDURES compiladas são compiladas no momento da criação;
- STORED PROCEDURES compiladas são recompiladas no momento de reinício do serviço do SQL Server;
- As tabelas acessadas devem ser otimizadas ou compiladas em memória;
- Não suporta o comando PRINT.

Verifique o exemplo adiante:

```
GO
CREATE PROCEDURE DBO.SP_PROCEDURE_COMPILADA
    WITH    NATIVE_COMPILATION,
           SCHEMABINDING
AS
BEGIN ATOMIC
    WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'US_ENGLISH')

    DECLARE @CONT INT = 0, @SOMA INT = 0

    WHILE @CONT <= 1000
    BEGIN
        SET @SOMA += @CONT
        SET @CONT += 1
    END

    SELECT @SOMA AS SOMA_DE_1_A_1000

END;
GO
```

Resultado:

Results		Messages	
	Soma_de_1_a_1000		
1	500500		

Pontos principais

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

- 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**;
- Podemos executar comando **SHELL**, **PYTHON**, **R** e **JAVA** dentro do SQL SERVER;
- É possível compilar nativamente uma **PROCEDURE**. Ao compilar nativamente, ganhamos performance na execução.



Stored procedures

Teste seus conhecimentos





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.

Bruma 3971-18422008-18
Morinitsu

4. Qual é a função do comando XP_CMDSHELL?

- ☐ a) Executa um comando qualquer.
- ☐ b) Executa um comando SHELL.
- ☐ c) Não existe esse comando no SQL.
- ☐ d) Esse comando é antigo, de quando o SQL era executado no DOS.
- ☐ e) É um comando para configuração de comandos de erro.

5. Qual a vantagem de compilação nativa de PROCEDURES?

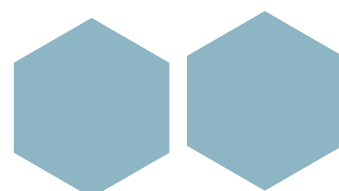
- ☐ a) Segurança.
- ☐ b) Evita o acesso direto nas tabelas.
- ☐ c) Simplifica o código.
- ☐ d) É mais rápida, pois o código já fica compilado.
- ☐ e) Evita a utilização de programas externos.



Stored procedures



Mãos à obra!



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: **2017**, **2018** e **2019**;
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_TIPOPORDUTO**);
6. Insira os tipos **TESTE** e **TESTE2**;
7. Crie uma procedure que exclua um tipo de produto (**TB_TIPOPORDUTO**). 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: **2017**, **2018** e **2019**;
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)
```

Laboratório 3

A – Trabalhando com PROCEDURES compiladas nativamente

Para a realização deste exercício, siga o roteiro adiante:

1. Execute o comando de criação da tabela otimizada em memória:

```
CREATE TABLE TB_RESUMO_VENDAS (  
    ID_RESUMO INT NOT NULL IDENTITY PRIMARY KEY NONCLUSTERED,  
    ANO INT,  
    MES INT,  
    TOTAL DECIMAL(10,2)  
)  
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_ONLY)  
GO
```

2. Insira os valores resumidos da tabela **TB_PEDIDO** na tabela **TB_RESUMO_VENDA**;
3. Crie uma PROCEDURE que retorne os dados por ano, ordenado por mês. Essa procedure deve ser compilada nativamente;
4. Execute a PROCEDURE para testar o retorno dos dados.



Funções

- Funções e STORED PROCEDURES;
- Funções definidas pelo usuário;
- Funções escalares;
- Funções tabulares;
- Campos computados com funções.



8.1. 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.

Neste capítulo, abordaremos diversos aspectos relativos às funções do SQL e sua utilização.

8.2. 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**.

8.3. 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.

! Não se pode criar uma função de usuário com uma cláusula **OUTPUT INTO** que tenha como destino uma tabela.

8.4. 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.

Adiante, temos diversos exemplos de funções escalares:

```
USE db_Ecommerce
```

- 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 = DATENAME( 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( '2017.11.12' )

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:

```
-----  
SELECT 10 + (20-10) * RAND()  
SELECT 5 + (10-5) * RAND()  
SELECT 1 + (10-1) * RAND()  
-- OBS.: @MIN + (@MAX - @MIN + 1) * RAND()  
-----  
  
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, ID_DEPARTAMENTO, ID_CARGO, SALARIO, DATA_ADMISSAO)
VALUES ('ZÉ LUIZ', 1, 1, 1000, GETDATE())

INSERT INTO TB_EMPREGADO
(NOME, ID_DEPARTAMENTO, ID_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 = '2019.05.28'
```

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 '2019.05.28' AND '2019.05.28
23:59'
```

A função **FN_TRUNCA_DATA** criada a seguir resolve esse problema:

```
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 ) = '2019.05.28'
-- 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 20/03/2017 é 01/03/2017):

```
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 ( '2019.1.25', 'ANIV. SÃO PAULO' )
INSERT INTO FERIADOS VALUES ( '2019.3.4', 'CARNAVAL' )
INSERT INTO FERIADOS VALUES ( '2019.3.5', 'CARNAVAL' )
INSERT INTO FERIADOS VALUES ( '2019.3.6', 'CARNAVAL' )
INSERT INTO FERIADOS VALUES ( '2019.4.19', 'PÁSCOA' )
INSERT INTO FERIADOS VALUES ( '2019.4.21', 'TIRADENTES' )
INSERT INTO FERIADOS VALUES ( '2019.5.1', 'TRABALHO' )
INSERT INTO FERIADOS VALUES ( '2019.6.20', 'CORPUS CHRISTI' )
```

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, '2019.1.1', '2019.1.15')
```


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, '2019.12.20', '2019.1.15')
```

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
        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', '2019.1.1', '2019.1.15')
SELECT DBO.FN_DIF_DATAS('M', '2018.1.2', '2019.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2018.1.20', '2019.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2018.8.2', '2019.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2017.8.2', '2019.2.15')
SELECT DBO.FN_DIF_DATAS('M', '2017.8.20', '2019.2.15')
SELECT DBO.FN_DIF_DATAS('A', '2018.8.2', '2019.2.15')
SELECT DBO.FN_DIF_DATAS('A', '2016.8.2', '2019.10.15')
SELECT DBO.FN_DIF_DATAS('A', '2016.10.2', '2019.10.15')
SELECT DBO.FN_DIF_DATAS('A', '2016.11.2', '2019.10.15')
SELECT DBO.FN_DIF_DATAS('A', '2016.10.20', '2019.10.20')
```

- 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:

```

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%'

```

8.5. 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**.

8.5.1. 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 T-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**.

Adiante, temos um exemplo de função tabular:

- 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(ID_DEPARTAMENTO) FROM TB_DEPARTAMENTO
  SET @I = 1;
  WHILE @I <= @TOT
  BEGIN
    -- Se existir o departamento de código = @I...
    IF EXISTS( SELECT * FROM TB_DEPARTAMENTO
               WHERE ID_DEPARTAMENTO = @I )
    BEGIN
      -- Inserir na tabela de retorno os funcionários do
      -- departamento código @I
      INSERT INTO @TotDeptos
      SELECT ID_DEPARTAMENTO, NOME, 'D', SALARIO
      FROM TB_EMPREGADO WHERE ID_DEPARTAMENTO = @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 ID_DEPARTAMENTO, 'TOTAL DO DEPTO.: ', 'T',
             SUM( SALARIO )
      FROM TB_EMPREGADO WHERE ID_DEPARTAMENTO = @I
      GROUP BY ID_DEPARTAMENTO;
    END -- IF EXISTS
    SET @I = @I + 1;
  END -- WHILE
  RETURN
END -- FUNCTION

GO

SELECT * FROM FN_TOT_DEPTOS()
```

Repare que o total do departamento será mostrado na consulta:

	COD_DEPTO	NOME	TIPO	VALOR
1	1	CASSIANO	D	1200.00
2	1	CARLOS FERNANDO	D	8300.00
3	1	JOSÉ	D	3300.00
4	1	JOÃO	D	5000.00
5	1	ROBERTO ALEXANDRO	D	800.00
6	1	JORGE	D	4500.00
7	1	ARNALDO	D	890.00
8	1	ROGÉRIO	D	4500.00
9	1	RONALDO	D	3300.00
10	1	JOSÉ	D	8300.00
11	1	PEDRO	D	890.00

8.5.2. 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.

Adiante, temos um exemplo de função tabular:

- 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( '2017.1.1', '2017.12.31' )
ORDER BY ANO, MES
```

	MES	ANO	MAIOR_VALOR
1	1	2017	17998.70
2	2	2017	9531.01
3	3	2017	13233.50
4	4	2017	22871.25
5	5	2017	11134.00
6	6	2017	8551.76
7	7	2017	9822.43
8	8	2017	8726.26
9	9	2017	9542.17
10	10	2017	9860.86
11	11	2017	10229.01
12	12	2017	9420.35

8.6. Campos computados com funções

No capítulo 2, 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_ITENSPEDIDO**. 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_ITENSPEDIDO  
WHERE ID_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(ID_PEDIDO)
```

Realizando a consulta, o campo **VALOR_CALC** realiza o cálculo automaticamente:

```
SELECT ID_PEDIDO, VLR_TOTAL, VALOR_CALC FROM TB_PEDIDO
```

Resultado:

	ID_PEDIDO	VLR_TOTAL	VALOR_CALC
1	1	2.00	NULL
2	2	30.70	30.70
3	3	59.76	59.76
4	4	603.25	603.25
5	5	72.39	72.39
6	6	970.57	970.57
7	7	153.90	153.90
8	8	874.00	874.00

Pontos principais

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

- Uma **função (FUNCTION)** é um objeto que contém um bloco de comandos T-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**;
- 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**;
- Podemos criar campos calculados que utilizam funções para trazer as informações relacionadas com outros campos.



Funções

Teste seus conhecimentos



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. Verifique a função adiante e escolha qual é a afirmação correta:

```
CREATE FUNCTION FN_Calcula (@ANO INT, @TOTAL DECIMAL(10,2) )
    RETURNS VARCHAR(20)
AS BEGIN
    DECLARE @RET VARCHAR(20);

    INSERT INTO TB_CALCULO VALUES ( @ANO , @TOTAL)

    IF @@ERROR
        SET @RET = 'ERRO'
    ELSE
        SET @RET = 'SUCESSO'

    RETURN (@RET)
END
```

- ☐ a) A função retorna um erro quando não consegue inserir.
- ☐ b) Gera um erro de sintaxe.
- ☐ c) A função insere valores na tabela TB_CALCULO.
- ☐ d) A função insere valores na tabela TB_CALCULO e mostra se ocorreu um erro.
- ☐ e) Utilize TRY CATCH em vez de @@ERROR.

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)
```

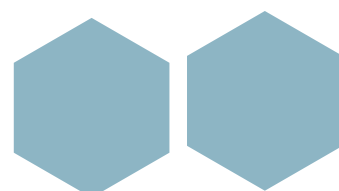
- ☐ a) 4
- ☐ b) 5
- ☐ c) 20
- ☐ d) 0 (zero), pois não é possível converter um INT em FLOAT.
- ☐ e) NULO.



