

Apostila de Linguagem SQL (2009)

Focada em MS SQL SERVER 2005

Prof. Ms. Eduardo Rosalém Marcelino
ermarc@itelefonica.com.br

Bibliografias utilizadas:

ELMASRI, Ramez; NAVATHE, S.B., **Sistemas de Bancos de Dados**, 4º ed. São Paulo: Pearson, 2005.

DATE, C.J. **Introdução a Sistemas de Banco de Dados**. 7 ed. São Paulo: Campus, 2000.

MACHADO, Felipe N.R. **Projeto de Banco de Dados – Uma visão prática**. 9 ed. São Paulo: Érica, 2000.

BATTISTI, Julio. **SQL SERVER 2005 ADMINISTRAÇÃO & DESENVOLVIMENTO: CURSO COMPLETO**. São Paulo: Axcel Books, 2005

MSDN, <http://msdn.microsoft.com/pt-br/default.aspx>

BORGES, N.A., **Curso Microsoft SQL SERVER 2000 – Utilizando a linguagem Transact-SQL**. Centro Universitário Fundação Santo André.

Índice:

História e características da linguagem SQL	4
Tabelas utilizadas nos exemplos desta apostila:	5
CREATE DATABASE	6
Tipos de Dados	7
Create Table	8
Instrução INSERT	8
Instrução DELETE	8
Instrução UPDATE	8
Instrução SELECT	9
ALIAS (Apelido)	10
Cláusula Where (Restrições)	11
BETWEEN (Condição com uma faixa de valores)	11
LIKE (Comparação com uma parte de uma literal)	12
IN (Condição com valores fixos)	13
DISTINCT (Valores únicos)	13
TOP (limitando a quantidade de registros)	14
Cláusula ORDER BY	15
FUNÇÕES DE AGRUPAMENTO	16
COUNT (Contador de registros)	16
SUM (Somatória de valores)	17
AVG (Média de valores)	17
MIN, MAX (Menor e maior valor)	18
Cláusula Group By	19
Cláusula Having	20
Exercícios	21
JOINS (Junções)	22
INNER JOIN (junção interna)	22
OUTER JOIN (junção externa)	23
LEFT JOIN (junção externa à esquerda)	23
RIGHT JOIN (junção externa à direita)	24
FULL OUTER JOIN (junção externa total)	25
SELF JOIN (Auto Junção)	26
Cláusula Union	27
CAST e CONVERT	29
Subconsultas (Subqueries ou Consultas Encadeadas)	30
EX. 1: Utilizando um subconsulta para retornar um campo adicional:	30
EX. 2: Utilizando um subconsulta para restringir os registros retornados	31
EX. 3: Aplicação da Cláusula TOP e Operadores In e Not In em subconsultas	32
Visões (Views)	33
Principais estruturas do Transact-SQL	36
BEGIN...END (Transact-SQL)	36
IF...ELSE (Transact-SQL)	36
WHILE (Transact-SQL)	36
CASE (Transact-SQL)	37
Stored Procedures (Procedimentos Armazenados)	39
Parâmetros de Entrada e Saída	43
SP´s que acessam dados do banco	43
Tabelas temporárias	44
Triggers (Gatilhos)	46
TRIGGER INSERT	48
TRIGGER DELETE	48
TRIGGER UPDATE	49
Habilitando e Desabilitando Triggers	50
Cursors (Transact-SQL)	51
FETCH (Transact-SQL)	53
@@FETCH_STATUS (Transact-SQL)	53
Exemplos	54
Funções definidas pelo usuário (FUNCTIONS)	57
Benefícios da função definida pelo usuário	57
Componentes de uma função definida pelo usuário	57

	3
Exemplo de valor escalar:	58
Exemplo de valor de tabela:	58
Controle de Transação.....	60
Controle de exceção (TRY...CATCH)	61
Recuperando informações de erro.....	63
Usando TRY...CATCH em uma transação.....	64

História e características da linguagem SQL

Fonte: <http://sweet.ua.pt/~a35438/Outros/SQL.htm>

A história do SQL começa em 1970 com a publicação por E. F. Codd, no ACM Journal, de um artigo intitulado "A Relational Model of Data for Large Shared Data Banks". O modelo proposto por Codd é hoje considerado a base de trabalho para qualquer Sistema Gerenciador de Base de Dados Relacional (SGBDR).

A primeira implementação da linguagem SEQUEL foi realizada pela IBM e tinha por objectivo a implementação do modelo de Codd. A evolução desta linguagem veio a dar origem ao SQL. A primeira implementação comercial de SQL foi realizada pela Relational Software, Inc., hoje conhecida por Oracle Corporation.

Nos dias de hoje, a linguagem SQL é considerada um standard dos Sistemas de Gestão de Base de Dados Relacionais (SGBDR); por isso, todos os fabricantes a integram nos seus produtos.

A linguagem SQL pertence à 4ª Geração das Linguagens de Programação. Não é, no entanto, uma evolução das linguagens de 3ª Geração (Pascal, BASIC, C, COBOL, FORTRAN), já que estas têm características bem diferentes:

- Existência de Variáveis, vectores, ...;
- Existência de instruções condicionais (if, switch, case, ...);
- Existência de ciclos (for, while, do...while, repeat...until);
- Possibilidade de escrita de funções e procedimentos;

Nenhuma destas características existe no SQL, havendo maior ligação entre a 3ª e 5ª gerações de linguagens de programação do que com a 4ª. A linguagem SQL destina-se, por isso e pela sua simplicidade, não só a informáticos, como também a gestores, utilizadores, administradores de bases de dados, etc..

No entanto, a sua principal diferença em relação às linguagens de 3ª geração é a ausência nestas de um objetivo pré-definido, coisa que no SQL está bem determinado: proporcionar a interface entre o SGBDR e o usuário, através da manipulação de dados.

A linguagem SQL implementa os conceitos definidos no modelo relacional, reduzindo assim as incompatibilidades entre os sistemas e evitando a opção por arquitecturas proprietárias que implicam maiores custos de desenvolvimento e maior esforço financeiro e humano por parte dos desenvolvedores.

Com a linguagem SQL é possível:

- Criar, alterar e remover todos os componentes de uma base de dados, como tabelas, índices, views, etc.;
- Inserir, alterar e apagar dados;
- Interrogar a base de dados;
- Controlar o acesso dos usuários à base de dados, e às operações a que cada um deles tem acesso;
- Obter a garantia da consistência e integridade dos dados;

A linguagem SQL é composta por vários conjuntos de comandos, como por exemplo:

DDL (Data Definition Language): comandos para definir ou modificar a composição das tabelas, apagar tabelas, criar índices, definir "views", especificar direitos de acesso a tabelas e views;

DML (interactive Data Manipulation Language): inclui uma linguagem de consulta baseada em álgebra relacional e em cálculo relacional sobre registos; inclui também comandos para inserir, apagar e modificar registos na base de dados;

A linguagem de SQL inclui ainda a capacidade de verificação de integridade dos dados, bem como comandos para especificação do princípio e fim de transacções; algumas implementações permitem o impedimento explícito de acesso aos dados, para controle de acesso concorrenciais.

Tabelas utilizadas nos exemplos desta apostila:

<pre>select * from funcionarios</pre>							<pre>Select * From setores</pre>		
	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc		setor_id	setor_nome
1	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000	1	1	Faturamento
2	2	Ana	NULL	1	4780.00	1971-07-05 00:00:00.000	2	2	Contabilidade
3	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000	3	3	RH
4	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	4	4	Compras
5	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000	5	5	Estoque
6	6	Eduardo	1	2	300.00	1991-08-05 00:00:00.000	6	6	Farmácia
7	7	Anderson	1	2	500.00	1977-07-09 00:00:00.000	7	7	Recepção
8	8	Fábio	1	2	350.00	1989-05-06 00:00:00.000	8	8	Desenvolvimento
9	9	João	1	2	980.00	1981-02-03 00:00:00.000	9	9	Engenharia
10	10	José	2	2	321.00	1981-01-01 00:00:00.000	10	10	CAD
11	11	Ingrid	2	3	422.00	1982-02-02 00:00:00.000			
12	12	Bruno	2	3	890.00	1963-05-25 00:00:00.000			
13	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000			
14	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000			
15	15	Daniel	3	NULL	600.00	1984-12-06 00:00:00.000			
16	16	Valdir	3	NULL	900.00	1984-07-09 00:00:00.000			
17	17	Kleber	3	NULL	800.63	1957-02-02 00:00:00.000			

CREATE DATABASE

Cria um novo banco de dados e os arquivos usados para armazená-lo. Cada banco de dados tem um proprietário que pode executar atividades especiais no banco de dados. O proprietário é o usuário que cria o banco de dados. O proprietário do banco de dados pode ser alterado usando `sp_changedbowner`.

Cada banco de dados tem no mínimo dois arquivos, um arquivo primário e um arquivo de log de transações.

Ex 1:

Criando um banco de dados sem especificar os arquivos:

Obs: neste caso, o banco de dados será salvo na pasta padrão do SQL Server.

```
create database DBLocadora -- cria o banco de dados
```

Ex 2: Verificando a existência do banco antes de criá-lo. Caso ele exista, o banco será apagado.

```
use master -- vai para o banco de dados principal do sql server (master)

IF DB_ID ('DBLocadora') IS NOT NULL -- verifica se o banco de dados já existe
    DROP DATABASE DBLocadora -- se já existir, apaga

create database DBLocadora -- cria o banco de dados
```

Ex 3: Complementando o exemplo anterior, agora vamos especificar onde o arquivo de dados e o arquivo de Log serão salvos.

```
use master -- vai para o banco de dados principal do sql server

IF DB_ID ('DBLocadora') IS NOT NULL -- verifica se o banco de dados já existe
    DROP DATABASE DBLocadora -- se já existir, apaga

create database DBLocadora -- cria o banco de dados
ON ( NAME = dbLocadora_dados, FILENAME = 'd:\BD\dbLocadora_dados.mdf' )
LOG ON ( NAME = dbLocadora_Log, FILENAME = 'd:\BD\dbLocadora_Log.ldf' )
```

Tipos de Dados

Os principais tipos de dados do SQL Server são:

	Tipo	Capacidade	Observações
Inteiro	Bigint	De -2^{63} (-9,223,372,036,854,775,808) até $2^{63}-1$ (9,223,372,036,854,775,807)	
	Int	De -2^{31} (-2,147,483,648) até $2^{31} - 1$ (2,147,483,647).	
	Smallint	De 0 até 255	
Bit	Bit	0 ou 1, normalmente utilizado como um valor lógico (Boolean)	
Real	Decimal	De $-10^{38} + 1$ até $10^{38} - 1$	
	Float	De $-1.79E + 308$ até $-2.23E - 308$, 0 e de $2.23E + 308$ até $1.79E + 308$	
	real	De $-3.40E + 38$ até $-1.18E - 38$, 0 e de $1.18E - 38$ até $3.40E + 38$	
Data e hora	Datetime	De 1 de Janeiro de 1753 até 31 de dezembro de 9999	Precisão de 3.33 milsegundos
	Smalldatetime	Data e hora de 1 de Janeiro de 1900 até 6 de Junho de 2079	Precisão de um minuto.
String	Char	Até 8 mil caracteres	Tamanho fixo
	Varchar	Até 8 mil caracteres	Tamanho variável
	Text	Até $2^{31} - 1$ (2,147,483,647) caracteres	Tamanho variável
	Nchar	Até 4 mil caracteres	Tamanho fixo, unicode
	Nvarchar	Até 4 mil caracteres	Tamanho variável, unicode

Outros tipos suportados são: Money, Cursor, Variant (normalmente utilizado na Automação OLE), Table, capaz de armazenar um conjunto de resultados, rowversion (timestamp) e GUID (uniqueidentifier). O tipo sysname é utilizado para armazenar nomes de objetos, como tabelas, colunas etc.

Create Table

Permite criar uma nova tabela no banco de dados. Este comando possui diversas opções, portanto, será apresentada a sua versão mais simples. Para maiores detalhes, consulte o MSDN.

Ex 1: Cria uma tabela chamada fitas no banco de dados dbLocadora, definindo inclusive a chave primária.

```
use dbLocadora
GO

create table fitas
(
    codigo          int          primary key,
    descricao       varchar(20),
    data_cadastro   datetime,
    valor           decimal(10,2) NOT NULL
)
```

Ex 2: Criando uma tabela locacao_item com uma chave primária formada por 2 campos.

```
create table locacao_item
(
    locacao_id int NOT NULL,
    fita_cod   int NOT NULL,
    valor      decimal (10, 2) NULL default 0,

    CONSTRAINT PK_locacao_item PRIMARY KEY ( locacao_id, fita_cod )
)
```

Instrução INSERT

Insere um registro em uma tabela.

Sintaxe:

```
insert into TABELA (campo1, campo2, campoN) values (valor1, valor2, valorN)
```

Instrução DELETE

Apaga ou mais registros em uma tabela.

Sintaxe:

```
Delete from tabela where condição
```

Obs: a cláusula **where** é opcional, porém se não for utilizado, TODOS OS REGISTROS DA TABELA SERÃO APAGADOS!!!!

Instrução UPDATE

Altera um ou mais registros em uma tabela.

Sintaxe:

```
Update TABELA
Set      campo1 = valor1,
        campo2 = valor2,
        campoN = valorN
WHERE condição
```

Obs: a cláusula **where** é opcional, porém se não for utilizado, TODOS OS REGISTROS DA TABELA SERÃO ALTERADOS!!!!

Instrução SELECT

Instrui o programa principal do banco de dados para retornar a informação como um conjunto de registros.

Sintaxe

```
SELECT [predicado { * | tabela.* | [tabela.]campo1 [AS alias1] [, [tabela.]campo2 [AS alias2] [, ...]]}
FROM expressão tabela
WHERE condição
GROUP BY campo(s)
HAVING condição
ORDER BY campo(s)
```

A parte em azul é opcional.

A instrução SELECT em detalhes:

Parte	Descrição
predicado	Um dos seguintes predicados: DISTINCT ou TOP. Você usa o predicado para restringir o número de registros que retornam.
*	Especifica que todos os campos da tabela ou tabelas especificadas serão selecionados.
tabela	O nome da tabela que contém os campos dos quais os registros são selecionados.
campo1, campo2	Os nomes dos campos dos quais os dados serão recuperados. Se você incluir mais de um campo, eles serão recuperados na ordem listada.
alias1, alias2	Os nomes que serão usados como títulos de colunas em vez dos nomes originais das colunas na tabela.
expressão tabela	Tabela(s), subselect(s), etc.. contendo os dados que você deseja recuperar.

Para executar esta operação, o programa principal de banco de dados procura a tabela ou tabelas especificadas, extrai as colunas escolhidas, seleciona as linhas que satisfazem o critério e classifica ou agrupa as linhas resultantes na ordem especificada.

A instrução SELECT **não muda** os dados no banco de dados.

SELECT é normalmente a primeira palavra em uma instrução SQL. A maior parte das instruções SQL são instruções SELECT.

A sintaxe mínima da instrução SELECT é:

```
Select Campo(s) From Tabela
```

Você pode usar um asterisco (*) para selecionar todos os campos na tabela. O exemplo abaixo seleciona todos os campos na tabela Funcionarios:

```
SELECT * FROM Funcionarios
```

Se o nome de um campo estiver incluído em mais de uma tabela utilizada na instrução SQL, preceda-o com o nome da tabela e o operador . (ponto). Veja o exemplo abaixo: O campo setor_id existe nas duas tabelas utilizadas na instrução SQL, portanto devemos indicar também a tabela, precedendo o nome do campo.

```
Select funcionarios.func_nome, setores.setor_nome
From funcionarios
Inner join setores on setores.setor_id = funcionarios.setor_id
```

ALIAS (Apelido)

Literal que identifica um campo, uma função ou uma tabela

Sintaxe:

```
SELECT (<campo> ou <função> ou <literal>) <alias do retorno>
FROM <tabela> <alias da tabela>
```

Exemplo:

```
Select func_id as "Código do Funcionário"
From funcionarios
```

Observe que a coluna func_id passou a chamar [Código do Funcionário].

Observe o exemplo abaixo:

```
Select func_salario * 1.10
From funcionarios
where func_nome like '%b%'
```

	(No column name)
1	385.0000
2	979.0000
3	1123.1000
4	880.6930

Observe no resultado da instrução acima que a coluna resultante da operação aritmética está aparecendo como "No column name".

Para resolver este problema, devemos criar uma alias para o campo resultado, como na instrução abaixo:

```
Select func_salario * 1.10 as "Novo Salário"
From funcionarios
where func_nome like '%b%'
```

	Novo Salário
1	385.0000
2	979.0000
3	1123.1000
4	880.6930

A palavra "as" é opcional. As aspas (") só são necessárias caso a palavra seja composta, ou com caracteres especiais. Também pode ser utilizado [] ou apóstrofo (').

Cláusula Where (Restrições)

Com esta cláusula é possível fazermos restrição dos registros a serem manipulados, como veremos posteriormente ela também poderá ser usados para outros comandos do SQL.

Sintaxe:

```
SELECT <campo>
FROM <tabela>
WHERE <condição>
```

Exemplo:

```
Select func_salario * 1.10 as "Novo Salário"
From funcionarios
where func_salario >= 5000
```

	Novo Salário
1	5500.0000
2	7700.0000

Principais operadores utilizados

Operador	Significado
=	Igual
>	Maior
<	Menor
>=	Maior ou igual
<=	Meno ou igual
<> ou !=	Diferente
Is null	Igual a nulo
Is not null	Diferente de nulo
Not	Operador de negação.

Além dos operadores acima, existem também alguns operadores especiais, explicados a seguir.

BETWEEN (Condição com uma faixa de valores)

O operador BETWEEN determina a faixa de valores aceito de uma pesquisa. Este comando substitui os operadores '>=' e '<='.

Sintaxe:

```
SELECT <campo> FROM <tabela>
WHERE <campo> BETWEEN <valor_inicial> AND <valor_final>
```

Exemplo:

```
Select func_salario * 1.10 as "Novo Salário"
From funcionarios
where func_salario between 330 and 550
```

	Novo Salário
1	550.0000
2	385.0000
3	464.2000

LIKE (Comparação com uma parte de uma literal)

O operador LIKE faz a comparação somente com uma parte de uma literal, desconsidera tudo que possa vir antes ou depois do valor passado. Os caracteres '%' e '_' significam qual a parte a ser desconsiderada.

Sintaxe:

```
SELECT <campo> FROM <tabela>
WHERE <campo> LIKE <[%]parte_literal[%]>
```

Principais curingas para o operador Like:

Curinga	Descrição	Exemplo
%	Qualquer cadeia de zero ou mais caracteres.	WHERE title LIKE '%computer%' localiza todos os títulos de livro com a palavra 'computer' em qualquer lugar no título do livro.
_ (sublinhado)	Qualquer caractere único.	WHERE au_fname LIKE '_ean' localiza todos os nomes de quatro letras que terminam com ean (Dean, Sean e assim por diante).

Exemplos:

Retorna todos os funcionários que iniciam com a letra 'a'

```
Select func_nome
From funcionarios
where func_nome like 'a%'
```

	func_nome
1	Ana
2	Antonio
3	Anderson

Retorna todos os funcionários que terminam com a letra 'a'

```
Select func_nome
From funcionarios
where func_nome like '%a'
```

	func_nome
1	Maria
2	Ana
3	Carla
4	Bruna
5	Daniela

Retorna todos os funcionários que contém a letra 'b' em qualquer parte do nome

```
Select func_nome
From funcionarios
where func_nome like '%b%'
```

	func_nome
1	Fábio
2	Bruno
3	Bruna
4	Kleber

Retorna todos os registros que possuam um nome que inicie com "Brun", podendo variar apenas a última letra.

```
Select func_nome From funcionarios
where func_nome like 'Brun_'
```

	func_nome
1	Bruno
2	Bruna

IN (Condição com valores fixos)

O operador IN determina os valores fixos aceitos de uma pesquisa.

Sintaxe:

```
SELECT <campo> FROM <tabela>
WHERE <campo> IN (<lista_de_valores>)
```

Exemplo:

```
select *
from funcionarios
where func_id in (1,5,9)
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc
1	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000
2	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000
3	9	João	1	2	980.00	1981-02-03 00:00:00.000

A instrução acima poderia ser feita utilizando-se o operador OR. Veja o código a seguir:

```
select *
from funcionarios
where func_id = 1 or func_id = 5 or func_id =9
```

O operador NOT antecedendo o operador IN traz todos os registros que não estejam no conjunto especificado.

DISTINCT (Valores únicos)

O operador DISTINCT elimina todas as replicações, significando que somente dados distintos serão apresentados como resultado de uma pesquisa.

Sintaxe:

```
SELECT DISTINCT <campo>
FROM <tabela>
```

Exemplo:

A instrução abaixo irá retornar apenas os setores da tabela funcionários, porém apenas uma ocorrência de cada. Observe o valor NULL. O operador distinct também o considera um valor único.

```
select distinct setor_id
from funcionarios
```

	setor_id
1	NULL
2	1
3	2
4	3

TOP (limitando a quantidade de registros)

Indica que somente um primeiro conjunto ou porcentagem de linhas especificado será retornado de um conjunto de resultados de consulta.

Sintaxe:

```
Select TOP ( expression ) [ PERCENT ] * from Tabela
```

Onde *expression* pode ser um número ou uma porcentagem de linhas.

Se a consulta incluir uma cláusula ORDER BY, serão retornadas as primeiras linhas *expression*, ou porcentagem de linhas *expression*, solicitadas pela cláusula ORDER BY.

Exemplos:

A instrução abaixo irá retornar apenas os 5 primeiros registros, obedecendo a ordenação solicitada.

```
select top 5 *
from funcionarios
order by func_salario desc
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc
1	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000
2	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000
3	2	Ana	NULL	1	4780.00	1971-07-05 00:00:00.000
4	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000
5	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000

A instrução abaixo irá retornar apenas os 10% primeiros registros, obedecendo a ordenação solicitada.

```
select top 10 percent *
from funcionarios
order by func_salario desc
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc
1	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000
2	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000

Cláusula ORDER BY

ORDER BY é opcional. Entretanto, se você quiser exibir seus dados na ordem classificada, você deve utilizar ORDER BY. O padrão ordem de classificação é ascendente (A a Z, 0 a 9).

Exemplo:

```
select * from setores
order by setor_nome

--ou

select * from setores
order by setor_nome asc
```

	setor_id	setor_nome
1	10	CAD
2	4	Compras
3	2	Contabilidade
4	8	Desenvolvimento
5	9	Engenharia
6	5	Estoque
7	6	Farmácia
8	1	Faturamento
9	7	Recepção
10	3	RH

Para classificar em ordem descendente (Z a A, 9 a 0), adicione a palavra reservada DESC ao final de cada campo que você quiser classificar em ordem descendente.

Exemplo:

```
select * from setores
order by setor_nome desc
```

	setor_id	setor_nome
1	3	RH
2	7	Recepção
3	1	Faturamento
4	6	Farmácia
5	5	Estoque
6	9	Engenharia
7	8	Desenvolvimento
8	2	Contabilidade
9	4	Compras
10	10	CAD

ORDER BY é normalmente o último item em uma instrução SQL.

Você pode incluir campos adicionais na cláusula ORDER BY. Os registros são classificados primeiro pelo primeiro campo listado depois de ORDER BY. Os registros que tiverem valores iguais naquele campo são classificados pelo valor no segundo campo listado e assim por diante.

Exemplo:

```
select setor_id, func_nome from funcionarios
order by setor_id, func_nome
```

	setor_id	func_nome	func_id
1	NULL	Daniel	15
2	NULL	Kleber	17
3	NULL	Valdir	16
4	1	Ana	2
5	1	Antonio	4
6	1	Carla	3
7	1	Maria	1
8	1	Nestor	5
9	2	Anderson	7
10	2	Eduardo	6
11	2	Fábio	8
12	2	João	9
13	2	José	10
14	3	Bruna	13
15	3	Bruno	12
16	3	Daniela	14
17	3	Ingrid	11

FUNÇÕES DE AGRUPAMENTO

- COUNT (CONTAR)
- SUM (SOMAR)
- AVG (MÉDIA)
- MAX (MÁXIMO)
- MIN (MÍNIMO)

COUNT (Contador de registros)

A função COUNT retorna a quantidade de registros correspondentes a uma pesquisa.

Sintaxe:

```
SELECT COUNT([distinct] campo ou * ) FROM <tabela>
```

Exemplos:

Retornar a quantidade de registros na tabela funcionarios:

```
select count(*)
from funcionarios
```

	(No column name)
1	17

Retornar a quantidade de setores informados na tabela funcionarios:

```
select count(setor_id)
from funcionarios
```

	(No column name)
1	14

Retornar a quantidade de setores informados na tabela funcionarios, porém sem contar as repetições.