Funções



Mãos à obra!



Laboratório 1

A – Trabalhando com funções

1. Crie uma função, chamada **FN_HORA**, que retorne o valor da HORA do servidor;
2. Crie uma função **FN_DATA** que, passando um valor **DATETIME**, retorne somente a data, truncando a hora;
3. Crie uma função que receba uma data como parâmetro e que retorne o nome do mês dessa data em português;
4. 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.

5. 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) )
```

6. 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 TB_ITENSPEDIDO I
JOIN TB_PRODUTO Pr ON I.ID_PRODUTO=Pr.ID_PRODUTO
JOIN TB_PEDIDO Pe ON I.ID_PEDIDO=Pe.ID_PEDIDO
WHERE
    Pe.DATA_EMISSAO BETWEEN @DT1 AND @DT2
GROUP BY Pr.ID_PRODUTO, Pr.DESCRICAO
```

7. Crie uma função que receba o código do cliente e retorne a quantidade de pedidos realizados;
8. Realize uma consulta que apresente o Código do Cliente, Nome do cliente e a quantidade de pedidos realizados.

B – 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.

9

Triggers

- Triggers;
- Triggers DML;
- Triggers DDL;
- Trigger de logon;
- Aninhamento de triggers;
- Recursividade de triggers.



9.1. Introdução

O SQL Server oferece dois mecanismos primários para reforçar as regras de negócio e a integridade dos dados: as **CONSTRAINTS** e os **TRIGGERS**.

Neste capítulo, serão descritos os tipos de triggers que o SQL Server oferece e que podem ser criados, e as ações executadas por eles.

9.2. 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.

9.2.1. 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.

9.2.2. 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 dos 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.

9.2.3. 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**.

9.2.4. 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.

9.2.5. 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.

9.2.5.1. 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.

9.2.5.2. 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.

9.2.5.3. 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**.

9.3. 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.

9.3.1. 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

9.3.2. 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.

9.3.3. 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.

9.3.4. 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.

9.3.5. 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 sobrescritas. 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:

```
CREATE TABLE EMPREGADOS_HIST_SALARIO
(
    NUM_MOVTO          INT IDENTITY,
    ID_EMPREGADO        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 = ID_EMPREGADO, @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
(ID_EMPREGADO, DATA_ALTERACAO, SALARIO_ANTIGO,
SALARIO_NOVO)
VALUES
(@CODFUN, GETDATE(), @SALARIO_ANTIGO, @SALARIO_
NOVO)
END

-- Testar
UPDATE TB_EMPREGADO SET SALARIO = SALARIO * 1.2
WHERE ID_EMPREGADO = 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 ID_EMPREGADO 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_ANTI = 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
    (ID_EMPREGADO, DATA_ALTERACAO, SALARIO_ANTI, SALARIO_NOVO)
    SELECT I.ID_EMPREGADO, GETDATE(), D.SALARIO, I.SALARIO
    FROM INSERTED I JOIN DELETED D ON I.ID_EMPREGADO = D.ID_
EMPREGADO
    WHERE I.SALARIO <> D.SALARIO
END

-- Testar
DELETE FROM EMPREGADOS_HIST_SALARIO

UPDATE TB_EMPREGADO SET SALARIO = SALARIO * 1.2
WHERE ID_EMPREGADO IN (4,5,7)
--
SELECT * FROM EMPREGADOS_HIST_SALARIO
```

O resultado é o seguinte:

	NUM_MOVTO	ID_EMPREGADO	DATA_ALTERACAO	SALARIO_ANTI	SALARIO_NOVO
1	3	7	2019-05-28 13:02:48.433	5400.00	6480.00
2	4	5	2019-05-28 13:02:48.433	1440.00	1728.00
3	5	4	2019-05-28 13:02:48.433	720.00	864.00

- 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 ID_PEDIDO = P.ID_PEDIDO)
  FROM TB_PEDIDO P JOIN DELETED D
    ON P.ID_PEDIDO = D.ID_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 ID_PEDIDO = P.ID_PEDIDO)
  FROM TB_PEDIDO P JOIN INSERTED I
    ON P.ID_PEDIDO = I.ID_PEDIDO
END

-- Testar
SELECT * FROM TB_PEDIDO WHERE ID_PEDIDO = 1000
-- Pedido 1000 -> VLR_TOTAL = 380
SELECT * FROM TB_ITENSPEDIDO WHERE ID_PEDIDO = 1000
-- Possui um único item com PR_UNITARIO = 1
UPDATE TB_ITENSPEDIDO SET PR_UNITARIO = 2
WHERE ID_PEDIDO = 1000
--
SELECT * FROM TB_PEDIDO WHERE ID_PEDIDO = 1000
-- VLR_TOTAL = 760
```

Resultado após a execução:

	ID_PEDIDO	ID_CLIENTE	ID_EMPREGADO	DATA_EMISSAO	VLR_TOTAL	STATUS	OBSERVACAO	VALOR_CALC
1	1000	48	3	2017-05-28 00:00:00.000	760.00	E	NULL	760.00

• 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.ID_CLIENTE = D.ID_
CLIENTE
END

-- TESTAR
SELECT ID_CLIENTE, NOME, SN_ATIVO FROM TB_CLIENTE
--
DELETE FROM TB_CLIENTE WHERE ID_CLIENTE IN (4,7,11)
--
SELECT ID_CLIENTE, NOME, SN_ATIVO FROM TB_CLIENTE
WHERE ID_CLIENTE IN (4,7,11)
```

Na figura a seguir, podemos conferir o resultado:

	ID_CLIENTE	NOME	SN_ATIVO
1	4	QCQQBPNPUKNHDSFBMEKESJNMJMQDDP	N
2	7	TCSGVJISYISLFFELSMMLYSBMPRQNK	N
3	11	OVRUUMSLRNTIJVGPCVLHFJQTKRSDKD	N

9.4.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 T-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.

9.4.1.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:

Eventos no nível de banco de dados		
CREATE_APPLICATION_ROLE	ALTER_APPLICATION_ROLE	DROP_APPLICATION_ROLE
CREATE_ASSEMBLY	ALTER_ASSEMBLY	DROP_ASSEMBLY
CREATEASYMMETRICKEY	ALTERASYMMETRICKEY	DROPASYMMETRICKEY
ALTER_AUTHORIZATION	ALTER_AUTHORIZATION_DATABASE	
CREATE_BROKER_PRIORITY	CREATE_BROKER_PRIORITY	CREATE_BROKER_PRIORITY
CREATE_CERTIFICATE	ALTER_CERTIFICATE	DROP_CERTIFICATE
CREATE_CONTRACT	DROP_CONTRACT	
CREATE_CREDENTIAL	ALTER_CREDENTIAL	DROP_CREDENTIAL
GRANT_DATABASE	DENY_DATABASE	REVOKE_DATABASE
CREATE_DATABASE_AUDIT_SPECIFICATION	ALTER_DATABASE_AUDIT_SPECIFICATION	DENY_DATABASE_AUDIT_SPECIFICATION
CREATE_DATABASE_ENCRYPTION_KEY	ALTER_DATABASE_ENCRYPTION_KEY	DROP_DATABASE_ENCRYPTION_KEY
CREATE_DEFAULT	DROP_DEFAULT	
BIND_DEFAULT	UNBIND_DEFAULT	
CREATE_EVENT_NOTIFICATION	DROP_EVENT_NOTIFICATION	
CREATE_EXTENDED_PROPERTY	ALTER_EXTENDED_PROPERTY	DROP_EXTENDED_PROPERTY
CREATE_FULLTEXT_CATALOG	ALTER_FULLTEXT_CATALOG	DROP_FULLTEXT_CATALOG
CREATE_FULLTEXT_INDEX	ALTER_FULLTEXT_INDEX	DROP_FULLTEXT_INDEX
CREATE_FULLTEXT_STOPLIST	ALTER_FULLTEXT_STOPLIST	DROP_FULLTEXT_STOPLIST
CREATE_FUNCTION	ALTER_FUNCTION	DROP_FUNCTION
CREATE_INDEX	ALTER_INDEX	DROP_INDEX
CREATE_MASTER_KEY	ALTER_MASTER_KEY	DROP_MASTER_KEY
CREATE_MESSAGE_TYPE	ALTER_MESSAGE_TYPE	DROP_MESSAGE_TYPE
CREATE_PARTITION_FUNCTION	ALTER_PARTITION_FUNCTION	DROP_PARTITION_FUNCTION
CREATE_PARTITION_SCHEME	ALTER_PARTITION_SCHEME	DROP_PARTITION_SCHEME
CREATE_PLAN_GUIDE	ALTER_PLAN_GUIDE	DROP_PLAN_GUIDE
CREATE_PROCEDURE	ALTER_PROCEDURE	DROP_PROCEDURE
CREATE_QUEUE	ALTER_QUEUE	DROP_QUEUE
CREATE_REMOTE_SERVICE_BINDING	ALTER_REMOTE_SERVICE_BINDING	DROP_REMOTE_SERVICE_BINDING
CREATE_SPATIAL_INDEX		
RENAME		