```
select count( distinct setor_id )
from funcionarios
```

	(No column name)
1	3

SUM (Somatória de valores)

A função SUM faz a somatória dos valores de um campo.

Sintaxe:

```
SELECT SUM(<campo>) FROM <tabela>
```

Exemplo:

Soma de todos os funcionários cadastrados

```
select sum( func_salario ) as "Soma dos Salários"
from funcionarios
```

	Soma dos Salários
1	27614.63

AVG (Média de valores)

A função AVG faz a média dos valores de um campo.

Sintaxe:

```
SELECT AVG(<campo>) FROM <tabela>
```

Exemplo:

Média salarial dos funcionários cadastrados:

```
select avg( func_salario ) as "Média salarial"
from funcionarios
```

	Média salarial
1	1624.390000

O resultado acima também pode ser conseguido com a instrução abaixo:

```
select sum( func_salario ) / count(*) as "Média salarial"
from funcionarios
```

MIN, MAX (Menor e maior valor)

As funções MIN e MAX retornam respectivamente o menor e o maior valor encontrado para um campo.

Sintaxe:

```
SELECT MIN(<campo>) FROM <tabela>
```

```
SELECT MAX(<campo>) FROM <tabela>
```

Exemplo

Retornando o menor e o maior salário dos funcionários cadastrados

```
select min( func_salario ) as "Menor salário" ,  
       max(func_salario ) as "Maior Salário"  
from funcionarios
```

	Menor salário	Maior Salário
1	300.00	7000.00

Cláusula Group By

Esta cláusula agrupa um conjunto de linhas selecionadas em um conjunto de linhas de resumo pelos valores de uma ou mais colunas ou expressões. Uma linha é retornada para cada grupo. Funções de agregação na lista de <seleção> da cláusula SELECT fornecem informações sobre cada grupo em vez de linhas individuais.

Expressões na cláusula GROUP BY podem conter colunas de tabelas, exibições ou tabelas derivadas na cláusula FROM. Não é exigido que as colunas apareçam na lista de <seleção> da cláusula SELECT.

Cada coluna de tabela ou exibição em qualquer expressão de não agregação na lista de <seleção> deve estar incluída na lista GROUP BY:

As seguintes instruções são permitidas:

- SELECT ColumnA, ColumnB FROM T GROUP BY ColumnA, ColumnB
- SELECT ColumnA + ColumnB FROM T GROUP BY ColumnA, ColumnB
- SELECT ColumnA + ColumnB FROM T GROUP BY ColumnA + ColumnB
- SELECT ColumnA + ColumnB + constant FROM T GROUP BY ColumnA, ColumnB

As seguintes instruções **NÃO** são permitidas:

- SELECT ColumnA, ColumnB FROM T GROUP BY ColumnA + ColumnB
- SELECT ColumnA + constant + ColumnB FROM T GROUP BY ColumnA + ColumnB.

Detalhes sobre Group By:

- Se funções de agregação forem incluídas na <lista de seleção> da cláusula SELECT, GROUP BY calculará um valor resumido para cada grupo. São conhecidas como agregações de vetor.
- Linhas que não atendem às condições na cláusula WHERE são removidas antes que qualquer operação de agrupamento seja executada.
- A cláusula HAVING é usada com a cláusula GROUP BY para filtrar grupos no conjunto de resultados.
- A cláusula GROUP BY não ordena o conjunto de resultados. Use a cláusula ORDER BY para ordenar o conjunto de resultados.
- Se uma coluna de agrupamento contiver valores nulos, todos os valores nulos serão considerados iguais e colocados em um único grupo.
- Não é possível usar GROUP BY com um alias para substituir um nome de coluna na cláusula AS, a menos que o alias substitua um nome de coluna em uma tabela derivada na cláusula FROM.
- GROUP BY ou HAVING não podem ser usados diretamente em colunas de **ntext**, **text** ou **image**. Essas colunas podem ser usadas como argumentos em funções que retornam um valor de outro tipo de dados, como SUBSTRING() e CAST().
- Não utilize SELECT *. Especifique os campos já que depois você deverá especificá-los na cláusula Group by.

Exemplo:

Selecionar a quantidade de funcionários por setor

```
select setor_id, count(*) as "Total de funcionários no setor"
from funcionarios
where setor_id is not null
group by setor_id
```

	setor_id	Total de funcionários no setor
1	1	5
2	2	5
3	3	4

Cláusula Having

A cláusula HAVING define condições na cláusula GROUP BY semelhante ao modo que WHERE interage com SELECT. O critério de pesquisa WHERE é aplicado antes da execução da operação de agrupamento; o critério de pesquisa HAVING é aplicado depois que a operação de agrupamento é executada. A sintaxe HAVING é semelhante à sintaxe WHERE, exceto se HAVING contiver funções de agregação. Cláusulas HAVING podem consultar quaisquer dos itens que aparecem na lista de seleção.

Compreendendo a sequência correta onde as cláusulas WHERE, GROUP BY e HAVING são aplicadas, ajuda a codificar consultas eficientes:

- A cláusula WHERE é usada para filtrar as linhas que resultam das operações especificadas na cláusula FROM.
- A cláusula GROUP BY é usada para agrupar o resultado da cláusula WHERE.
- A cláusula HAVING é usada para filtrar linhas do resultado agrupado.

Para qualquer critério de pesquisa que poderia ser aplicado antes ou depois da operação de agrupamento, é mais eficiente especificá-los na cláusula WHERE. Isto reduz o número de linhas que precisam ser agrupadas. Os únicos critérios de pesquisa que deveriam ser especificados na cláusula HAVING são os critérios de pesquisa que devem ser aplicados depois da operação de agrupamento ser executada. Quando possível, coloque todos os critérios de pesquisa na cláusula WHERE em vez de na cláusula HAVING.

Exemplo:

```
select setor_id, count(*) as "Total de funcionários no setor"
from funcionarios
where setor_id is not null
group by setor_id
having count(*) > 4
```

	setor_id	Total de funcionários no setor
1	1	5
2	2	5

Exercícios

- 1-) Selecione o maior e o menor código de funcionário.
- 2-) Selecione a média salarial
- 3-) Selecione todos os campos apenas dos 5 primeiros funcionários ordenados por ordem decrescente de salário
- 4-) Selecione todos os setores que terminam com a letra 'a'.
- 5-) Selecione o setor_id da tabela funcionários, de forma distinta .
- 6-) Selecione a média salarial dos funcionários que são gerentes.
- 7-) Selecione o código do gerente e a quantidade de funcionários que trabalham para ele.
- 8-) Selecione todos os funcionários que nasceram antes de 1974 e que ganham mais de 4000,00 OU aqueles que trabalham no setor 1 e que ganham mais de 1000,00.

JOINS (Junções)

Para que possamos recuperar informações de um banco de dados, temos, muitas vezes, a necessidade de acessar simultaneamente várias tabelas relacionadas entre si. Algumas dessas consultas necessitam realizar uma junção (Join) entre tabelas, para poder extrair dessa junção as informações necessárias para a consulta formulada.

Uma cláusula SQL combina registro de duas uma ou mais tabelas. Ela cria um conjunto que pode ser salvo como uma tabela ou utilizada de outra forma. Um JOIN é um meio de combinar campos de duas tabelas utilizando-se valores em comum. O padrão SQL ANSI especifica 4 tipos de Joins: Inner, Outer, Left e Right. Em casos especiais, uma tabela pode fazer JOIN consigo mesma, chamado self-join.

INNER JOIN (junção interna)

O tipo default de junção de uma tabela é a junção interna (inner join), na qual uma tupla é inclusa no resultado somente se existir uma tupla que combine na outra relação. Por exemplo, na consulta abaixo, somente os empregados que possuem um setor válido na tabela de SETORES serão inclusos no resultado. Uma tupla FUNCIONARIOS que possua no campo setor_id um valor NULL ou mesmo um código de setor que não exista na tabela setores não será incluída no resultado final.

Ex:

```
select funcionarios.*, setores.setor_nome
from funcionarios
inner join setores on funcionarios.setor_id = setores.setor_id
```

	func_id	func_nome	gerente_id	setor_id	func_salario	fund_dataNasc	setor_nome
1	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000	Faturamento
2	2	Ana	NULL	1	4780.00	1971-07-05 00:00:00.000	Faturamento
3	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000	Faturamento
4	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	Faturamento
5	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000	Faturamento
6	6	Eduardo	1	2	300.00	1991-08-05 00:00:00.000	Contabilidade
7	7	Anderson	1	2	500.00	1977-07-09 00:00:00.000	Contabilidade
8	8	Fábio	1	2	350.00	1989-05-06 00:00:00.000	Contabilidade
9	9	João	1	2	980.00	1981-02-03 00:00:00.000	Contabilidade
10	10	José	2	2	321.00	1981-01-01 00:00:00.000	Contabilidade
11	11	Ingrid	2	3	422.00	1982-02-02 00:00:00.000	RH
12	12	Bruno	2	3	890.00	1963-05-25 00:00:00.000	RH
13	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000	RH
14	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000	RH

SQL permite duas formas diferentes de expressar JOINS. A primeira, chamada de “notação join explícita” usa a palavra chave JOIN. A segunda, chamada de “notação join implícita” lista as tabelas para a junção na cláusula FROM, separando as tabelas por vírgula. O exemplo abaixo reproduz o mesmo resultado do exemplo apresentado acima.

```
select funcionarios.*, setores.setor_nome
from funcionarios, setores
where funcionarios.setor_id = setores.setor_id
```

OUTER JOIN (junção externa)

Dada 2 tabelas relacionadas R e S, um conjunto de operações, chamado junções externas (outer joins), pode ser utilizado quando queremos manter todas as tuplas em R, ou todas as tuplas em S, ou todas aquelas em ambas as relações no resultado da junção, independentemente se elas têm ou não tuplas correspondentes na outra relação. Isso satisfaz a necessidade de consultas nas quais as tuplas de duas tabelas são combinadas pelo casamento de linhas correspondentes, mas sem a perda de qualquer tupla por falta de valores casados.

LEFT JOIN (junção externa à esquerda)

Este caso de outer join mantém toda tupla na primeira relação ou na relação R à esquerda. Se nenhuma tupla correspondente é encontrada em S, então os atributos de S no resultado da junção serão preenchidos ou “enchidos” com valores NULL. O resultado dessas operações é mostrado no exemplo abaixo.

```
select funcionarios.*, setores.setor_nome
from funcionarios
LEFT OUTER join setores on funcionarios.setor_id = setores.setor_id

-- a palavra OUTER é opcional!

--Ou em SQL Server

select funcionarios.*, setores.setor_nome
from funcionarios, setores
where funcionarios.setor_id *= setores.setor_id
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc	setor_nome
1	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000	Faturamento
2	2	Ana	NULL	1	4780.00	1971-07-05 00:00:00.000	Faturamento
3	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000	Faturamento
4	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	Faturamento
5	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000	Faturamento
6	6	Eduardo	1	2	300.00	1991-08-05 00:00:00.000	Contabilidade
7	7	Anderson	1	2	500.00	1977-07-09 00:00:00.000	Contabilidade
8	8	Fábio	1	2	350.00	1989-05-06 00:00:00.000	Contabilidade
9	9	João	1	2			Contabilidade
10	10	José	2	2			Contabilidade
11	11	Ingrid	2	3			RH
12	12	Bruno	2	3			RH
13	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000	RH
14	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000	RH
15	15	Daniel	3	NULL	600.00	1984-12-06 00:00:00.000	NULL
16	16	Valdir	3	NULL	900.00	1984-07-09 00:00:00.000	NULL
17	17	Kleber	3	NULL	800.63	1957-02-02 00:00:00.000	NULL

Registros que não possuem um valor correspondente na tabela SETORES.

No exemplo acima, a tabela considerada “primeira relação” é a tabela que está no FROM, ou seja, a tabela FUNCIONARIOS. Como foi utilizado o LEFT OUTER JOIN (OUTER pode ser omitido), foram retornadas todas as tuplas da tabela FUNCIONÁRIOS, mesmo que o campo Setor_id seja NULL ou não haja um valor correspondente na tabela SETORES.

RIGHT JOIN (junção externa à direita)

Este caso de outer join mantém todas as tuplas na segunda relação, ou relação S à direita, no resultado final. Veja o exemplo abaixo.

```
select funcionarios.*, setores.setor_nome
from funcionarios
RIGHT OUTER join setores on funcionarios.setor_id = setores.setor_id

-- a palavra OUTER é opcional!

-- ou em SQL SERVER:
select funcionarios.*, setores.setor_nome
from funcionarios, setores
where funcionarios.setor_id =* setores.setor_id
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc	setor_nome
1	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000	Faturamento
2	2	Ana	NULL	1	4780.00	1971-07-05 00:00:00.000	Faturamento
3	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000	Faturamento
4	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	Faturamento
5	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000	Faturamento
6	6	Eduardo	1	2	300.00	1991-08-05 00:00:00.000	Contabilidade
7	7	Anderson	1			00:00:00.000	Contabilidade
8	8	Fábio	1			00:00:00.000	Contabilidade
9	9	João	1			00:00:00.000	Contabilidade
10	10	José	2			00:00:00.000	Contabilidade
11	11	Ingrid	2			00:00:00.000	RH
12	12	Bruno	2	3	890.00	1963-05-25 00:00:00.000	RH
13	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000	RH
14	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000	RH
15	NULL	NULL	NULL	NULL	NULL	NULL	Compras
16	NULL	NULL	NULL	NULL	NULL	NULL	Estoque
17	NULL	NULL	NULL	NULL	NULL	NULL	Farmácia
18	NULL	NULL	NULL	NULL	NULL	NULL	Recepção
19	NULL	NULL	NULL	NULL	NULL	NULL	Desenvolvimento
20	NULL	NULL	NULL	NULL	NULL	NULL	Engenharia
21	NULL	NULL	NULL	NULL	NULL	NULL	CAD

Registros da tabela SETORES que não possuem um valor correspondente na tabela FUNCIONARIOS.

Neste caso, foram retornados todos os registros da tabela SETORES, mesmo que não haja um correspondente na tabela FUNCIONARIOS.

FULL OUTER JOIN (junção externa total)

Este é outro caso de outer join, onde são mantidas todas as tuplas em ambas as relações, esquerda e direita, quando não são encontradas as tuplas com os valores casados, preenchendo-as com valores NULL conforme a necessidade.

```
select funcionarios.*, setores.setor_nome
from funcionarios
FULL OUTER join setores on funcionarios.setor_id = setores.setor_id

-- a palavra OUTER é opcional!
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc	setor_nome
1	15	Daniel	3	NULL	600.00	1984-12-06 00:00:00.000	NULL
2	16	Valdir	3	NULL	900.00	1984-07-09 00:00:00.000	NULL
3	17	Kleber	3	NULL	800.63	1957-02-02 00:00:00.000	NULL
4	1	Maria	NULL	1	5000.00	1974-10-25 00:00:00.000	Faturamento
5	2	Ana	NULL	1	4780.00	1971-07-05 00:00:00.000	Faturamento
6	3	Carla	NULL	1	7000.00	1974-10-10 00:00:00.000	Faturamento
7	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	Faturamento
8	5	Natália	1	1	700.00	1976-11-05 00:00:00.000	Faturamento
9	6	Lucas	1	2	1000.00	1980-05-05 00:00:00.000	Contabilidade
10	7	Thaís	1	2	1000.00	1980-05-05 00:00:00.000	Contabilidade
11	8	Thaís	1	2	1000.00	1980-05-05 00:00:00.000	Contabilidade
12	9	Thaís	1	2	1000.00	1980-05-05 00:00:00.000	Contabilidade
13	10	Thaís	1	2	1000.00	1980-05-05 00:00:00.000	Contabilidade
14	11	Ingrid	2	3	422.00	1982-02-02 00:00:00.000	RH
15	12	Bruno	2	3	890.00	1963-05-25 00:00:00.000	RH
16	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000	RH
17	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000	RH
18	NULL	NULL	NULL	NULL	NULL	NULL	Compras
19	NULL	NULL	NULL	NULL	NULL	NULL	Estoque
20	NULL	NULL	NULL	NULL	NULL	NULL	Farmácia
21	NULL	NULL	NULL	NULL	NULL	NULL	Recepção
22	NULL	NULL	NULL	NULL	NULL	NULL	Desenvolvimento
23	NULL	NULL	NULL	NULL	NULL	NULL	Engenharia
24	NULL	NULL	NULL	NULL	NULL	NULL	CAD

Todos os registros das duas tabelas foram retornados. Quando houve correspondência (os registros não marcados pelo retângulo vermelho) o SQL retornou os dados relacionados. Quando não houve correspondência (retângulo vermelho), os campos foram preenchidos com NULL no resultado.

SELF JOIN (Auto Junção)

Self Join é um tipo de junção onde é feito um JOIN de uma tabela com ela mesma. Veja o exemplo abaixo:

Vamos listar todos os funcionários com seus respectivos gerentes. Não há a tabela GERENTES, pois os gerentes também estão cadastrados na tabela FUNCIONARIOS. O que define um gerente de um funcionário é o preenchimento do código do funcionário no campo gerente_id. Vejamos como resolver este problema:

```
select funcionarios.*, Gerentes.func_nome as "Gerente_Nome"
from funcionarios
Inner join funcionarios as "gerentes" on gerentes.func_id = funcionarios.gerente_id

-- a palavra reservada "as" é opcional, assim como as aspas (")
```

	func_id	func_nome	gerente_id	setor_id	func_salario	fund_dataNasc	Gerente_Nome
1	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	Maria
2	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000	Maria
3	6	Eduardo	1	2	300.00	1991-08-05 00:00:00.000	Maria
4	7	Anderson	1	2	500.00	1977-07-09 00:00:00.000	Maria
5	8	Fábio	1	2	350.00	1989-05-06 00:00:00.000	Maria
6	9	João	1	2	980.00	1981-02-03 00:00:00.000	Maria
7	10	José	2	2	321.00	1981-01-01 00:00:00.000	Ana
8	11	Ingrid	2	3	422.00	1982-02-02 00:00:00.000	Ana
9	12	Bruno	2	3	890.00	1963-05-25 00:00:00.000	Ana
10	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000	Ana
11	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000	Ana
12	15	Daniel	3	NULL	600.00	1984-12-06 00:00:00.000	Carla
13	16	Valdir	3	NULL	900.00	1984-07-09 00:00:00.000	Carla
14	17	Kleber	3	NULL	800.63	1957-02-02 00:00:00.000	Carla

Como estamos trabalhando com duas tabelas de mesmo nome, precisamos de um apelido, um "alias" para uma das tabelas. Fizemos isso com a tabela utilizada no Inner Join, chamando-a de GERENTES. Também demos um apelido para o campo resultante do nome do funcionário que representa o gerente: gerente_nome.

Na expressão SQL acima, "GERENTES" é chamado de "alias" (apelido). Um apelido representa a mesma tabela a qual está referenciando. Um "alias" é muito importante quando há redundância nos nomes das colunas de duas ou mais tabelas que estão envolvidas em uma expressão. Ao invés de utilizar o "alias", é possível utilizar o nome da tabela, mas isto pode ficar cansativo em consultas muito complexas além do que, impossibilitaria a utilização da mesma tabela mais que uma vez em uma expressão SQL, como no exemplo acima.

Cláusula Union

Combina os resultados de duas ou mais consultas em um único conjunto de resultados, que inclui todas as linhas pertencentes a todas as consultas da união. A operação UNION é diferente de usar junções que combinam colunas de duas tabelas.

A seguir são apresentadas as regras básicas de combinação dos conjuntos de resultados de duas consultas usando UNION:

- ✓ O número e a ordem das colunas devem ser iguais em todas as consultas.
- ✓ Os tipos de dados devem ser compatíveis.

Caso seja necessário ordenar os dados, deve-se ordenar pelos campos indicados na primeira instrução SQL. Caso seja usado um alias, este deverá ser utilizado na ordenação.

Ex:

```
select funcionarios.func_id as "codigo", funcionarios.func_nome as "nome"
from funcionarios
where funcionarios.func_nome like 'A%'
union
select setores.setor_id , setores.setor_nome
from setores
```

	codigo	nome
1	1	Faturamento
2	2	Ana
3	2	Contabilidade
4	3	RH
5	4	Antonio
6	4	Compras
7	5	Estoque
8	6	Farmácia
9	7	Anderson
10	7	Recepção
11	8	Desenvolvimento
12	9	Engenharia
13	10	CAD

A cláusula **ALL** (em frente a cláusula Union) incorpora todas as linhas nos resultados. **Isto inclui duplicatas.** Se não for especificado, as linhas duplicadas serão removidas.

Sem a cláusula ALL

```
select funcionarios.func_id as "codigo", funcionarios.func_nome as "nome"
from funcionarios
where funcionarios.func_nome like 'A%'

union

select funcionarios.func_id as "codigo", funcionarios.func_nome as "nome"
from funcionarios
where funcionarios.func_nome like 'A%'
```

	codigo	nome
1	2	Ana
2	4	Antonio
3	7	Anderson

Com a cláusula ALL

```
select funcionarios.func_id as "codigo", funcionarios.func_nome as "nome"
from funcionarios
where funcionarios.func_nome like 'A%'

union all

select funcionarios.func_id as "codigo", funcionarios.func_nome as "nome"
from funcionarios
where funcionarios.func_nome like 'A%'
```

	codigo	nome
1	2	Ana
2	4	Antonio
3	7	Anderson
4	2	Ana
5	4	Antonio
6	7	Anderson

CAST e CONVERT

Esta função converte uma expressão de um tipo de dados para outro.

Obs: Há também a função CONVERT, que possui algumas funcionalidades especiais para o tipo de dados datetime.

Sintaxe:

```
CAST ( expression AS data_type [ (length) ] )
CONVERT ( data_type [ (length) ] , expression [ , style ] )
```

Onde:

expression : É qualquer expressão válida.

data_type : É o tipo de dados de designado.

length : É um inteiro opcional que especifica o comprimento do tipo de dados designado. O valor padrão é 30.

style : É uma expressão de inteiro que especifica como a função CONVERT deve ser convertida para *expression*.

Ex: Concatenando valores:

```
select '1' + '1'
```

	(No column name)
1	11

Ex: convertendo valores e efetuando uma soma

```
select cast('1' as int) + cast('1' as int)
```

	(No column name)
1	2

Ex. da função Convert:

```
--103 British/French dd/mm/yy
select convert(varchar, getdate(), 103)
```

	(No column name)
1	16/02/2009

Ex. Retornando apenas o ano com a função substring:

```
-- SUBSTRING ( expression ,start , length )
select substring( convert(varchar, getdate(), 103), 7,4)
```

	(No column name)
1	2009

Ex: O resultado acima poderia ser executado também de outra forma:

```
select year( getdate() )
```

```
-- há também as funções day e month, além da função DATEPART ( porção , data )
```

	(No column name)
1	2009

Subconsultas (Subqueries ou Consultas Encadeadas)

Uma subconsulta é uma consulta aninhada em uma instrução SELECT, INSERT, UPDATE ou DELETE, ou em subconsulta. Uma subconsulta pode ser usada em qualquer lugar em que é permitida uma expressão, exceto na cláusula ORDER BY. Neste exemplo, uma subconsulta é usada como uma expressão de coluna denominada MaxUnitPrice em uma instrução SELECT.

Uma subconsulta também é chamada de uma consulta interna ou seleção interna, enquanto a instrução que contém uma subconsulta também é chamada de uma consulta externa ou seleção externa. **Na maioria das vezes, há uma junção entre a consulta interna e a consulta externa.**

Muitas instruções Transact-SQL que incluem subconsultas podem ser alternativamente formuladas como junções. Outras perguntas só podem ser feitas com subconsultas. Em Transact-SQL, normalmente não há nenhuma diferença de desempenho entre uma instrução que inclui uma subconsulta e uma versão equivalente semanticamente que não inclui. Entretanto, em alguns casos em que a existência deve ser verificada, uma junção tem um desempenho melhor. Em outros casos, a consulta aninhada deve ser processada para cada resultado da consulta externa para assegurar a eliminação de duplicatas. Em tais casos, uma abordagem de junção geraria resultados melhores.

(A consulta SELECT de uma subconsulta sempre é incluída em parênteses.)

EX. 1: Utilizando um subconsulta para retornar um campo adicional:

Este tipo de subconsulta só pode retornar 1 registro e 1 coluna.

No exemplo abaixo, serão retornados todos os funcionários e seus respectivos gerentes. Este mesmo exemplo foi utilizado para explicar SELFJOIN, porém utilizando JOIN no lugar de subconsulta.

```
select funcionarios.*,
       (select Gerentes.func_nome from funcionarios as "Gerentes"
        where gerentes.func_id = funcionarios.gerente_id) as "Gerente_Nome"
from funcionarios
where gerente_id is not null
```

	func_id	func_nome	gerente_id	setor_id	func_salario	fund_dataNasc	Gerente_Nome
1	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000	Maria
2	5	Nestor	1	1	700.00	1949-11-05 00:00:00.000	Maria
3	6	Eduardo	1	2	300.00	1991-08-05 00:00:00.000	Maria
4	7	Anderson	1	2	500.00	1977-07-09 00:00:00.000	Maria
5	8	Fábio	1	2	350.00	1989-05-06 00:00:00.000	Maria
6	9	João	1	2	980.00	1981-02-03 00:00:00.000	Maria
7	10	José	2	2	321.00	1981-01-01 00:00:00.000	Ana
8	11	Ingrid	2	3	422.00	1982-02-02 00:00:00.000	Ana
9	12	Bruno	2	3	890.00	1963-05-25 00:00:00.000	Ana
10	13	Bruna	2	3	1021.00	1977-12-03 00:00:00.000	Ana
11	14	Daniela	2	3	2050.00	1984-11-01 00:00:00.000	Ana
12	15	Daniel	3	NULL	600.00	1984-12-06 00:00:00.000	Carla
13	16	Valdir	3	NULL	900.00	1984-07-09 00:00:00.000	Carla
14	17	Kleber	3	NULL	800.63	1957-02-02 00:00:00.000	Carla

Observe que para cada registro da tabela externa FUNCIONARIOS, o SQL irá fazer uma subconsulta na tabela GERENTES para pesquisar o nome do mesmo.

No exemplo abaixo, iremos retornar os setores e a sua quantidade de funcionários.

```
select setores.*,
       (select count(*) from funcionarios where
        funcionarios.setor_id = setores.setor_id) as "Total de Funcionários"
from setores
```

-- poderíamos substituir as "" por []

	setor_id	setor_nome	Total de Funcionários
1	1	Faturamento	5
2	2	Contabilidade	5
3	3	RH	4
4	4	Compras	0
5	5	Estoque	0
6	6	Farmácia	0
7	7	Recepção	0
8	8	Desenvolvimento	0
9	9	Engenharia	0
10	10	CAD	0

EX. 2: Utilizando um subconsulta para restringir os registros retornados

Podemos utilizar uma subconsulta para restringir os registros que são retornados para o resultado final. O resultado de uma subconsulta apresentada com IN (ou com NOT IN) é uma lista com zeros ou outros valores. Depois dos resultados da subconsulta retornarem, a consulta exterior os utiliza.

Ex: No exemplo a seguir, iremos retornar os setores e a sua quantidade de funcionários, porém, diferentemente do exemplo anterior, iremos retornar apenas aqueles que possuem ao menos 1 funcionário. Obs: Há várias outras formas de se resolver este problema.

```
select setores.*,
       (select count(*) from funcionarios where
        funcionarios.setor_id = setores.setor_id) as "Total de Funcionários"
from setores
where (select count(*) from funcionarios where
       funcionarios.setor_id = setores.setor_id) > 0
```

--ou

```
select setores.*,
       (select count(*) from funcionarios where
        funcionarios.setor_id = setores.setor_id) as "Total de Funcionários"
from setores
where exists(select func_id from funcionarios where
             funcionarios.setor_id = setores.setor_id)
```

	setor_id	setor_nome	Total de Funcionários
1	1	Faturamento	5
2	2	Contabilidade	5
3	3	RH	4

EX. 3: Aplicação da Cláusula TOP e Operadores In e Not In em subconsultas

O operador TOP (qtde) (SQL SERVER) limita a quantidade de registros retornada pela instrução SQL. Ex: Se executarmos a instrução SQL:

```
Select TOP 1 * from setores
```

Apenas 1 registro (tupla) irá retornar. Podemos utilizá-lo com quaisquer outras cláusulas, como Order by, Where, etc. Esta cláusula é interessante quando sabemos exatamente a quantidade de registros que precisamos, evitando o processamento de informações desnecessárias.