Eventos no nível de banco de dados

CREATE_ROLE	ALTER_ROLE	DROP_ROLE
ADD_ROLE_MEMBER	DROP_ROLE_MEMBER	
CREATE_ROUTE	ALTER_ROUTE	DROP_ROUTE
CREATE_RULE	DROP_RULE	
BIND_RULE	UNBIND_RULE	
CREATE_SCHEMA	ALTER_SCHEMA	DROP_SCHEMA
CREATE_SEARCH_PROPERTY_LIST	ALTER_SEARCH_PROPERTY_LIST	DROP_SEARCH_PROPERTY_LIST
CREATE_SEQUENCE_EVENTS	CREATE_SEQUENCE_EVENTS	CREATE_SEQUENCE_EVENTS
CREATE_SERVER_ROLE	ALTER_SERVER_ROLE	DROP_SERVER_ROLE
CREATE_SERVICE	ALTER_SERVICE	DROP_SERVICE
ALTER_SERVICE_MASTER_KEY	BACKUP_SERVICE_MASTER_KEY	RESTORE_SERVICE_MASTER_KEY
ADD_SIGNATURE	DROP_SIGNATURE	
ADD_SIGNATURE_SCHEMA_OBJECT	DROP_SIGNATURE_SCHEMA_OBJECT	
CREATE_SPATIAL_INDEX	ALTER_INDEX	DROP_INDEX
CREATE_STATISTICS	DROP_STATISTICS	UPDATE_STATISTICS
CREATE_SYMMETRIC_KEY	ALTER_SYMMETRIC_KEY	DROP_SYMMETRIC_KEY
CREATE_SYNONYM	DROP_SYNONYM	
CREATE_TABLE	ALTER_TABLE	DROP_TABLE
CREATE_TRIGGER	ALTER_TRIGGER	DROP_TRIGGER
CREATE_TYPE	DROP_TYPE	
CREATE_USER	ALTER_USER	DROP_USER
CREATE_VIEW	ALTER_VIEW	DROP_VIEW
CREATE_XML_INDEX	ALTER_INDEX	DROP_INDEX
CREATE_XML_SCHEMA_COLLECTION	ALTER_XML_SCHEMA_COLLECTION	DROP_XML_SCHEMA_COLLECTION

Eventos no nível do servidor

ALTER_AUTHORIZATION_SERVER	ALTER_SERVER_CONFIGURATION	ALTER_INSTANCE
CREATE_AVAILABILITY_GROUP	ALTER_AVAILABILITY_GROUP	DROP_AVAILABILITY_GROUP
CREATE_CREDENTIAL	ALTER_CREDENTIAL	DROP_CREDENTIAL
CREATE_CRYPTOGRAPHIC_PROVIDER	ALTER_CRYPTOGRAPHIC_PROVIDER	DROP_CRYPTOGRAPHIC_PROVIDER
CREATE_DATABASE	ALTER_DATABASE	DROP_DATABASE
CREATE_ENDPOINT	ALTER_ENDPOINT	DROP_ENDPOINT
CREATE_EVENT_SESSION	ALTER_EVENT_SESSION	DROP_EVENT_SESSION
CREATE_EXTENDED_PROCEDURE	DROP_EXTENDED_PROCEDURE	
CREATE_LINKED_SERVER	ALTER_LINKED_SERVER	DROP_LINKED_SERVER
CREATE_LINKED_SERVER_LOGIN	DROP_LINKED_SERVER_LOGIN	
CREATE_LOGIN	ALTER_LOGIN	DROP_LOGIN
CREATE_MESSAGE	ALTER_MESSAGE	DROP_MESSAGE
CREATE_REMOTE_SERVER	ALTER_REMOTE_SERVER	DROP_REMOTE_SERVER

Eventos no nível do servidor		
CREATE_RESOURCE_POOL	ALTER_RESOURCE_POOL	DROP_RESOURCE_POOL
GRANT_SERVER	DENY_SERVER	REVOKE_SERVER
ADD_SERVER_ROLE_MEMBER	DROP_SERVER_ROLE_MEMBER	
CREATE_SERVER_AUDIT	ALTER_SERVER_AUDIT	DROP_SERVER_AUDIT
CREATE_SERVER_AUDIT_SPECIFICATION	ALTER_SERVER_AUDIT_SPECIFICATION	DROP_SERVER_AUDIT_SPECIFICATION
CREATE_WORKLOAD_GROUP	ALTER_WORKLOAD_GROUP	DROP_WORKLOAD_GROUP

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/ObjectTyp[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
```

9.5.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**:

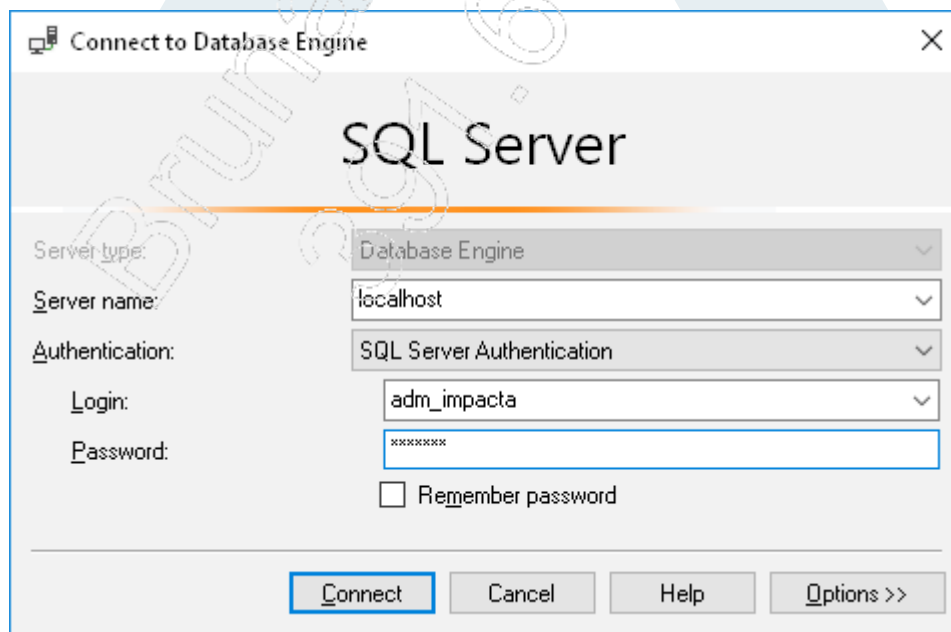
```
CREATE LOGIN Adm_Impacta WITH PASSWORD = 'Imp@ct@12'
```

- Criação do trigger que bloqueia qualquer acesso através do SSMS com usuários de aplicação:

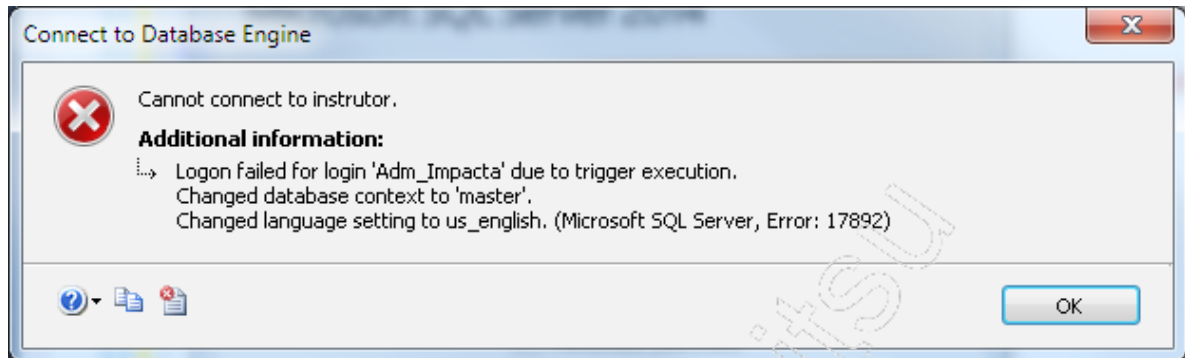
```
CREATE TRIGGER TRG_Bloqueio_SSMS
ON ALL SERVER
FOR LOGON
AS
BEGIN

    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_AuditLogin  
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	I 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	

9.6. 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.

9.6.1. 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**.

9.7. 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, conseqüentemente, 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 do capítulo.

- 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 login**;
- 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





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

Brúna C B Morimits
391.642.208-18

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?

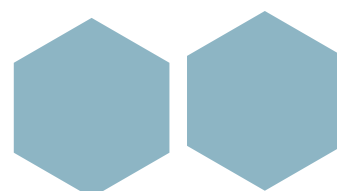
- ☐ a) Podemos utilizar a tabela UPDATED para verificar as informações dos registros.
- ☐ b) A tabela UPDATED está disponível somente em triggers.
- ☐ c) Ao efetuarmos uma atualização do SQL, é disponibilizada a tabela INSERTED com as informações da atualização.
- ☐ d) As tabelas INSERTED e DELETED estão 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!



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** que faça um **ROLLBACK**, caso exista uma tentativa de inserção de valor de preço de venda menor que zero na tabela **TB_PRODUTO**. O **TRIGGER** deve ser para as ações: **INSERT** e **UPDATE**;

4. Crie a tabela de auditoria da tabela de cargos a seguir:

```
CREATE TABLE TB_AUDITORIA_CARGO
(
  ID                INT IDENTITY PRIMARY KEY,
  DATA            DATETIME,
  USUARIO          VARCHAR(20),
  CARGO_ANTIGO     VARCHAR(30),
  CARGO_NOVO       VARCHAR(30),
  SALARIO_INIC_ANTIGO DECIMAL(10,2),
  SALARIO_INIC_NOVO DECIMAL(10,2),
  ACAO             CHAR(1),
)
```

5. Desenvolva um **TRIGGER** que carregue a tabela criada anteriormente com as informações dos comandos **INSERT**, **UPDATE** e **DELETE** da tabela **TB_CARGO**. Para retornar o nome do usuário, utilize a função: **Suser_SName()**;

6. Crie um trigger de DDL para o banco de dados **DB_ECOMMERCE** 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)
)
```

7. Crie um trigger de LOGON para gravação de auditoria para acesso ao servidor. Utilize a tabela:

```
CREATE TABLE MASTER.DBO.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
)
GO
```


10

Acesso a recursos externos

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



10.1.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.

10.2.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;

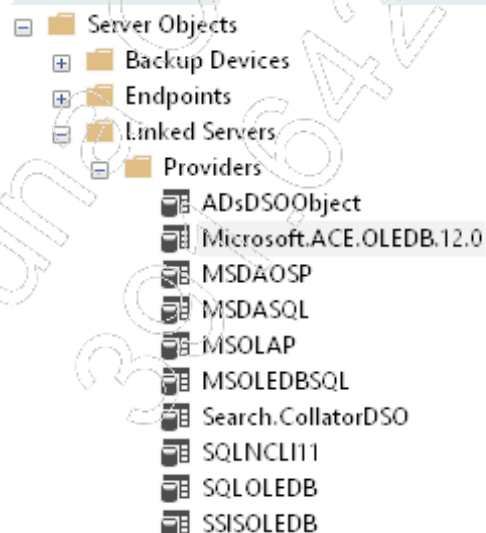
3. Acesse o Excel e crie a tabela **PESSOA** no SQL Server:

```
CREATE DATABASE DB_EXTERNO
GO
USE DB_EXTERNO
CREATE TABLE TB_PESSOA
(
    COD INT,
    NOME VARCHAR(50)
)
```

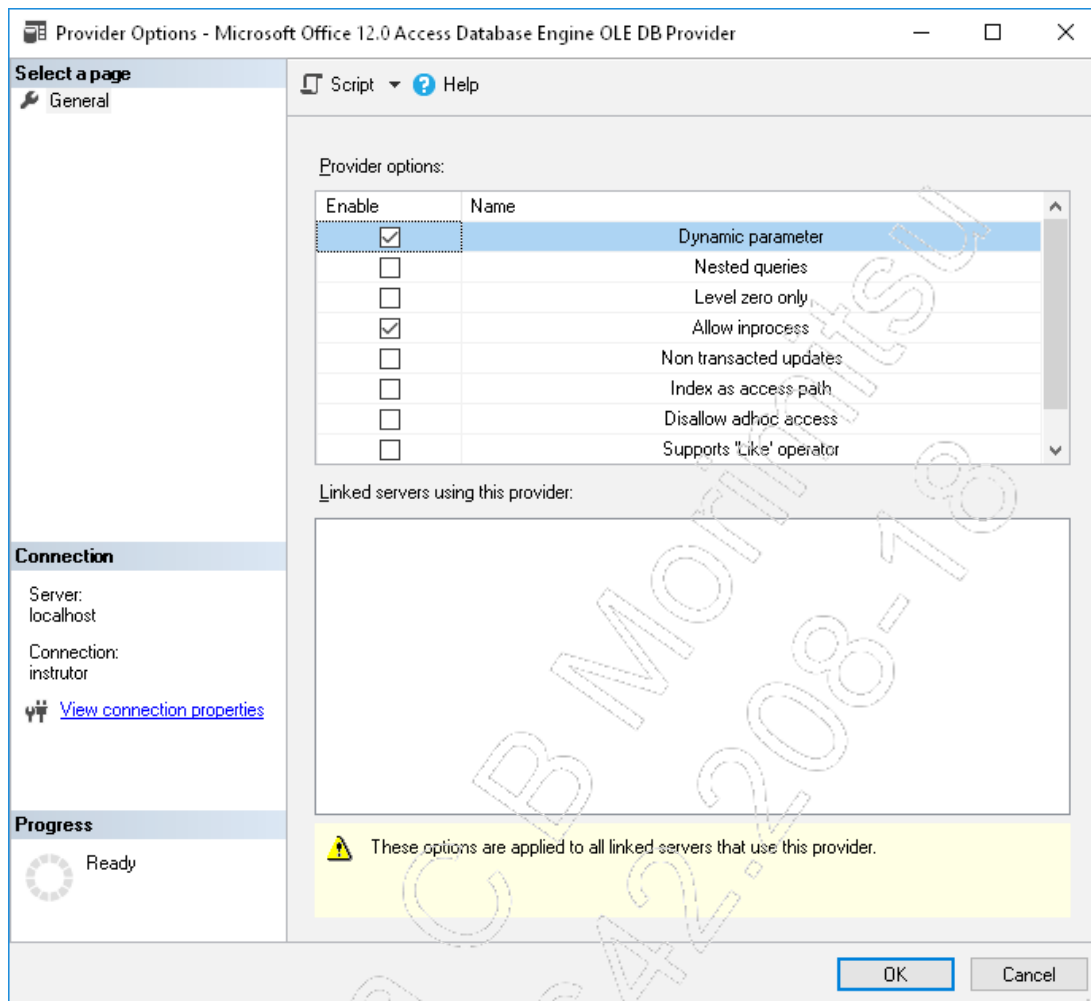
Na planilha do Excel **PESSOA**, existem as seguintes informações:

COD	NOME
1	Antonio da Silva
2	Carlos Eduardo
3	Elias Ricardo
4	Caio Caneta
5	Silvia Maria
6	Ana Carla
7	Carlos de Souza
8	Altino Antunes
9	Alex Carlos
10	Maria das Rosas

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.XLS',
'SELECT COD, NOME FROM [CLIENTE$]')
```

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

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

SELECT * FROM TB_PESSOA
```

9. Exporte dados para o Excel:

```
--EXPORTAR DADOS
INSERT INTO OPENROWSET('Microsoft.ACE.OLEDB.12.0',
'Excel 12.0;Database=C:\DADOS\PESSOA.XLS',
'SELECT COD, NOME FROM [CLIENTE$]')
VALUES
(11 , 'JOSÉ MARIA'),
(12 , 'MARIA JOSÉ'),
(13 , 'ANA MARIA')

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

Resultado da consulta no Excel:

Results		Messages
	COD	NOME
1	1	Antonio da Silva
2	2	Carlos Eduardo
3	3	Elias Ricardo
4	4	Caio Caneta
5	5	Silvia Maria
6	6	Ana Carla
7	7	Carlos de Souza
8	8	Altino Antunes
9	9	Alex Carlos
10	10	Maria das Rosas
11	11	JOSÉ MARIA
12	12	MARIA JOSÉ
13	13	ANA MARIA

10. Acesse o Access e consulte a tabela **PRODUTOS**:

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

11. Crie uma **JOIN** entre tabelas do Access:

```
SELECT P.COD_PRODUTO, P.DESCRICAO, U.UNIDADE
FROM OPENROWSET('MICROSOFT.ACE.OLEDB.12.0',
    'C:\DADOS\PEDIDOS.ACCDB';
    'ADMIN';'', PRODUTOS) AS P
JOIN OPENROWSET('MICROSOFT.ACE.OLEDB.12.0',
    'C:\DADOS\PEDIDOS.ACCDB';
    'ADMIN';'', UNIDADES) AS U
ON P.COD_UNIDADE = U.COD_UNIDADE
```

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
```

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

10.3.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' | 'widenative' } ]
    [ [ , ] FIELDTERMINATOR = 'field_terminator' ]
    [ [ , ] FIRSTROW = first_row ]
    [ [ , ] FIRE_TRIGGERS ]
    [ [ , ] FORMATFILE = 'format_file_path' ]
    [ [ , ] KEEPIDENTITY ]
    [ [ , ] KEEPNULLS ]
    [ [ , ] 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 TB_BULK_INSERT  
( CODIGO          INT,  
  NOME            VARCHAR(40),  
  DATA_NASCIMENTO DATETIME )
```

- **Execução do comando BULK INSERT**

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

- Consulta da tabela

```
SELECT * FROM TB_BULK_INSERT
```

Resultado:

Results		Messages	
	CODIGO	NOME	DATA_NASCIMENTO
1	1	ANTONIO CARLOS	1974-11-12 00:00:00.000
2	2	ELIAS MARIA	1963-07-01 00:00:00.000
3	3	EDUARDO JOAQUIM	1984-02-05 00:00:00.000
4	4	DIEGO APARECIDO	1999-10-29 00:00:00.000

10.4.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.

10.4.1.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 db_Ecommerce
GO
```

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


O resultado é apresentado em um link XML:

Results	Messages
XML_F52E2B61-18A1-11d1-B105-00805F49916B	
1	<row ID_PRODUTO="1" COD_TIPO="1" DESCRICAO="ABRIDOR...

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

```
<row ID_PRODUTO="1" ID_TIPO="1" DESCRICAO="ABRIDOR SACA-ROLHA TESTE ADO" PRECO_VENDA="0.0570" />
<row ID_PRODUTO="2" ID_TIPO="2" DESCRICAO="PORTA-LAPIS COM PEZINHO" PRECO_VENDA="1.8396" />
<row ID_PRODUTO="3" ID_TIPO="3" DESCRICAO="REGUA DE 20 CM" PRECO_VENDA="1.1532" />
<row ID_PRODUTO="4" ID_TIPO="4" DESCRICAO="PENTE PEQUENO" PRECO_VENDA="2.3063" />
<row ID_PRODUTO="5" ID_TIPO="4" DESCRICAO="PENTE JACARE" PRECO_VENDA="3.7340" />
<row ID_PRODUTO="6" ID_TIPO="5" DESCRICAO="SPECIAL PEN GOLD" PRECO_VENDA="5.0519" />
<row ID_PRODUTO="7" ID_TIPO="5" DESCRICAO="CANETA STAR I" PRECO_VENDA="0.8786" />
<row ID_PRODUTO="8" ID_TIPO="5" DESCRICAO="SPECIAL PEN STAR II" PRECO_VENDA="3.4870" />
<row ID_PRODUTO="9" ID_TIPO="5" DESCRICAO="CANETA SPECIAL CITRICA" PRECO_VENDA="3.5143" />
<row ID_PRODUTO="10" ID_TIPO="5" DESCRICAO="CANETA POP" PRECO_VENDA="2.4161" />
<row ID_PRODUTO="11" ID_TIPO="6" DESCRICAO="CHAVEIRO SUPER LENTE" PRECO_VENDA="0.3844" />
<row ID_PRODUTO="12" ID_TIPO="6" DESCRICAO="CHAVEIRO VEGAS" PRECO_VENDA="2.7456" />
<row ID_PRODUTO="13" ID_TIPO="6" DESCRICAO="CHAVEIRO BOMBA C/LOGO" PRECO_VENDA="4.6675" />
<row ID_PRODUTO="14" ID_TIPO="6" DESCRICAO="CHAVEIRO CADEADO" PRECO_VENDA="4.3655" />
<row ID_PRODUTO="15" ID_TIPO="6" DESCRICAO="CHAVEIRO NINJA" PRECO_VENDA="3.4045" />
<row ID_PRODUTO="16" ID_TIPO="1" DESCRICAO="ABRIDOR SEM FURO" PRECO_VENDA="2.1142" />
<row ID_PRODUTO="17" ID_TIPO="7" DESCRICAO="BOTTON CORACAO" PRECO_VENDA="4.3106" />
<row ID_PRODUTO="18" ID_TIPO="7" DESCRICAO="BOTTON GLOBO" PRECO_VENDA="0.0000" />
<row ID_PRODUTO="19" ID_TIPO="6" DESCRICAO="CHAVEIRO DOLLAR" PRECO_VENDA="4.7773" />
<row ID_PRODUTO="20" ID_TIPO="103" DESCRICAO="PAQUIMETRO" PRECO_VENDA="4.8871" />
<row ID_PRODUTO="21" ID_TIPO="9" DESCRICAO="PORTA MOEDAS" PRECO_VENDA="3.5143" />
```

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

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

Resultado:

```
<Produtos>
  <Produto ID_PRODUTO="1" ID_TIPO="1" DESCRICAO="ABRIDOR SACA-ROLHA TESTE ADO" PRECO_VENDA="0.0570" />
  <Produto ID_PRODUTO="2" ID_TIPO="2" DESCRICAO="PORTA-LAPIS COM PEZINHO" PRECO_VENDA="1.8396" />
  <Produto ID_PRODUTO="3" ID_TIPO="3" DESCRICAO="REGUA DE 20 CM" PRECO_VENDA="1.1532" />
  <Produto ID_PRODUTO="4" ID_TIPO="4" DESCRICAO="PENTE PEQUENO" PRECO_VENDA="2.3063" />
  <Produto ID_PRODUTO="5" ID_TIPO="4" DESCRICAO="PENTE JACARE" PRECO_VENDA="3.7340" />
  <Produto ID_PRODUTO="6" ID_TIPO="5" DESCRICAO="SPECIAL PEN GOLD" PRECO_VENDA="5.0519" />
  <Produto ID_PRODUTO="7" ID_TIPO="5" DESCRICAO="CANETA STAR I" PRECO_VENDA="0.8786" />
  <Produto ID_PRODUTO="8" ID_TIPO="5" DESCRICAO="SPECIAL PEN STAR II" PRECO_VENDA="3.4870" />
  <Produto ID_PRODUTO="9" ID_TIPO="5" DESCRICAO="CANETA SPECIAL CITRICA" PRECO_VENDA="3.5143" />
  <Produto ID_PRODUTO="10" ID_TIPO="5" DESCRICAO="CANETA POP" PRECO_VENDA="2.4161" />
  <Produto ID_PRODUTO="11" ID_TIPO="6" DESCRICAO="CHAVEIRO SUPER LENTE" PRECO_VENDA="0.3844" />
  <Produto ID_PRODUTO="12" ID_TIPO="6" DESCRICAO="CHAVEIRO VEGAS" PRECO_VENDA="2.7456" />
  <Produto ID_PRODUTO="13" ID_TIPO="6" DESCRICAO="CHAVEIRO BOMBA C/LOGO" PRECO_VENDA="4.6675" />
  <Produto ID_PRODUTO="14" ID_TIPO="6" DESCRICAO="CHAVEIRO CADEADO" PRECO_VENDA="4.3655" />
  <Produto ID_PRODUTO="15" ID_TIPO="6" DESCRICAO="CHAVEIRO NINJA" PRECO_VENDA="3.4045" />
  <Produto ID_PRODUTO="16" ID_TIPO="1" DESCRICAO="ABRIDOR SEM FURO" PRECO_VENDA="2.1142" />
  <Produto ID_PRODUTO="17" ID_TIPO="7" DESCRICAO="BOTTON CORACAO" PRECO_VENDA="4.3106" />
```


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 &.

```
SELECT ID_PRODUTO, ID_TIPO, DESCRICAO, PRECO_VENDA
FROM TB_PRODUTO
WHERE ID_PRODUTO = 1
FOR XML RAW('Produto'), ROOT('Produtos')
```

```
<Produtos>
  <Produto ID_PRODUTO="1" ID_TIPO="1" DESCRICAO="ABRIDOR SACA & ROLHA" PRECO_VENDA="0.0570" />
</Produtos>
```

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

```
SELECT ID_PRODUTO, ID_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 ID_TIPO é nulo e a tag simplesmente some:

```
<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>
  <ID_TIPO>100</ID_TIPO>
  <DESCRICAO>MAQUINA VARETAR</DESCRICAO>
  <PRECO_VENDA>5.1617</PRECO_VENDA>
</Produto>
<Produto>
  <ID_PRODUTO>39</ID_PRODUTO>
  <ID_TIPO>6</ID_TIPO>
  <DESCRICAO>PRISILHA</DESCRICAO>
  <PRECO_VENDA>1.0982</PRECO_VENDA>
</Produto>
<Produto>
  <ID_PRODUTO>40</ID_PRODUTO>
  <ID_TIPO>6</ID_TIPO>
  <DESCRICAO>ARGOLAS</DESCRICAO>
  <PRECO_VENDA>4.5852</PRECO_VENDA>
</Produto>
```

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

```
SELECT ID_PRODUTO, ID_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:

```
<Produtos>  
  <Produto>  
    <ID_PRODUTO>35</ID_PRODUTO>  
    <ID_TIPO>5</ID_TIPO>  
    <DESCRICAO>CANETA STAR II</DESCRICAO>  
    <PRECO_VENDA>4.7225</PRECO_VENDA>  
  </Produto>  
  <Produto>  
    <ID_PRODUTO>36</ID_PRODUTO>  
    <ID_TIPO xsi:nil="true" />  
    <DESCRICAO>KEY RING BIG LOCK</DESCRICAO>  
    <PRECO_VENDA xsi:nil="true" />  
  </Produto>  
  <Produto>  
    <ID_PRODUTO>37</ID_PRODUTO>  
    <ID_TIPO xsi:nil="true" />  
    <DESCRICAO>SPECIAL KEY RING</DESCRICAO>  
    <PRECO_VENDA xsi:nil="true" />  
  </Produto>  
  <Produto>  
    <ID_PRODUTO>38</ID_PRODUTO>  
    <ID_TIPO>100</ID_TIPO>  
    <DESCRICAO>MAQUINA VARETAR</DESCRICAO>  
    <PRECO_VENDA>5.1617</PRECO_VENDA>  
  </Produto>  
</Produtos>
```

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

```
-- o nome da coluna dá nome ao elemento de cada campo
SELECT Empregado.ID_EMPREGADO AS Codigo, 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>
    <Codigo>1</Codigo>
    <NOME>OLAVO </NOME>
    <DATA_ADMISSAO>2009-11-20T00:00:00</DATA_ADMISSAO>
    <SALARIO>3000.00</SALARIO>
  </Empregado>
  <Empregado>
    <Codigo>2</Codigo>
    <NOME>JOSE </NOME>
    <DATA_ADMISSAO>2017-07-24T00:00:00</DATA_ADMISSAO>
    <SALARIO>660.00</SALARIO>
  </Empregado>
  <Empregado>
    <Codigo>3</Codigo>
    <NOME>MARCELO </NOME>
    <DATA_ADMISSAO>2013-06-15T00:00:00</DATA_ADMISSAO>
    <SALARIO>2880.00</SALARIO>
  </Empregado>
  <Empregado>
    <Codigo>4</Codigo>
    <NOME>PAULO </NOME>
    <DATA_ADMISSAO>2011-07-03T00:00:00</DATA_ADMISSAO>
    <SALARIO>864.00</SALARIO>
  </Empregado>
</Empregados>
```

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

```
SELECT
Empregado.ID_EMPREGADO, Empregado.NOME, Empregado.DATA_
ADMISSAO,
Empregado.SALARIO, Dependente.ID_DEPENDENTE, Dependente.NOME,
Dependente.DATA_NASCIMENTO
FROM TB_EMPREGADO Empregado JOIN TB_DEPENDENTE Dependente ON
Empregado.ID_EMPREGADO = Dependente.ID_EMPREGADO
FOR XML AUTO, ROOT('Empregados'), ELEMENTS XSINIL
```

Resultado:

```
<Empregados xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Empregado>
    <ID_EMPREGADO>1</ID_EMPREGADO>
    <NOME>OLAVO </NOME>
    <DATA_ADMISSAO>2009-11-20T00:00:00</DATA_ADMISSAO>
    <SALARIO>3000.00</SALARIO>
    <ID_DEPENDENTE>2</ID_DEPENDENTE>
    <NOME>PEDRO</NOME>
    <DATA_NASCIMENTO xsi:nil="true" />
  </Empregado>
  <Empregado>
    <ID_EMPREGADO>2</ID_EMPREGADO>
    <NOME>JOSE </NOME>
    <DATA_ADMISSAO>2017-07-24T00:00:00</DATA_ADMISSAO>
    <SALARIO>660.00</SALARIO>
    <Dependente>
      <ID_DEPENDENTE>3</ID_DEPENDENTE>
      <NOME>CARLOS</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
    <Dependente>
      <ID_DEPENDENTE>4</ID_DEPENDENTE>
      <NOME>ALBERTO</NOME>
      <DATA_NASCIMENTO xsi:nil="true" />
    </Dependente>
  </Empregado>
</Empregados>
```

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

```
SELECT
  Cliente.ID_CLIENTE AS IdCliente, Cliente.NOME AS Cliente,
  Pedidos.id_PEDIDO AS IdPedido, Pedidos.VLR_TOTAL AS
  VlrPedido,
  Pedidos.DATA_EMISSAO AS Emissao
FROM TB_CLIENTE Cliente JOIN TB_PEDIDO Pedidos ON Cliente.ID_
CLIENTE = Pedidos.ID_CLIENTE
WHERE Pedidos.DATA_EMISSAO BETWEEN '2017.1.1' AND '2017.1.31'
FOR XML AUTO, ROOT('Clientes')
```

Resultado:

```
<Clientes>
  <Cliente IdCliente="4" Cliente="QCQQBNPUKNHDSFBMEKESJNMJMJDOP">
    <Pedidos IdPedido="207" VlrPedido="0.00" Emissao="2017-01-02T00:00:00" />
  </Cliente>
  <Cliente IdCliente="14" Cliente="BQBCCYFGMVGGIUOXJHNMTRTDGGAMQ">
    <Pedidos IdPedido="301" VlrPedido="1206.50" Emissao="2017-01-09T00:00:00" />
  </Cliente>
  <Cliente IdCliente="16" Cliente="DERFOLHPQCHDWOHTKFNTIRXTMIFSGO">
    <Pedidos IdPedido="190" VlrPedido="96.14" Emissao="2017-01-01T00:00:00" />
  </Cliente>
  <Cliente IdCliente="18" Cliente="GAQWBXKDVPRUQRDUENRQXKFBOATIGA">
    <Pedidos IdPedido="248" VlrPedido="15.39" Emissao="2017-01-06T00:00:00" />
    <Pedidos IdPedido="7645" VlrPedido="5362.38" Emissao="2017-01-15T00:00:00" />
  </Cliente>
  <Cliente IdCliente="22" Cliente="UABJBSQWCCTGVLGGREKEIXMGYJQL">
    <Pedidos IdPedido="7682" VlrPedido="3969.36" Emissao="2017-01-28T00:00:00" />
  </Cliente>
  <Cliente IdCliente="24" Cliente="NIKMNRCUHQYUIGHHFMRVTCNMVYNVNEE">
    <Pedidos IdPedido="245" VlrPedido="2473.80" Emissao="2017-01-06T00:00:00" />
  </Cliente>
  <Cliente IdCliente="26" Cliente="VTIVMWIDUJXSNNWJAVAIJMGLTQOHH">
    <Pedidos IdPedido="296" VlrPedido="610.85" Emissao="2017-01-09T00:00:00" />
  </Cliente>
  ... ..
```

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