Os operador IN (aqui ele tem o sentido de “está contido”) é um operador de conjunto. Podemos utilizado, por exemplo, para selecionar os funcionários que pertençam a um dos setores: 1, 2, 5 ou 7. A sintaxe ficaria assim:

```
Select * from funcionarios
where setor_id IN (1,2,5,7)
```

Da mesma forma, se quisermos todos os funcionários que NÃO pertençam aos setores 1, 2, 5 ou 7, basta introduzir o operador NOT na sintaxe anterior:

```
Select * from funcionarios
where setor_id NOT IN (1,2,5,7)
```

Observe que o operador IN opera sobre um conjunto de dados. E esse conjunto pode ser proveniente do resultado de uma subconsulta, desde que ela retorne apenas 1 coluna (campo).

Ex: Novamente, vamos retornar apenas os setores que possuem, ao menos, 1 funcionário.

```
Select * from setores
where setor_id in (select funcionarios.setor_id from funcionarios
                   where funcionarios.setor_id = setores.setor_id)

/* para melhorar o desempenho, podemos utilizar o operador TOP, pois nos interessa
saber se há pelo menos 1 funcionário no setor, não sendo necessário retornar todas
as tuplas da subconsulta! */

Select * from setores
where setor_id in (select TOP 1 funcionarios.setor_id from funcionarios
                   where funcionarios.setor_id = setores.setor_id)
```

	setor_id	setor_nome
1	1	Faturamento
2	2	Contabilidade
3	3	RH

O resultado acima pode ser obtido sem a utilização de subconsultas, utilizando-se a cláusula DISTINCT.

Visões (Views)

View é uma instrução que retorna dados e é salva no banco de dados com um nome, ou seja, passa a ser um objeto do banco de dados. Quando uma view é executada, esta retorna um conjunto de dados no formato de uma tabela. Uma view pode retornar dados de uma ou mais tabelas.

As principais vantagens na utilização de views são:

- ✓ Ocultar a complexidade do acesso a dados, criando uma view que consolida o acesso a várias tabelas;
- ✓ Simplificar a atribuição de permissões, uma vez que uma view pode ser pressionada de forma mais simples do que se fossem pressionadas todas as tabelas que a constituem.
- ✓ Facilitar o desenvolvimento e manutenção de aplicações, dado que se alguma tabela sofrer alguma alteração, a aplicação terá que ser modificada, o que pode ser evitado caso seja utilizada uma view.

As principais limitações na utilizações de view são:

- ✓ Somente podemos utilizar uma view no banco de dados atual, embora este possa acessar dados de tabelas de outros bancos de dados ou até mesmo de outros servidores.
- ✓ Uma view pode ter, no máximo, 10234 colunas.
- ✓ O usuário que está criando a view deve, obviamente, ter acesso a todas as colunas incluídas na view.
- ✓ Podemos aninhar view, isto é, uma view pode referenciar outra, até um nível de 32 views aninhadas.

É possível modificar os dados de uma tabela base subjacente através de uma view, porém há uma série de restrições que não serão tratadas aqui. Para saber mais sobre este assunto, acesso o link MSDN abaixo:

<http://msdn.microsoft.com/pt-br/library/ms180800.aspx>

Tipos de view que podem ser criadas:

■ Visão Idêntica

TABELA

A	B	C

VISÃO

A	B	C

■ Visão por Seleção de Colunas

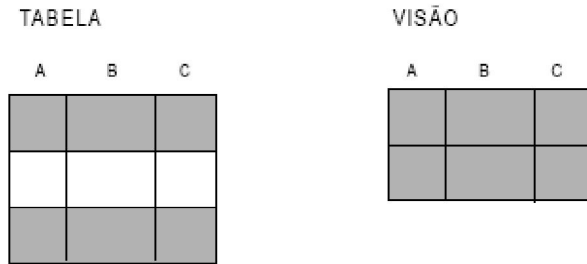
TABELA

A	B	C

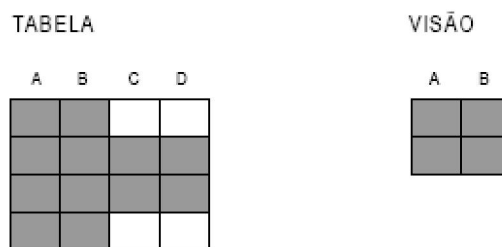
VISÃO

A	C

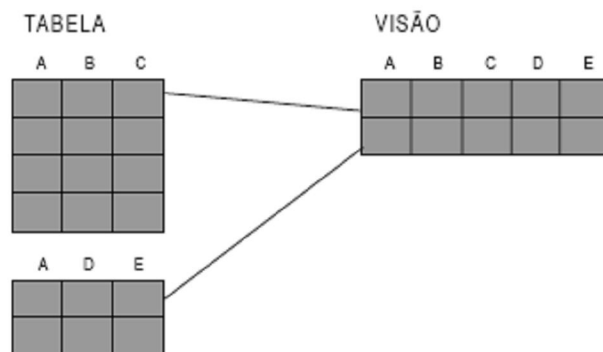
■ Visão por Seleção de Linhas



■ Visão por Seleção de Linhas e Colunas



■ Visão por Junção de Tabelas



Sintaxe básica para criação de uma view:

```
CREATE VIEW nome_da_view
AS instrução_sql
```

Onde:

Nome_da_view : É o nome da exibição. Os nomes de exibição devem seguir as regras para identificadores. A especificação do nome do proprietário da exibição é opcional.

Instrução_sql: É a instrução SELECT que define a exibição. A instrução pode usar mais de uma tabela e outras exibições. Permissões apropriadas são necessárias para selecionar os objetos referenciados na cláusula SELECT da exibição criada.

Uma exibição não tem que ser um subconjunto simples das linhas e colunas de uma determinada tabela. Uma exibição pode ser criada usando mais de uma tabela ou outras exibições com uma cláusula VIEW de qualquer complexidade.

Em uma definição de exibição indexada, a instrução SELECT deve ser uma instrução de tabela simples ou uma JOIN de várias tabelas com agregação opcional.

As cláusulas SELECT em uma definição de exibição não podem incluir o seguinte:

- ✓ Cláusulas COMPUTE ou COMPUTE BY
- ✓ Uma cláusula ORDER BY, a não ser que exista também uma cláusula TOP na lista de seleção da instrução SELECT
- ✓ A palavra-chave INTO
- ✓ A cláusula OPTION
- ✓ Uma referência para uma tabela temporária ou uma variável de tabela.

Ex:

```
create view vw_func_setor
as
select funcionarios.func_nome as "Nome do funcionario",
       setores.setor_nome as "Nome do Setor"
from funcionarios
inner join setores on setores.setor_id = funcionarios.setor_id
```

Ao selecionarmos os dados da view, o resultado será:

```
select * from vw_func_setor
```

	Nome do funcionario	Nome do Setor
1	Maria	Faturamento
2	Ana	Faturamento
3	Carla	Faturamento
4	Antonio	Faturamento
5	Nestor	Faturamento
6	Eduardo	Contabilidade
7	Anderson	Contabilidade
8	Fábio	Contabilidade
9	João	Contabilidade
10	José	Contabilidade
11	Ingrid	RH
12	Bruno	RH
13	Bruna	RH
14	Daniela	RH

Caso seja necessário alterar a view, utilize o commando ALTER no lugar do create.

Ex:

```
alter view vw_func_setor
as
select funcionarios.func_nome as "Nome do funcionario",
       setores.setor_nome as "Nome do Setor"
from funcionarios
inner join setores on setores.setor_id = funcionarios.setor_id and
                    setores.setor_nome like 'F%'
```

Após alterar qualquer uma das tabelas que compõe uma view, é necessário atualizar a view com o comando `sp_refreshview nome_da_view`.

Ex: `sp_refreshview vw_func_setor`

Principais estruturas do Transact-SQL

BEGIN...END (Transact-SQL)

Engloba uma série de instruções Transact-SQL de modo que um grupo de instruções Transact-SQL pode ser executado. BEGIN e END são palavras-chave da linguagem de controle de fluxo.

Sintaxe:

```
BEGIN
    comandos
END
```

IF...ELSE (Transact-SQL)

Impõe condições na execução de uma instrução Transact-SQL. A instrução Transact-SQL que segue uma palavra-chave IF e sua condição será executada se a condição for satisfeita: a expressão Booleana retorna TRUE. A palavra-chave opcional ELSE introduz outra instrução Transact-SQL que será executada quando a condição IF não for satisfeita: a expressão Booleana retorna FALSE.

Sintaxe

```
IF Boolean_expression
    { sql_statement | statement_block }
[ ELSE
    { sql_statement | statement_block } ]
```

A parte em azul é opcional.

Argumentos

- ***Boolean_expression***: É uma expressão que retorna TRUE ou FALSE. Se a expressão booleana contiver uma instrução SELECT, a instrução SELECT deverá ser incluída entre parênteses.
- ***{sql_statement | statement_block}***: É qualquer instrução ou agrupamento de instruções Transact-SQL, conforme definido por meio de um bloco de instruções. A menos que um bloco de instruções seja usado, a condição IF ou ELSE poderá afetar o desempenho de somente uma instrução Transact-SQL.

Para definir um bloco de instruções, use as palavras-chave BEGIN e END de controle de fluxo.

Os testes IF podem ser aninhados depois de outro IF ou seguindo um ELSE. O limite do número de níveis aninhados depende da memória disponível.

WHILE (Transact-SQL)

Define uma condição para a execução repetida de uma instrução ou um bloco de instruções SQL. As instruções serão executadas repetidamente desde que a condição especificada seja verdadeira. A execução de instruções no loop WHILE pode ser controlada internamente ao loop com as palavras-chave BREAK e CONTINUE.

Sintaxe

```
WHILE Boolean_expression
{ sql_statement | statement_block | BREAK | CONTINUE }
```

Argumentos

- **Boolean_expression:** É uma expressão que retorna **TRUE** ou **FALSE**. Se a expressão booleana contiver uma instrução SELECT, a instrução SELECT deverá ser incluída entre parênteses.
- **{sql_statement | statement_block}:** É qualquer instrução Transact-SQL ou agrupamento de instruções, conforme definido com um bloco de instruções. Para definir um bloco de instruções, use as palavras-chave BEGIN e END de controle de fluxo.
- **BREAK:** Causa uma saída do loop WHILE mais interno. Quaisquer instruções que apareçam depois da palavra-chave END, que marca o final do loop, serão executadas.
- **CONTINUE:** Faz com que o loop WHILE seja reiniciado, ignorando quaisquer instruções depois da palavra-chave CONTINUE.

Se dois ou mais loops WHILE estiverem aninhados, o BREAK interno será encerrado para o próximo loop mais externo. Todas as instruções após o fim da primeira execução do loop interno e o loop mais externo seguinte serão reiniciadas.

CASE (Transact-SQL)

Avalia uma lista de condições e retorna uma das várias expressões de resultado possíveis.

A expressão CASE tem dois formatos:

- A expressão CASE simples compara uma expressão com um conjunto de expressões simples para determinar o resultado.
- A expressão CASE pesquisada avalia um conjunto de expressões booleanas para determinar o resultado.

Os dois formatos dão suporte a um argumento ELSE opcional.

CASE pode ser usada em qualquer instrução ou cláusula que permita uma expressão válida. Por exemplo, você pode usar CASE em instruções, como SELECT, UPDATE, DELETE e SET, e em cláusulas, como select_list, IN, WHERE, ORDER BY e HAVING.

Sintaxe

Simple CASE expression:

```
CASE input_expression
  WHEN when_expression THEN result_expression [ ...n ]
  [ ELSE else_result_expression ]
END
```

Searched CASE expression:

```
CASE
  WHEN Boolean_expression THEN result_expression [ ...n ]
  [ ELSE else_result_expression ]
END
```

Argumentos

- **input_expression:** É a expressão avaliada quando o formato CASE simples é usado. *input_expression* é qualquer expressão válida.
- **WHEN when_expression:** É uma expressão simples à qual *input_expression* é comparada quando o formato CASE simples é usado. *when_expression* é qualquer expressão válida. Os tipos de dados de *input_expression* e cada *when_expression* devem ser iguais ou ser uma conversão implícita.

- **THEN *result_expression*:** É a expressão retornada quando *input_expression* igual a *when_expression* é avaliada como TRUE ou *Boolean_expression* é avaliada como TRUE. *result_expression* é qualquer expressão válida.
- **ELSE *else_result_expression*:** É a expressão retornada se nenhuma operação de comparação for avaliada como TRUE. Se esse argumento for omitido e nenhuma operação de comparação for avaliada como TRUE, CASE retornará NULL. *else_result_expression* é qualquer expressão válida. Os tipos de dados de *else_result_expression* e qualquer *result_expression* devem ser iguais ou ser uma conversão implícita.
- **WHEN *Boolean_expression*:** É a expressão booleana avaliada ao usar o formato CASE pesquisado. *Boolean_expression* é qualquer expressão booleana válida.

Valores de retorno

Expressão CASE simples:

A expressão CASE simples opera comparando a primeira expressão com a expressão em cada cláusula WHEN por equivalência. Se essas expressões forem equivalentes, a expressão na cláusula THEN será retornada.

- Permite somente uma verificação de igualdade.
- Avalia *input_expression* e, na ordem especificada, avalia *input_expression* = *when_expression* para cada cláusula WHEN.
- Retorna a *result_expression* da primeira *input_expression* = *when_expression* avaliada como TRUE.
- Se nenhuma *input_expression* = *when_expression* for avaliada como TRUE, o Mecanismo de Banco de Dados do SQL Server retornará a *else_result_expression*, se uma cláusula ELSE for especificada, ou um valor NULL, se nenhuma cláusula ELSE for especificada.

Exemplo:

```
USE AdventureWorks;
GO
SELECT    ProductNumber, Category =
          CASE ProductLine
            WHEN 'R' THEN 'Road'
            WHEN 'M' THEN 'Mountain'
            WHEN 'T' THEN 'Touring'
            WHEN 'S' THEN 'Other sale items'
            ELSE 'Not for sale'
          END,
          Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

Expressão CASE pesquisada:

- Na ordem especificada, avalia *Boolean_expression* para cada cláusula WHEN.
- Retorna a *result_expression* da primeira *Boolean_expression* avaliada como TRUE.
- Se nenhuma *Boolean_expression* for avaliada como TRUE, o Mecanismo de Banco de Dados retornará a *else_result_expression*, se uma cláusula ELSE for especificada, ou um valor NULL, se nenhuma cláusula ELSE for especificada.

Exemplo:

```
select func_nome, func_salario,
       case
         when func_salario >=0 and func_salario <= 500 then 'Assalariado'
         when func_salario >500 and func_salario <= 2000 then 'Salário padrão'
         when func_salario > 2000 then 'Cargo de confiança'
         else 'Salário negativo!!!!???'
       end as "Situação"
from funcionarios
```

	func_nome	func_salario	Situação
1	Maria	5000.00	Cargo de confiança
2	Ana	4780.00	Cargo de confiança
3	Carla	7000.00	Cargo de confiança
4	Antonio	1000.00	Salário padrão
5	Nestor	700.00	Salário padrão
6	Eduardo	300.00	Assalariado
7	Anderson	500.00	Assalariado
8	Fábio	350.00	Assalariado
9	João	980.00	Salário padrão
10	José	321.00	Assalariado
11	Ingrid	422.00	Assalariado
12	Bruno	890.00	Salário padrão
13	Bruna	1021.00	Salário padrão
14	Daniela	2050.00	Cargo de confiança
15	Daniel	600.00	Salário padrão
16	Valdir	900.00	Salário padrão
17	Kleber	800.63	Salário padrão

O SQL Server é permitido somente para 10 níveis de aninhamento em expressões CASE.

A expressão CASE não pode ser usada para controlar o fluxo de execução de instruções Transact-SQL, blocos de instruções, funções definidas pelo usuário e procedimentos armazenados. Para obter uma lista dos métodos de controle de fluxo, consulte Linguagem de controle de fluxo (Transact-SQL).

Stored Procedures (Procedimentos Armazenados)

Fontes adicionais: http://lmasters.uol.com.br/artigo/7932/bancodedados/dominando_stored_procedures/
<http://www.devmedia.com.br/articles/viewcomp.asp?comp=2213>

Uma Stored Procedure (Procedimento Armazenado) ou simplesmente SP, é uma coleção de instruções implementadas com linguagem T-SQL (Transact-Sql, no SQL Server 2000/2005), que, uma vez armazenadas ou salvas, ficam dentro do servidor de forma pré-compilada, aguardando que um usuário do banco de dados faça sua execução. Geralmente, assim como VIEWS fazem com relatórios e dados estatísticos escalonáveis, os SP's encapsulam tarefas repetitivas, desde um simples INSERT, passando por inserções por lote, updates e algumas outras instruções mais complexas, como, efetuar uma efetivação de saque em uma conta de um determinado cliente em uma instituição bancária ou efetivar saídas de mercadorias seguido por baixa em estoque. Elas oferecem suporte a variáveis declaradas pelo próprio usuário, uso de expressões condicionais, de laço e muitos outros recursos, os quais veremos alguns mais à frente.

As vantagens no uso de Stored Procedures são claras:

- Modularidade: passamos a ter o procedimento dividido das outras partes do software, bastando alterarmos somente as suas operações para que se tenha as modificações por toda a aplicação;
- Diminuição de I/O: uma vez que são passados parâmetros para o servidor, chamando o procedimento armazenado, as operações são executadas usando-se o processamento do servidor. No final da execução, podem ser retornados os resultados de uma transação para o cliente, sendo assim, não há um tráfego imenso e rotineiro de dados pela rede;
- Rapidez na execução: as stored procedures, após salvas no servidor, ficam somente aguardando, já em uma posição da memória cache, serem chamadas para executarem uma operação, ou seja, como estão pré-compiladas, as ações também já estão pré-carregadas, dependendo somente dos valores dos parâmetros. Após a primeira execução, elas se tornam ainda mais rápidas;

- Segurança de dados: podemos também, ocultar a complexidade do banco de dados para usuários, deixando que sejam acessados somente dados pertinentes ao tipo de permissão atribuída ao usuário ou mesmo declarando se a Stored Procedure é proprietária ou pública, podendo ser também criptografada.

Existem certos tipos de Stored Procedures para o SQL Server 2000 e para o SQL Server 2005, são eles:

- **System Stored Procedures ou Procedimentos Armazenados do Sistema:** nas duas versões do SGBD, são criados no momento da instalação e ficam armazenados no banco de dados chamado master, junto com as entidades e outros procedimentos próprios do sistema. São utilizados das mais diversas formas. Um exemplo clássico é o SP_HELPINDEX, para checar os índices de uma determinada tabela, ou o SP_REFRESHVIEW para criar uma View.
- **Local Stored Procedure ou Procedimentos Armazenados locais:** esses procedimentos são criados em bancos de dados de usuários individuais, ou seja, no SQL Server 2000 são proprietárias. Já no SQL Server 2005, tem o nome de Stored Procedures Temporárias Locais, que iniciam com um sinal de # e somente a conexão que a criou poderá executá-la. Ainda no SQL Server 2005, existem os Procedimentos Temporários Globais, que podem ser utilizados de forma global, ou seja, por qualquer usuário desta conexão e iniciam com ##. Ao ser encerrada a conexão onde a SP foi criada, elas serão eliminadas automaticamente.
- **Extended Stored Procedure ou Procedimentos Armazenados Extendidos:** são comuns às duas versões do SGBD Microsoft. Executam funções externas e do sistema operacional e iniciam com "xp_". Esses procedimentos são implementados como Dynamic-link Librarys (DLL), executadas fora do ambiente do SQL Server.
- **User-Defined Stored Procedure ou Procedimento Armazenados Definidos pelo Usuário:** estes são criados em bancos de dados pelo próprio usuário, como o nome já diz. São utilizados para realizar tarefas repetitivas, facilitando a manutenção e alteração, já que estão em um só ponto da aplicação/Banco de Dados.

Alguns cuidados devem ser ressaltados, antes mesmo de começarmos a criar nossos primeiros Procedimentos Armazenados:

- Temos a nossa disposição ou à disposição de tais procedimentos, um espaço máximo de 128 MB, o que nos deixa bem à vontade;
- Procedimentos Armazenados podem fazer referência à VIEWS e TRIGGERS, bem como à tabelas temporárias;
- Caso um Stored Procedure crie uma tabela temporária local, essa tabela somente existirá até o fim da execução do procedimento;
- Somente os usuários que são membros da role de servidor SYSADMIN ou da role de Banco de Dados db_owner e db_admin têm permissão para executar o comando CREATE PROCEDURE, permissão que pode ser atribuída por membro da role SYSADMIN.
- Os comandos: CREATE DEFAULT, CREATE RULE, CREATE TRIGGER e CREATE VIEW não podem estar em meio a uma construção de Stored Procedure.

SINTAXE BÁSICA:


```
CREATE PROCEDURE <nome_procedimento>
```

```
    @param 1 data type,  
    @param 2 data type,  
    @param n data type...
```

```
WITH...
```

```
AS
```

```
    Statement1  
    Statement2  
    Statement3  
    Statementn...
```

```
GO
```

Uma SP para retornar todos os registros da tabela funcionarios:

```
create procedure Retorna_funcionarios as  
  
select * from funcionarios
```

Claro que poderíamos fazer a mesma coisa sem a necessidade de uma SP!

Para executá-la:

```
exec retorna_funcionarios
```

Exemplo de uma SP para validar CPF:

```
CREATE procedure validaCPF (@CPF as varchar(11) )  
  
as  
BEGIN  
  
-- declaração das variáveis locais  
declare @n int  
declare @soma int  
declare @multi int  
declare @digito1 int  
declare @digito2 int  
  
if len(rtrim(ltrim(@CPF))) <> 11  
begin  
    Return 0 -- sai da stored procedure caso o CPF esteja no tamanho incorreto  
end  
  
-- calculando o primeiro digito...  
set @soma = 0  
set @multi = 10  
set @n = 1
```

```

WHILE (@n <= 9 )
begin
    set @soma = @soma + cast(SUBSTRING(@cpf, @n, 1) as int) * @multi;
    set @multi = @multi -1;

    set @n = @n + 1
end

set @soma = @soma % 11 -- % -> módulo

if @soma <=1
    set @digito1 = 0
else
    set @digito1 = 11 - @soma

--calculando o segundo digito...
set @soma = 0
set @multi = 11
set @n = 1

WHILE (@n <= 9 )
begin
    set @soma = @soma + cast(SUBSTRING(@cpf, @n, 1) as int) * @multi;
    set @multi = @multi -1;

    set @n = @n + 1
end

set @soma = (@soma + @digito1 * @multi);

set @soma = @soma % 11 -- % -> módulo

if @soma <=1
    set @digito2 = 0
else
    set @digito2 = 11 - @soma

--comparando os digitos digitados com os calculados...

--print 'digito 1: ' + cast( @digito1 as varchar)
--print 'digito 2: ' + cast( @digito2 as varchar)
--print char(13) -- pula uma linha!!!

if (cast(SUBSTRING(@cpf, 10, 1) as int) = @digito1) and
   (cast(SUBSTRING(@cpf, 11, 1) as int) = @digito2)
    Return 1
else
    Return 0

END

```

Para testar a SP acima, digite:

```

declare @retorno int
declare @CPF varchar(15);
set @CPF = '12345678909'

exec @retorno = validaCPF @CPF

```

```

if (@retorno = 0)
    print 'Incorreto'
else
    print 'Correto'

```

Parâmetros de Entrada e Saída

O tipo de parâmetro default do SQL é sempre o de entrada. Para criar um parâmetro de saída, utilize a palavra reservada **output** após a declaração do nome do parâmetro e também no momento da chamada (execução) Exemplo:

```

create procedure sp_calcula
(
    @valor1 int,
    @valor2 int,
    @resultado int output
)
as
Begin
    set @resultado = @valor1 + @valor2
end

```

Para executar a SP acima, digite:

```

declare @resultado int
execute sp_calcula 5,7, @resultado output
print 'Resultado: ' + cast (@resultado as varchar)

```

SP's que acessam dados do banco

Ex: Stored Procedure para gravar um setor na tabela setores. Caso o setor já exista, a SP retorna 0 ou 1 caso seja gravado com sucesso.

```

create procedure sp_inserirSetor( @codigo int, @descricao varchar(50) )
as
begin
    if not exists( select setor_id from setores where setor_id = @codigo )
    begin
        insert into setores ( setor_id, setor_nome )
        values (@codigo, @descricao )
        return 1 -- gravado com sucesso!
    end
    else
        return 0 -- erro ao gravar! registro já existe.
    end
end

```

Para testar a SP abaixo, digite:

```
declare @retorno int

execute @retorno = sp_insererSetor 11, 'Teste'

if @retorno = 1
    print 'Gravado com sucesso'
else
    print 'Registro já existe.'
```

Tabelas temporárias

Você pode criar tabelas temporárias globais e locais. Tabelas temporárias locais (# no prefixo) são visíveis somente na sessão corrente, e tabelas temporárias globais (## no prefixo) são visíveis para todas as sessões.

O exemplo abaixo cria uma tabela temporária visível apenas na sessão do usuário que a criou (local).

```
create table #temporaria(
    codigo int,
    descricao varchar(30)
```

Tabelas temporárias são automaticamente apagadas quando elas estiverem fora de escopo, ou quando for utilizado explicitamente o comando drop table:

- Uma tabela temporária local criada em uma SP é apagada automaticamente quando a SP é terminada. Ela pode ser referenciada por outra SP aninhada dentro da SP que a criou.
- Todas as tabelas temporárias locais são apagadas automaticamente ao fim da sessão corrente.
- Tabelas temporárias globais são apagadas automaticamente quando a sessão que a criou terminar e também quando todas as outras tarefas que a estavam referenciando terminarem.

A SP abaixo irá retornar todos os registros da tabela funcionários com o salário atualizado. O campo gerente_id também foi removido. Porém, essas alterações serão realizadas em uma tabela temporária. A tabela original ainda conterá os registros sem qualquer alteração. Não seria necessário criar uma tabela temporária para resolver este problema....

```
alter procedure sp_atualizaSalario
as

select * into #TabTemp from funcionarios --cria um tablea temporária com a mesma
-- estrutura da tabela funcionarios

-- atualiza o salario da seguinte forma: cada ano de vida equivale a 1% a mais no
-- salário do funcionario
update #TabTemp set func_salario = func_salario *
    cast(
        '1.' + cast( DATEDIFF ( year , func_dataNasc , getdate() ) as varchar)
        as decimal(8,2) )

-- remove a coluna gerente_id da tabela temporária
alter table #TabTemp drop column gerente_id

-- retorna os dados da tabela temporária
select * from #TabTemp
```

Para executar:

```
execute sp_atualizaSalario
```

O exemplo abaixo cria uma tabela temporária com apenas 4 campos e a preenche com os dados da tabela funcionário.