```
SELECT
    Cliente.ID_CLIENTE AS IdCliente, Cliente.NOME AS Cliente,
    Pedidos.ID_PEDIDO AS IdPedido,
    Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS
    Emissao
FROM TB_CLIENTE Cliente JOIN TB_PEDIDO Pedidos ON Cliente.ID_
CLIENTE = Pedidos.ID_CLIENTE
WHERE Pedidos.DATA_EMISSAO BETWEEN '2017.1.1' AND '2017.1.31'
FOR XML AUTO, ROOT('Clientes'), ELEMENTS
```

Resultado:

```
<Clientes>
  <Cliente>
    <IdCliente>4</IdCliente>
    <Cliente>QCQBNPUKNHDSFBMEKESJNMJMJDQP</Cliente>
    <Pedidos>
      <IdPedido>207</IdPedido>
      <VlrPedido>0.00</VlrPedido>
      <Emissao>2017-01-02T00:00:00</Emissao>
    </Pedidos>
  </Cliente>
  <Cliente>
    <IdCliente>14</IdCliente>
    <Cliente>BPBQCCYFGMVGGIUOXJHNMTRTDGGAMQ</Cliente>
    <Pedidos>
      <IdPedido>301</IdPedido>
      <VlrPedido>1206.50</VlrPedido>
      <Emissao>2017-01-09T00:00:00</Emissao>
    </Pedidos>
  </Cliente>
  <Cliente>
    <IdCliente>16</IdCliente>
    <Cliente>DERFOLHPQCHDWOHTKFNTIRXTMIFSGO</Cliente>
    <Pedidos>
      <IdPedido>190</IdPedido>
      <VlrPedido>96.14</VlrPedido>
      <Emissao>2017-01-01T00:00:00</Emissao>
    </Pedidos>
  </Cliente>
</Clientes>
```

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

```
SELECT
  Cliente.ID_CLIENTE AS IdCliente, Cliente.NOME AS Cliente,
  Pedidos.ID_PEDIDO AS IdPedido,
  Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS
  Emissao,
  Itens.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.ID_CLIENTE = Pedidos.ID_
CLIENTE
JOIN TB_ITENSPEDIDO Itens ON Pedidos.ID_PEDIDO = Itens.ID_
PEDIDO
WHERE Pedidos.DATA_EMISSAO BETWEEN '2017.1.1' AND '2017.1.31'
-- Importante para ter um resultado correto
ORDER BY Cliente.NOME, Pedidos.ID_PEDIDO
FOR XML AUTO, ROOT('Clientes')
```

Resultado:

```
<Clientes>
<Cliente IdCliente="600" Cliente="AWEKQTBVTLQBVSTCXMEHCYPFONCMS">
  <Pedidos IdPedido="7651" VlrPedido="4439.12" Emissao="2017-01-17T00:00:00">
    <Itens IdItem="1" IdProduto="1" Quantidade="352" PrUnitario="0.0739" />
    <Itens IdItem="2" IdProduto="40" Quantidade="373" PrUnitario="4.1149" />
    <Itens IdItem="3" IdProduto="37" Quantidade="384" />
    <Itens IdItem="4" IdProduto="15" Quantidade="441" PrUnitario="3.0554" />
    <Itens IdItem="5" IdProduto="47" Quantidade="496" PrUnitario="3.7699" />
  </Pedidos>
</Cliente>
<Cliente IdCliente="390" Cliente="BLASTBVXSIFCJPEIIMRGHEEGLMSDS">
  <Pedidos IdPedido="321" VlrPedido="374.30" Emissao="2017-01-12T00:00:00">
    <Itens IdItem="1" IdProduto="10" Quantidade="200" PrUnitario="0.8800" />
    <Itens IdItem="2" IdProduto="56" Quantidade="200" PrUnitario="1.0900" />
  </Pedidos>
</Cliente>
<Cliente IdCliente="376" VlrPedido="798.00" Emissao="2017-01-15T00:00:00">
  <Itens IdItem="1" IdProduto="3" Quantidade="2000" PrUnitario="0.4200" />
</Pedidos>
</Cliente>
<Cliente IdCliente="468" Cliente="BLHVIFRNPOBPPTVEWLAQWBXNPTNIUK">
  <Pedidos IdPedido="7611" VlrPedido="4318.44" Emissao="2017-01-03T00:00:00">
    <Itens IdItem="1" IdProduto="60" Quantidade="170" PrUnitario="1.1088" />
    <Itens IdItem="2" IdProduto="5" Quantidade="204" PrUnitario="3.3510" />
    <Itens IdItem="3" IdProduto="51" Quantidade="135" PrUnitario="3.6714" />
    <Itens IdItem="4" IdProduto="22" Quantidade="362" PrUnitario="2.1437" />
    <Itens IdItem="5" IdProduto="58" Quantidade="152" PrUnitario="2.8336" />
    <Itens IdItem="6" IdProduto="29" Quantidade="164" PrUnitario="0.1478" />
    <Itens IdItem="7" IdProduto="47" Quantidade="502" PrUnitario="3.7699" />
  </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">
      ID_EMPREGADO      [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 ID_EMPREGADO
FOR XML EXPLICIT, ROOT('Empregados')
```

Resultado:

```
<Empregados>
  <Empregado Codigo="1">
    <Funcionario>OLAVO </Funcionario>
    <Renda>3000.00</Renda>
    <DataAdm>2009-11-20T00:00:00</DataAdm>
  </Empregado>
  <Empregado Codigo="2">
    <Funcionario>JOSE </Funcionario>
    <Renda>660.00</Renda>
    <DataAdm>2017-07-24T00:00:00</DataAdm>
  </Empregado>
  <Empregado Codigo="3">
    <Funcionario>MARCELO </Funcionario>
    <Renda>2880.00</Renda>
    <DataAdm>2013-06-15T00:00:00</DataAdm>
  </Empregado>
  <Empregado Codigo="4">
    <Funcionario>PAULO </Funcionario>
    <Renda>864.00</Renda>
    <DataAdm>2011-07-03T00:00:00</DataAdm>
  </Empregado>
```


- XML EXPLICIT

```
SELECT 1 AS Tag, NULL AS Parent,  
--      atributo ficará oculto  
ID_EMPREGADO [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 ID_EMPREGADO  
FOR XML EXPLICIT, ROOT('Empregados')
```

Resultado:

```
<Empregados>  
  <Empregado>  
    <Funcionario>OLAVO </Funcionario>  
    <Renda>3000.00</Renda>  
    <DataAdm>2009-11-20T00:00:00</DataAdm>  
  </Empregado>  
  <Empregado>  
    <Funcionario>JOSE </Funcionario>  
    <Renda>660.00</Renda>  
    <DataAdm>2017-07-24T00:00:00</DataAdm>  
  </Empregado>  
  <Empregado>  
    <Funcionario>MARCELO </Funcionario>  
    <Renda>2880.00</Renda>  
    <DataAdm>2013-06-15T00:00:00</DataAdm>  
  </Empregado>  
  <Empregado>  
    <Funcionario>PAULO </Funcionario>  
    <Renda>864.00</Renda>  
    <DataAdm>2011-07-03T00:00:00</DataAdm>  
  </Empregado>
```

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

```
SELECT 1 AS Tag, NULL AS Parent,
--      atributo ficará oculto
      ID_EMPREGADO      [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 ID_EMPREGADO
FOR XML EXPLICIT, ROOT('Empregados')
```

Resultado:

```
<Empregados>
  <Empregado>
    <Funcionario>OLAVO </Funcionario>
    <Renda>3000.00</Renda>
    <DataAdm>2009-11-20T00:00:00</DataAdm>
  </Empregado>
  <Empregado>
    <Funcionario>JOSE </Funcionario>
    <Renda>660.00</Renda>
    <DataAdm>2017-07-24T00:00:00</DataAdm>
  </Empregado>
  <Empregado>
    <Funcionario>MARCELO </Funcionario>
    <Renda>2880.00</Renda>
    <DataAdm>2013-06-15T00:00:00</DataAdm>
  </Empregado>
  <Empregado>
    <Funcionario>PAULO </Funcionario>
    <Renda>864.00</Renda>
    <DataAdm>2011-07-03T00:00:00</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,
      ID_EMPREGADO,
      NOME,
      SALARIO,
      DATA_ADMISSAO
FROM TB_EMPREGADO
FOR XML EXPLICIT , ROOT('Empregados')

```

Resultado:

```

<Empregados>
  <Empregados>
    <Empregado CODFUN="1">
      <Nome>OLAVO </Nome>
      <Salario>3000.00</Salario>
      <DataAdm>2009-11-20T00:00:00</DataAdm>
    </Empregado>
    <Empregado CODFUN="2">
      <Nome>JOSE </Nome>
      <Salario>660.00</Salario>
      <DataAdm>2017-07-24T00:00:00</DataAdm>
    </Empregado>
    <Empregado CODFUN="3">
      <Nome>MARCELO </Nome>
      <Salario>2880.00</Salario>
      <DataAdm>2013-06-15T00:00:00</DataAdm>
    </Empregado>
    <Empregado CODFUN="4">
      <Nome>PAULO </Nome>
      <Salario>864.00</Salario>
      <DataAdm>2011-07-03T00:00:00</DataAdm>
    </Empregado>
  </Empregados>
</Empregados>

```

- 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,
    ID_EMPREGADO,
    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.ID_EMPREGADO,
    E.NOME,
    E.SALARIO,
    E.DATA_ADMISSAO,
    D.ID_EMPREGADO, D.ID_DEPENDENTE, D.NOME, D.DATA_
NASCIMENTO
FROM TB_EMPREGADO E
JOIN TB_DEPENDENTE D ON E.ID_EMPREGADO = D.ID_EMPREGADO
ORDER BY [Empregado!2!CODFUN],[Dependente!3!CodFun!HIDE]
FOR XML EXPLICIT;
    
```