```
create procedure sp_funcionario_idade as

-- cria uma tabela temporária
create table #func_temp (
codigo int,
nome varchar(30),
data_nascimento datetime,
idade int)

-- insere os dados na tabela temporária
insert into #func_temp (codigo, nome, data_nascimento, idade)
select func_id, func_nome, func_datanasc, datediff( year, func_datanasc,
getdate())as "Idade"
from funcionarios

-- seleciona os dados da tabela temporária
select * from #func_temp
```

Para executá-la:

```
execute sp_funcionario_idade
```

Triggers (Gatilhos)

Fonte adicional: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=1695>

Uma trigger é um tipo especial de procedimento armazenado, que é executado sempre que há uma tentativa de modificar os dados (insert, update e delete) da tabela onde ela foi configurada..

- Chamados Automaticamente

Quando há uma tentativa de inserir, atualizar ou excluir os dados em uma tabela, e um TRIGGER tiver sido definido na tabela para essa ação específica, ele será executado automaticamente, não podendo nunca ser ignorado.

- Não podem ser chamados diretamente

Ao contrário dos procedimentos armazenados do sistema, os disparadores não podem ser chamados diretamente e não passam nem aceitam parâmetros.

- É parte de uma transação

O TRIGGER e a instrução que o aciona são tratados como uma única transação, que poderá ser revertida em qualquer ponto do procedimento, caso você queira usar "ROLLBACK", conceitos que veremos mais a frente.

Orientações básicas quando estiver usando TRIGGER.

- As definições de TRIGGERS podem conter uma instrução "ROLLBACK TRANSACTION", mesmo que não exista uma instrução explícita de "BEGIN TRANSACTION";
- Se uma instrução "ROLLBACK TRANSACTION" for encontrada, então toda a transação (a TRIGGER e a instrução que o disparou) será revertida ou desfeita. Se uma instrução no script da TRIGGER seguir uma instrução "ROLLBACK TRANSACTION", a instrução será executada, então, isso nos obriga a ter uma condição IF contendo uma cláusula RETURN para impedir o processamento de outras instruções.
- Não é uma boa prática utilizar "ROLLBACK TRANSACTION" dentro de suas TRIGGERS, pois isso gerará um retrabalho, afetando muito no desempenho de seu banco de dados, pois toda a consistência deverá ser feita quando uma transação falhar, lembrando que tanto a instrução quanto o TRIGGER formam uma única transação. O mais indicado é validar as informações fora das transações com TRIGGER, evitando que a transação seja desfeita.
- Para que um TRIGGER seja disparado, o usuário o qual entrou com as instruções, deverá ter permissão de acessar tanto a entidade e conseqüentemente ao TRIGGER.

Usos e aplicabilidade dos TRIGGERS

- Impor uma integridade de dados mais complexa do que uma restrição CHECK;
- Definir mensagens de erro personalizadas;
- Manter dados desnormalizados;
- Comparar a consistência dos dados – posterior e anterior – de uma instrução UPDATE;

As TRIGGERS são usadas com enorme eficiência para impor e manter integridade referencial de baixo nível, e não para retornar resultados de consultas. A principal vantagem é que elas podem conter uma lógica de processamento complexa.

Você pode usar TRIGGERS para atualizações e exclusões em cascata através de tabelas relacionadas em um banco de dados, impor integridades mais complexas do que uma restrição CHECK, definir mensagens de erro personalizadas, manter dados desnormalizados e fazer comparações dos momentos anteriores e posteriores a uma transação.

Quando queremos efetuar transações em cascata, por exemplo, uma TRIGGER de exclusão na tabela **PRODUTO** do banco de dados *Northwind* pode excluir os registros correspondentes em outras tabelas que possuem registros com os mesmos valores de **PRODUCTID** excluídos para que não haja quebra na integridade, como a dito popular “pai pode não ter filhos, mas filhos sem um pai não existe”.

Você pode utilizar os TRIGGERS para impor integridade referencial da seguinte maneira:

- Executando uma ação ou atualizações e exclusões em cascata:

A integridade referencial pode ser definida através do uso das restrições *FOREIGN KEY* e *REFERENCE*, com a instrução *CREATE TABLE*. As TRIGGERS fazem bem o trabalho de checagem de violações e garantem que haja coerência de acordo com a sua regra de negócios. Se você exclui um cliente, de certo, você terá que excluir também todo o seu histórico de movimentações. Não seria boa coisa se somente uma parte desta transação acontecesse.

- Criando disparadores de vários registros:

Quando mais de um registro é atualizado, inserido ou excluído, você deve implementar um TRIGGER para manipular vários registros.

CRIANDO TRIGGERS

As TRIGGERS são criadas utilizando a instrução *CREATE TRIGGER* que especifica a tabela onde ela atuará, para que tipo de ação ela irá disparar suas ações seguido pela instrução de conferência para disparo da ação. E meio a esses comandos, temos algumas restrições para o bom funcionamento. O SQL Server não permite que as instruções a seguir, sejam utilizadas na definição de uma TRIGGER:

- *ALTER DATABASE;*
- *CREATE DATABASE;*
- *DISKINIT;*
- *DISKRESIZE;*
- *DROP DATABASE;*
- *LOAD DATABASE;*
- *LOAD LOG;*
- *RECONFIGURE;*
- *REATORRE DATABASE;*
- *RESTORELOG.*

COMO FUNCIONAM OS TRIGGERS

- *Como funciona um TRIGGER INSERT;*
- *Como funciona um TRIGGER DELETE;*
- *Como funciona um TRIGGER UPDATE;*

Quando incluímos, excluímos ao alteramos algum registro em um banco de dados, são criadas tabelas temporárias que passam a conter os registros excluídos, inseridos e também o antes e depois de uma atualização.

Quando você exclui um determinado registro de uma tabela, na verdade você estará apagando a referência desse registro, que ficará, após o DELETE, numa tabela temporária de nome DELETED. Uma TRIGGER implementada com uma instrução SELECT poderá lhe trazer todos ou um número de registro que foram excluídos.

Assim como acontece com DELETE, também ocorrerá com inserções em tabelas, podendo obter os dados ou o número de linhas afetadas buscando na tabela INSERTED.

Já no UPDATE ou atualização de registros em uma tabela, temos uma variação e uma concatenação para verificar o antes e o depois.

De fato, quando executamos uma instrução UPDATE, a “engine” de qualquer banco de dados tem um trabalho semelhante, primeiro exclui os dados tupla e posteriormente faz a inserção do novo registro que ocupará aquela posição na tabela, ou seja, um DELETE seguido por um INSERT. Quando então, há uma atualização, podemos buscar o antes e o depois, pois o antes estará na tabela DELETED e o depois estará na tabela INSERTED.

Nada nos impede de retornar dados das tabelas temporárias (INSERTED, DELETED) de volta às tabelas de nosso banco de dados, mas atente-se, os dados manipulados são temporários assim como as tabelas e só estarão disponíveis nesta conexão. Após fechar, as tabelas e os dados não serão mais acessíveis.

TRIGGER INSERT

De acordo com a sua vontade, uma TRIGGER por lhe enviar mensagens de erro ou sucesso, de acordo com as transações. Estas mensagens podem ser definidas em meio a um TRIGGER utilizando condicionais IF e indicando em PRINT sua mensagem personalizada. Por exemplo, vamos criar uma tabela chamada tbl_usuario, com a qual simularemos um cadastro de usuários que você também poderá criar para fazer os seus testes. A cada inserção de novo usuário, podemos exibir uma mensagem personalizada de sucesso na inserção.

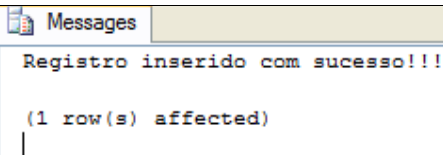
EX: A trigger abaixo será disparada sempre que um novo registro for inserido na tabela Funcionários:

```
create trigger trg_onInsertValue on Funcionarios
for INSERT
AS

    if (select count(*) from inserted) = 1
        print 'Registro inserido com sucesso!!!'
```

Para testá-lo, basta inserir um funcionário. Ex:

```
insert into funcionarios (func_id, func_nome)
values(100, 'xxx')
```



The screenshot shows a 'Messages' window with the following text: 'Registro inserido com sucesso!!!' and '(1 row(s) affected)'.

Existem várias outras abordagens para o uso de TRIGGERS com INSERTS, por exemplo, quando se faz um controle de entrada e saídas de produtos, podemos ter uma TRIGGER executando a baixa dos produtos no estoque (entrada) diante daqueles que foram solicitados num pedido (saída). Uma TRIGGER pode automatizar essa rotina.

TRIGGER DELETE

Podemos usar uma TRIGGER para monitorar certas exclusões de registros de acordo com a sua regra de negócios e também para proteger a integridade dos dados em um determinado banco de dados.

Alguns fatos devem ser considerados ao usar TRIGGERS DELETE:

- Quando um registro é acrescentado à tabela temporária DELETED, ele deixa de existir na tabela do banco de dados. Portanto, a tabela DELETED não apresentará registros em comum com as tabelas do banco de dados;

- É alocado espaço na memória para criar a tabela DELETED, que está sempre em cache;
- Um TRIGGER para uma ação DELETE não é executado para a instrução TRUNCATE TABLE porque a mesma não é registrada no log de transações.

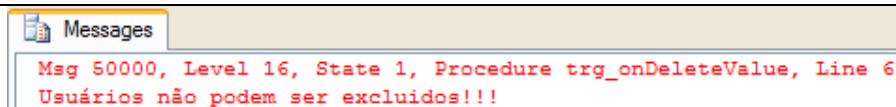
Podemos criar um TRIGGER não permita a exclusão de usuários de nossa tabela funcionarios, da seguinte forma:

```
create trigger trg onDeleteValue on Funcionarios
for DELETE
AS
    if (select count(*) from DELETED) >= 1
        RAISERROR('Usuários não podem ser excluidos!!!', 16, 1)

    ROLLBACK transaction
```

Para testar a trigger:

```
delete from funcionarios
```



Msg 50000, Level 16, State 1, Procedure trg_onDeleteValue, Line 6
Usuários não podem ser excluidos!!!

TRIGGER UPDATE

Partindo então do princípio que uma instrução *UPDATE* se apresenta em duas etapas(delete e insert), com ela podemos verificar o antes e o depois, já que os dados estarão disponíveis nas tabelas temporárias *DELETED* – momento anterior – *INSERTED* – momento atual, logo após o *UPDATE*.

A qualquer momento daqui por diante, poderemos então contemplar, sempre que executada uma instrução para atualização de registros, o antes e o depois, o que acabamos de definir no TRIGGER para *UPDATE*.

Vamos então, executar a instrução para atualizar o único registro que temos então em nossa tabela. Vamos alterar o nome do funcionário nº 100 para YYY. Veremos então como era o nome antes e como é o nome agora!

Primeiro, vamos criar a trigger de update

```
create trigger trg onUpdateValue on Funcionarios
for UPDATE
AS
    if (select count(*) from deleted) <> 0
    begin
        select * from DELETED
        select * from INSERTED
    end
```

Agora, vamos alterar o nome de XXX para YYY do funcionário n. 100:

```
update funcionarios set func_nome = 'YYY' where func_id = 100
```

	func_id	func_nome	gerente_id	setor_id
1	100	xxx	NULL	NULL

	func_id	func_nome	gerente_id	setor_id
1	100	YYY	NULL	NULL

Para apagar uma trigger: `Drop trigger nome_da_trigger`

Habilitando e Desabilitando Triggers

Podemos desabilitar temporariamente uma trigger. Para desabilitar uma trigger utilizamos o comando:

```
alter table nome_da_tabela
disable trigger nome_da_trigger
```

Ex: desabilitando a trigger de INSERT da tabela funcionarios:

```
alter table funcionarios
disable trigger trg_onInsertValue
```

Para saber as triggers associadas a uma tabela, digite:

```
exec sp_helpTrigger @tabname = funcionarios
```

	trigger_name	trigger_owner	isupdate	isdelete	isinsert	isafter	insteadof
1	trg_onInsertValue	dbo	0	0	1	1	0
2	trg_onDeleteValue	dbo	0	1	0	1	0

Para visualizar o código fonte de uma trigger (também funciona com Stored Procedures), digite:

```
exec sp_helpText trg_onInsertValue
```

	Text
1	create trigger trg_onInsertValue on Funcionarios
2	for INSERT
3	AS
4	
5	if (select count(*) from inserted) = 1
6	print 'Registro inserido com sucesso!!!'

Cursosores (Transact-SQL)

Fonte: <http://msdn.microsoft.com/pt-br/library/ms180169.aspx>

As instruções Microsoft SQL Server produzem um conjunto de resultados completo, mas há momentos em que os resultados são processados melhor uma linha de cada vez. Abrir um cursor em um conjunto de resultados permite o processamento do conjunto de resultados uma linha por vez. Você pode atribuir um cursor a uma variável ou a um parâmetro com um tipo de dados cursor.

Operações de cursor têm suporte nestas instruções:

Os cursosres estendem o resultado de processamento por:

- Permitirem o posicionamento em linhas específicas do conjunto de resultados.
- Recuperarem uma linha ou bloco de linhas de posições atuais em um conjunto de resultados.
- Oferecerem suporte às modificações de dados nas linhas da posição atual no conjunto de resultados.
- Oferecerem suporte a diferentes níveis de visibilidade às mudanças feitas por outros usuários ao banco de dados que são apresentados no conjunto de resultados.