Resultado:

```
<Empregados>
  <Empregado CODFUN="1">
    <Nome>OLAVO </Nome>
    <Salario>3000.00</Salario>
    <DataAdm>2009-11-20T00:00:00</DataAdm>
    <Dependente CodDep="2">
      <Nome>PEDRO</Nome>
    </Dependente>
  </Empregado>
  <Empregado CODFUN="2">
    <Nome>JOSE </Nome>
    <Salario>660.00</Salario>
    <DataAdm>2017-07-24T00:00:00</DataAdm>
    <Dependente CodDep="3">
      <Nome>CARLOS</Nome>
    </Dependente>
    <Dependente CodDep="4">
      <Nome>ALBERTO</Nome>
    </Dependente>
    <Dependente CodDep="5">
      <Nome>MARTINS</Nome>
    </Dependente>
    <Dependente CodDep="6">
      <Nome>MANOEL</Nome>
    </Dependente>
    <Dependente CodDep="7">
      <Nome>PAULO</Nome>
    </Dependente>
    <Dependente CodDep="8">
      <Nome>MARIA</Nome>
    </Dependente>
  </Empregado>
  <Empregado CODFUN="3">
    <Nome>MARCELO </Nome>
    <Salario>2880.00</Salario>
    <DataAdm>2013-06-15T00:00:00</DataAdm>
    <Dependente CodDep="9">
      <Nome>LUIZA</Nome>
    </Dependente>
  </Empregado>
  <Empregado CODFUN="4">
```

- 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 '&apos;';
            ELSE @C
        END
        SET @CONT += 1;
    END
    RETURN @RET;
END
GO
```

- XML PATH

```
SELECT
  CAST('<Codigo>' + CAST(C.ID_CLIENTE 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>3</Codigo>
    <Nome>SQSKRGYHTBLNWIAREAKTKDQPWQFOLE</Nome>
  </Cliente>
  <Cliente>
    <Codigo>4</Codigo>
    <Nome>QCQQBNUKHNHDSFBMEKESJNMJMJQDDP</Nome>
  </Cliente>
  <Cliente>
    <Codigo>5</Codigo>
    <Nome>TTOMDCPSXCWRXSQWJNOVURRENQDFKL</Nome>
  </Cliente>
  <Cliente>
    <Codigo>6</Codigo>
    <Nome>ANXTUDDIVPWQUVINKNFBVLEOSUODXI</Nome>
  </Cliente>
  <Cliente>
    <Codigo>7</Codigo>
    <Nome>TCSGVJISYSISLFFELSMLLYSBMPRQNK</Nome>
  </Cliente>
  <Cliente>
    <Codigo>8</Codigo>
    <Nome>QXVTRVGAOBVBXCAAQIGMMHNFFIFVJH</Nome>
  </Cliente>
```

- XML PATH (Mestre detalhe com atributo no detalhe)

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

Resultado:

```
<Clientes>
  <Cliente>
    <Codigo>36</Codigo>
    <Nome>ADSMIVQOCMSKMNMOXXWAKXUGMAVW</Nome>
  </Cliente>
  <Cliente>
    <Codigo>246</Codigo>
    <Nome>AFGPOBICSMGOGIVPWQKTOQWMNLUPME</Nome>
  </Cliente>
  <Cliente>
    <Codigo>302</Codigo>
    <Nome>AGMRBF5MVXKFNEOUCHIHIWGXVRPPGM</Nome>
  </Cliente>
  <Cliente>
    <Codigo>493</Codigo>
    <Nome>AGUKSBQWFRJEJNFBIRCRUQYPIVTEET</Nome>
  </Cliente>
  <Cliente>
    <Codigo>151</Codigo>
    <Nome>AIBFYQAKWHLTXHIMNRDQPKUSLXICM</Nome>
  </Cliente>
  <Cliente>
    <Codigo>270</Codigo>
    <Nome>AIVNLERCVTPCUUKJDSEJSOSIVWVPXW</Nome>
  </Cliente>
```


- XML PATH (Mestre detalhe com elementos no detalhe)

```

SELECT
    CAST('<Codigo>' + CAST(C.ID_CLIENTE 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( ID_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 ID_CLIENTE = C.ID_CLIENTE AND DATA_EMISSAO BETWEEN
'2007.1.1' AND '2007.1.31'
        ORDER BY ID_PEDIDO
        FOR XML PATH('Pedido'), TYPE)
FROM TB_CLIENTE C
ORDER BY C.NOME
FOR XML PATH('Cliente'), ROOT('Clientes')

```

Resultado:

```

<Clientes>
<Cliente>
  <Codigo>36</Codigo>
  <Nome>ADSMIVQOCMSKMEMMOXXWAKXUGMAVW</Nome>
</Cliente>
<Cliente>
  <Codigo>246</Codigo>
  <Nome>AFGPOBICSMGOGIVPWOKTOQWMNLUPME</Nome>
</Cliente>
<Cliente>
  <Codigo>302</Codigo>
  <Nome>AGMRBFSMVXKFNEOUCHIHIWGXRPPGM</Nome>
</Cliente>
<Cliente>
  <Codigo>493</Codigo>
  <Nome>AGUKSBQWFRJEJNFBIRCRUQYPIVTEET</Nome>
</Cliente>
<Cliente>
  <Codigo>151</Codigo>
  <Nome>AIBFYQAKWHLTXHIMNRDQPKUSLXICM</Nome>
</Cliente>
<Cliente>
  <Codigo>270</Codigo>
  <Nome>AIVNLERCVTPCUKKJDEJSOSIVVWPXW</Nome>

```

10.4.2. Métodos XML

Vejam, nos subtópicos a seguir, os métodos XML.

10.4.2.1. 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      ID_EMPREGADO, NOME, DATA_ADMISSAO
    FROM TB_EMPREGADO AS EMPREGADO
    FOR XML AUTO, ELEMENTS

)

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

Resultado:

```
<EMPREGADO>
  <ID_EMPREGADO>1</ID_EMPREGADO>
  <NOME>OLAVO </NOME>
  <DATA_ADMISSAO>2009-11-20T00:00:00</DATA_ADMISSAO>
</EMPREGADO>
<EMPREGADO>
  <ID_EMPREGADO>2</ID_EMPREGADO>
  <NOME>JOSE </NOME>
  <DATA_ADMISSAO>2017-07-24T00:00:00</DATA_ADMISSAO>
</EMPREGADO>
<EMPREGADO>
  <ID_EMPREGADO>3</ID_EMPREGADO>
  <NOME>MARCELO </NOME>
  <DATA_ADMISSAO>2013-06-15T00:00:00</DATA_ADMISSAO>
</EMPREGADO>
<EMPREGADO>
  <ID_EMPREGADO>4</ID_EMPREGADO>
  <NOME>PAULO </NOME>
  <DATA_ADMISSAO>2011-07-03T00:00:00</DATA_ADMISSAO>
</EMPREGADO>
<EMPREGADO>
  <ID_EMPREGADO>5</ID_EMPREGADO>
  <NOME>JOAO </NOME>
  <DATA_ADMISSAO>2012-11-21T00:00:00</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:

```
-- Declarando uma variável XML
DECLARE @XML XML

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

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

Resultado:

```
<NOME>OLAVO </NOME>
<NOME>JOSE </NOME>
<NOME>MARCELO </NOME>
<NOME>PAULO </NOME>
<NOME>JOAO </NOME>
<NOME>CARLOS ALBERTO</NOME>
<NOME>ELIANE </NOME>
<NOME>RUDGE </NOME>
<NOME>MARIA APARECIDA</NOME>
<NOME>FERNANDO </NOME>
<NOME>JOAO </NOME>
<NOME>OSMAR </NOME>
<NOME>CASSIANO </NOME>
<NOME>MARCO </NOME>
<NOME>ALTAMIR </NOME>
<NOME>ANA MARIA</NOME>
<NOME>CARLOS FERNANDO</NOME>
<NOME>SEBASTIÃO </NOME>
<NOME>FURTO </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')
```

```
<NOME>FERNANDO </NOME>
```

```
<NOME>OLAVO </NOME>
```

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

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

```
<EMPREGADO>
  <ID_EMPREGADO>3</ID_EMPREGADO>
  <NOME>MARCELO</NOME>
  <DATA_ADMISSAO>2013-06-15T00:00:00</DATA_ADMISSAO>
</EMPREGADO>
```

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

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

```
<DATA_ADMISSAO>2013-06-15T00:00:00</DATA_ADMISSAO>
```

10.4.2.2. 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 ID_EMPREGADO, NOME, DATA_ADMISSAO FROM TB_
    EMPREGADO AS EMPREGADO
    FOR XML AUTO, ELEMENTS )
SELECT @XML.value('(EMPREGADO/NOME)[1]', 'varchar(100)')
```

	(No column name)
1	OLAVO

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

...

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

	(No column name)
1	ALTAMIR

10.4.2.3.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 ID_EMPREGADO, NOME, DATA_ADMISSAO FROM TB_
    EMPREGADO AS EMPREGADO
    FOR XML AUTO, ELEMENTS )

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

	(No column name)
1	1

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

...

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

	(No column name)
1	1

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

...

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

	(No column name)
1	0

Junto com o comando **CASE**:

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

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

(No column name)	
1	Não EXISTE

10.4.2.4. 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 ID_EMPREGADO, 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)
```

Nome	
1	<NOME>OLAVO</NOME>
2	<NOME>JOSE</NOME>
3	<NOME>MARCELO</NOME>
4	<NOME>PAULO</NOME>
5	<NOME>JOAO</NOME>
6	<NOME>CARLOS ALBERTO</NOME>
7	<NOME>ELIANE</NOME>
8	<NOME>RUDGE</NOME>
9	<NOME>MARIA APARECIDA</NOME>
10	<NOME>FERNANDO</NOME>
11	<NOME>JOAO</NOME>
12	<NOME>OSCAR</NOME>

10.4.3.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 db_ECOMMERCE.DBO.TB_TIPOPRODUTO AS TIPO
FOR XML AUTO, ROOT(''RESULTADO''), ELEMENTS " ' +
' QUERYOUT "C:\DADOS\ARQUIVOXML.XML" -SLOCALHOST -t -w -t -T'

EXEC MASTER..XP_CMDSHELL @CMD
```

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

	output
1	NULL
2	Starting copy...
3	NULL
4	1 rows copied.
5	Network packet size (bytes): 4096
6	Clock Time (ms.) Total : 94 Average : (1...
7	NULL

10.4.4. 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	0	ABRIDOR
3	0	PORTA LÁPIS
4	0	REGUA
5	0	ACES.CHAVEIRO
6	0	CANETA
7	0	CHAVEIRO
8	0	BOTTON
9	0	MISTURADOR DE DRINKS
10	0	PORTA MOEDAS
11	0	CARTÃO PVC
12	0	YO-YO
13	0	MAQUINAS

10.5.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.

10.5.1.FOR JSON

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

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

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

```
SELECT top 2 ID_PRODUTO, ID_TIPO, DESCRICAO, PRECO_VENDA
FROM TB_PRODUTO
FOR JSON AUTO
```

	JSON_F52E2B61-18A1-11d1-B105-00805F49916B
1	[{"ID_PRODUTO":1,"ID_TIPO":1,"DESCRICAO":"ABRIDO...

```
[{"ID_PRODUTO":1,"ID_TIPO":1,"DESCRICAO":"ABRIDOR SACA  
& ROLHA","PRECO_VENDA":0.0570}, {"ID_PRODUTO":2,"ID_  
TIPO":2,"DESCRICAO":"PORTA-LAPIS COM PEZINHO","PRECO_  
VENDA":1.8396}]
```

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

```
SELECT  
  Cliente.ID_CLIENTE AS IdCliente, Cliente.NOME AS Cliente,  
  Pedidos.ID_PEDIDO AS IdPedido,  
  Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS  
  Emissao,  
  Itens.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.ID_CLIENTE = Pedidos.ID_  
CLIENTE  
JOIN TB_ITENSPEDIDO Itens ON Pedidos.ID_PEDIDO = Itens.ID_  
PEDIDO  
WHERE Pedidos.DATA_EMISSAO BETWEEN '2017.1.1' AND '2017.1.31'  
-- Importante para ter um resultado correto  
ORDER BY Cliente.NOME, Pedidos.ID_PEDIDO  
FOR JSON AUTO, ROOT('Clientes')
```

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

```
SELECT  
  Cliente.ID_CLIENTE AS IdCliente, Cliente.NOME AS Cliente,  
  Pedidos.ID_PEDIDO AS IdPedido,  
  Pedidos.VLR_TOTAL AS VlrPedido, Pedidos.DATA_EMISSAO AS  
  Emissao  
FROM TB_CLIENTE Cliente JOIN TB_PEDIDO Pedidos ON Cliente.ID_  
CLIENTE = Pedidos.ID_CLIENTE  
WHERE Pedidos.DATA_EMISSAO BETWEEN '2017.1.1' AND '2017.1.31'  
FOR JSON PATH, ROOT('Clientes')
```

10.5.2. 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"]')
```

Results		Messages	
	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;
```

Results		Messages	
	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;
```

Results		Messages	
	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	

10.5.3.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	
		NOME	
1		IMPACTA Treinamento	

		(No column name)	
1		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	
FONE			
1	(11)3342-1235		

10.5.4.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
```

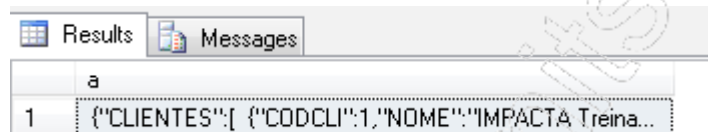
Results		Messages	
FONE			
1	["(11)3342-1234","(11)3342-1235"]		

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
```



Results	
	a
1	{\"CLIENTES\":[{\"CODCLI\":1,\"NOME\":\"IMPACTA Treina...

10.5.5. ISJSON

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

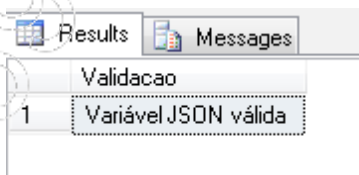
Neste exemplo, será validada uma variável do tipo JSON:

```
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 Validacao
```

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



Results	
	Validacao
1	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
	Validacao
1	Variável JSON inválida

10.5.6. 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_TIOPRODUTO**:

```
DECLARE @CMD VARCHAR(4000)

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

EXEC MASTER..XP_CMDSHELL @CMD
```

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



10.5.7. 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( ID_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( ID_TIPO INT, TIPO nvarchar(30)) AS RESULTADO
```

Messages

(16 rows affected)

Para validar, realize uma consulta na tabela:

```
SELECT * FROM TB_TIPO_JSON
```

COD_TIPO	TIPO
0	NÃO CADASTRADO
1	ABRIDOR
2	PORTA LÁPIS
3	REGUA
4	ACES.CHAVEIRO
5	CANETA
6	CHAVEIRO
7	BOTTON
8	MISTURADOR DE DRINKS
9	PORTA MOEDAS
10	CARTÃO 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 do capítulo.

- 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**.

Bruna C
391.642.208180



Acesso a recursos externos

Teste seus conhecimentos



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?

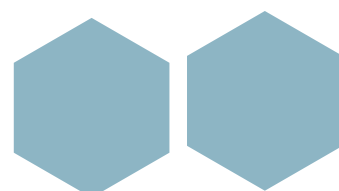
- ☐ a) CLOSEJSON
- ☐ b) OPENJSON
- ☐ c) JSON_VALUE
- ☐ d) JSON_QUERY
- ☐ e) ISJSON



Acesso a recursos externos



Mãos à obra!



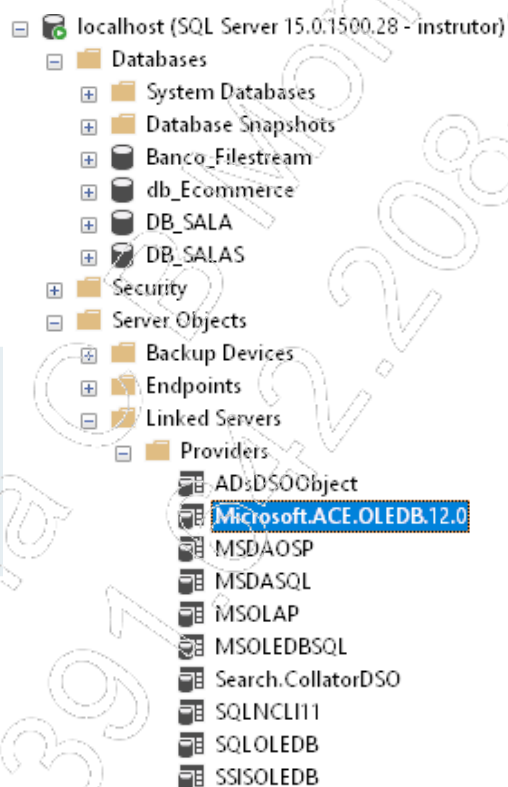
Laboratório 1

A – Configurando o SQL Server

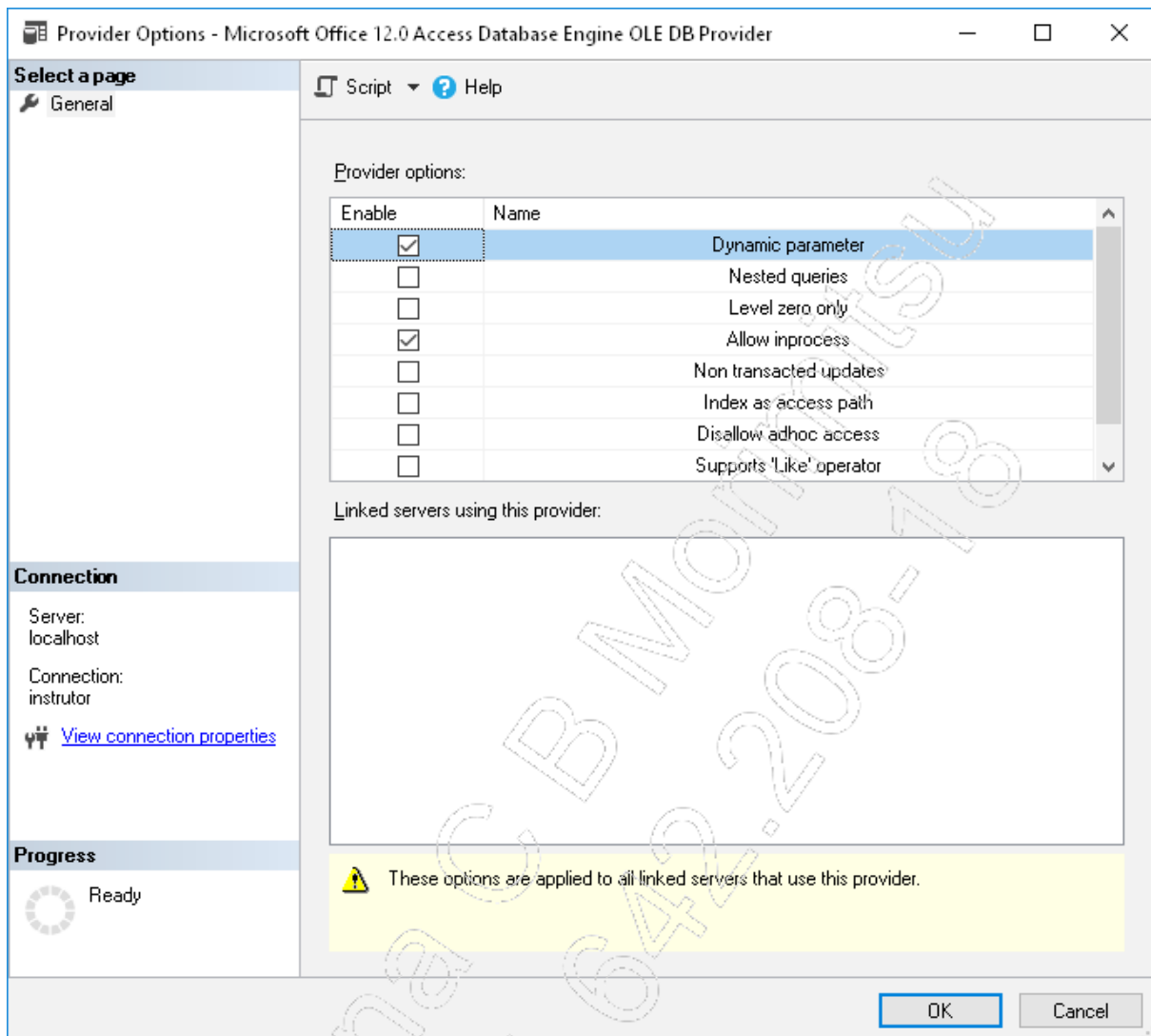
1. Execute os comandos adiante para habilitar opções avançadas e consultas distribuídas:

```
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 **Cap_10**;

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 **TB_PEDIDO** do banco **DB_ECOMMERCE**. Apresente as seguintes informações: **Num_pedido**, **Nome**, **VLR_TOTAL**, e **DATA_EMISSAO** 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 **DB_ECOMMERCE** 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 2018, 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="CELSON 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.