```
Transact-SQL Extended Syntax
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
    [ FORWARD_ONLY | SCROLL ]
    [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
    [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
    FOR select_statement
[ ; ]
```

cursor_name

É o nome cursor de servidor definido Transact-SQL. *cursor_name* deve atender às regras para identificadores.

SCROLL

Especifica que todas as opções de busca (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) estão disponíveis. Se SCROLL não for especificado, NEXT será a única opção de busca com suporte. SCROLL não poderá ser especificado se FAST_FORWARD também for especificado.

READ ONLY

Previne atualizações feitas por este cursor. O cursor não pode ser referenciado em uma cláusula WHERE CURRENT OF, em uma instrução UPDATE ou DELETE. Essa opção anula a capacidade padrão de um cursor ser atualizado.

LOCAL

Especifica que o escopo do cursor é local para o lote, procedimento armazenado, ou acionador no qual o cursor foi criado. O nome de cursor só é válido dentro desse escopo. O cursor pode ser referenciado através de variáveis de cursor local no parâmetro OUTPUT do lote, acionador ou procedimento armazenado. Um parâmetro OUTPUT é usado para devolver o cursor local ao lote procedimento armazenado ou acionador de chamada, que pode nomear o parâmetro a uma variável de cursor para referenciar o cursor, depois que o procedimento armazenado terminar. O cursor é implicitamente desalocado quando o lote, procedimento armazenado ou acionador é encerrado, a menos que o cursor tenha sido repassado como um parâmetro OUTPUT. Se for repassado em um parâmetro OUTPUT, o cursor será desalocado quando a última variável que referenciada for desalocada ou extrapolar o escopo.

GLOBAL

Especifica que o escopo do cursor é global para a conexão. O nome de cursor pode ser referenciado em qualquer procedimento armazenado ou lote executado pela conexão. O cursor só é desalocado implicitamente na desconexão.

FORWARD_ONLY

Especifica se o cursor só pode ser rolado da primeira à última linha. FETCH NEXT é a única opção de busca com suporte. Se FORWARD_ONLY for especificado sem as palavras-chave STATIC, KEYSET ou DYNAMIC, o cursor operará como um cursor DYNAMIC. Quando FORWARD_ONLY ou SCROLL não estiverem especificados, FORWARD_ONLY será o padrão; a não ser que as palavras-chave STATIC, KEYSET ou DYNAMIC estejam especificadas. Os cursores STATIC, KEYSET e DYNAMIC seguem o padrão SCROLL. Diferentemente das APIs de banco de dados, como ODBC e ADO, os cursores Transact-SQL STATIC, KEYSET e DYNAMIC oferecem suporte para FORWARD_ONLY

STATIC

Define um cursor que faz uma cópia temporária dos dados a serem usados por ele. Todas as solicitações para o cursor são respondidas dessa tabela temporária em **tempdb**; logo, as modificações feitas na tabelas base não são refletidas nos dados retornados de buscas feitas nesse cursor, que não permite modificações.

KEYSET

Especifica que a associação e a ordem de linhas no cursor são fixas, quando o cursor é aberto. O conjunto de chaves que identificam exclusivamente as linhas é construído em uma tabela no **tempdb**, conhecida como **keyset**.

Alterações em valores não chave nas tabelas base, realizadas pelo proprietário do cursor ou confirmadas por outros usuários, são visíveis como rolagens do proprietário ao redor do cursor. Inserções feitas por outros usuários não são visíveis (não é possível fazer inserções por um cursor de servidor Transact-SQL). Se uma linha for excluída, uma tentativa de buscá-la retorna um @@FETCH_STATUS de -2. Atualização de valores de chave externos ao cursor lembram a exclusão de uma linha antiga, seguida de uma inserção de uma nova linha. A linha com os novos valores não é visível, e as tentativas de se buscar a linha com os valores antigos retornam um @@FETCH_STATUS de -2. Os novos valores ficarão visíveis se a atualização for feita através do cursor, especificando-se a cláusula WHERE CURRENT OF.

DYNAMIC

Define um cursor que reflete todas as mudanças de dados feitas às linhas no seu conjunto de resultados conforme você rola o cursor. Os valores de dados, ordem e associação das linhas podem sofrer alterações a cada busca. Cursores dinâmicos não oferecem suporte para a opção de busca ABSOLUTE.

FAST_FORWARD

Especifica um cursor FORWARD_ONLY, READ_ONLY, com otimizações de desempenho habilitadas. FAST_FORWARD não poderá ser especificado se SCROLL ou FOR_UPDATE também o for.

READ_ONLY

Previne atualizações feitas por este cursor. O cursor não pode ser referenciado em uma cláusula WHERE CURRENT OF, em uma instrução UPDATE ou DELETE. Essa opção anula a funcionalidade padrão de um cursor para ser atualizado.

SCROLL_LOCKS

Especifica que as atualizações ou exclusões posicionadas realizadas pelo cursor terão sucesso garantido. O SQL Server bloqueia as linhas como são lidas no cursor, para assegurar sua disponibilidade para

modificações posteriores. `SCROLL_LOCKS` não poderá ser especificado se `FAST_FORWARD` ou `STATIC` também o forem.

OPTIMISTIC

Especifica que as atualizações e exclusões posicionadas realizadas pelo cursor não terão sucesso se a linha foi atualizada desde que foi lida no cursor. O SQL Server não bloqueia linhas como são lidas no cursor. Em vez disso, ele usa comparações de valores de coluna **timestamp**, ou um valor de soma de verificação se a tabela não tiver nenhuma coluna **timestamp**, para determinar se a linha foi modificada depois de lida no cursor. Se a linha fosse modificada, a tentativa de atualização ou exclusão posicionada falharia. `OPTIMISTIC` não poderá ser especificado se `FAST_FORWARD` também for especificado.

select_statement

É uma instrução `SELECT` padrão que define o conjunto de resultados de um cursor. As palavras-chave `COMPUTE`, `COMPUTE BY`, `FOR BROWSE` e `INTO` não são permitidas em *select_statement*, de uma declaração de cursor.

FETCH (Transact-SQL)

Aqui está descrita apenas a sintaxe deste comando. Para mais informações, visite: <http://msdn.microsoft.com/pt-br/library/ms180152.aspx>

Recupera uma linha específica de um cursor de servidor Transact-SQL. A função `@@FETCH_STATUS` informa o status da última instrução `FETCH`.

Sintaxe

```

FETCH
    [ [ NEXT | PRIOR | FIRST | LAST
      | ABSOLUTE { n | @nvar }
      | RELATIVE { n | @nvar }
    ]
    FROM
    ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]

```

@@FETCH_STATUS (Transact-SQL)

Retorna o status do último cursor que a instrução `FETCH` emitiu em relação a **qualquer** cursor atualmente aberto pela conexão.

Sintaxe

```
@@FETCH_STATUS : integer
```

Valor de retorno

Valor de retorno Descrição

- | | |
|----|---|
| 0 | A instrução <code>FETCH</code> foi bem-sucedida. |
| -1 | A instrução <code>FETCH</code> falhou ou a linha estava além do conjunto de resultados. |
| -2 | A linha buscada está ausente. |

Exemplos

Um exemplo simples, onde são retornados todos os nomes de funcionários que contenham a letra 'd' no nome:

```
-- variáveis que serão utilizadas pelo cursor
declare @func_nome varchar(50)

-- declaração do cursor
declare cursor_teste cursor for
    select func_nome from funcionarios where func_nome like '%d%'

-- abertura do cursor
open cursor_teste

-- vai para o primeiro registro e preenche as variáveis correspondentes
FETCH NEXT FROM cursor_teste
INTO @func_nome

while @@fetch_Status = 0 -- enquanto houver registros, faça
Begin
    -- imprime os dados das variáveis
    print 'Nome: ' + @func_nome

    -- vai para o próximo registro e preenche as variáveis correspondentes
    FETCH NEXT FROM cursor_teste
    INTO @func_nome
end

close cursor_Testes -- fecha o cursor
deallocate cursor_teste -- desaloca o cursor da memória
```

```
Nome: Eduardo Silva
Nome: Anderson Silva
Nome: Ingrid Silva
Nome: Daniela
Nome: Daniel
Nome: Valdir Silva
```

Vamos incluir um pouco de lógica no exemplo acima: O cursor abaixo imprime o código e o nome e uma “situação salarial” de todos os funcionários que contenham a letra 'd' no nome.

```
-- variáveis que serão utilizadas pelo cursor
declare @func_id int
declare @func_nome varchar(50)
declare @func_salario money

-- variável auxiliar
declare @status varchar(15)

-- declaração do cursor
declare cursor_teste cursor for
    select func_id, func_nome, func_salario from funcionarios
    where func_nome like '%d%'

-- abertura do cursor
open cursor_teste

-- vai para o primeiro registro e preenche as variáveis correspondentes
FETCH NEXT FROM cursor_teste
INTO @func_id, @func_nome, @func_salario
```

```

while @@fetch_Status = 0 -- enquanto houver registros, faça
Begin

    if @func_salario < 500
        set @status = 'Ganha pouco'
    else if @func_salario < 1000
        set @status = 'Ganha +- '
    else
        set @status = 'Ganha bem!'

    -- imprime os dados das variáveis
    print 'Código ' + cast( @func_id as varchar) +
        '      Nome: ' + @func_nome + '      Status: ' + @status

    -- vai para o próximo registro e preenche as variáveis correspondentes
    FETCH NEXT FROM cursor_teste
    INTO @func_id, @func_nome, @func_salario
end

close cursor_Testes -- fecha o cursor
deallocate cursor_teste -- desaloca o cursor da memória

```

```

Código 6   Nome: Eduardo Silva   Status: Ganha pouco
Código 7   Nome: Anderson Silva  Status: Ganha +-
Código 11  Nome: Ingrid Silva    Status: Ganha pouco
Código 14  Nome: Daniela         Status: Ganha bem!
Código 15  Nome: Daniel          Status: Ganha +-
Código 16  Nome: Valdir Silva    Status: Ganha +-

```

O exemplo abaixo faz a exibição dos dados na ordem inversa: Observe a opção SCROLL na declaração do cursor!

```

declare @func_id int
declare @func_nome varchar(50)

declare cursor_teste SCROLL cursor for
    select func_id, func_nome from funcionarios

open cursor_teste

FETCH LAST FROM cursor_teste
INTO @func_id, @func_nome

while @@fetch_Status = 0
Begin
    print cast(@func_id as varchar(5)) + ' ' + @func_nome

    FETCH PRIOR FROM cursor_teste INTO @func_id, @func_nome
end

close cursor_Testes
deallocate cursor_teste

```

```

17 Kleber Silva
16 Valdir Silva
15 Daniel
14 Daniela
13 Bruna Silva
12 Bruno Silva
11 Ingrid Silva
10 José
9 João
8 Fábio
7 Anderson Silva
6 Eduardo Silva
5 Nestor Silva
4 Antonio

```

```

3  Carla Silva
2  Ana
1  Maria Silva

```

O exemplo abaixo mostra como alterar os dados de um cursor, usando o comando **update** com a opção **current of**:

A opção **current of cursor_name** na cláusula **WHERE** de um comando **Update** serve para atualizar o registro atual em **cursor_name** sem a necessidade de se especificar a chave da tabela.

```

declare @func_id int
declare @func_nome varchar(50)

declare cursor_teste cursor for select func_id, func_nome from funcionarios

open cursor_teste

FETCH NEXT FROM cursor_teste
INTO @func_id, @func_nome

while @@fetch_Status = 0
Begin
    if charindex('k', @func_nome) != 0
    begin
        set @func_nome = @func_nome + ' Souza' -- se tiver a letra k no nome

        update funcionarios
        set func_nome= @func_nome
        where current of cursor_teste
    end

    print cast(@func_id as varchar(5)) + ' ' + @func_nome

    FETCH NEXT FROM cursor_teste
    INTO @func_id, @func_nome
end

close cursor_Testes
deallocate cursor_teste

```

```

1  Maria Silva
2  Ana
3  Carla Silva
4  Antonio
5  Nestor Silva
6  Eduardo Silva
7  Anderson Silva
8  Fábio
9  João
10 José
11 Ingrid Silva
12 Bruno Silva
13 Bruna Silva
14 Daniela
15 Daniel
16 Valdir Silva

(1 row(s) affected)
17 Kleber Silva Souza

```


Funções definidas pelo usuário (FUNCTIONS)

<http://msdn.microsoft.com/pt-br/library/ms186755.aspx>

Este tema possui muitos detalhes que não serão abordados em aula, porém, podem ser consultados no link acima.

Assim como as funções em linguagens de programação, as funções definidas pelo usuário no Microsoft SQL Server são rotinas que aceitam parâmetros, executam uma ação, como um cálculo complexo e retornam o resultado dessa ação como um valor. O valor de retorno pode ser um único valor escalar ou um conjunto de resultados.

Benefícios da função definida pelo usuário

Os benefícios de usar funções definidas pelo usuário em SQL Server são:

- **Elas permitem programação modular:** Você pode criar a função uma vez, armazená-la no banco de dados e chamá-la quantas vezes quiser em seu programa. Funções definidas pelo usuário podem ser modificadas independentemente do código-fonte do programa.
- **Elas permitem execução mais rápida:** Semelhantemente aos procedimentos armazenados, Transact-SQL as funções definidas pelo usuário reduzem o custo de compilação do código Transact-SQL colocando os planos em cache e reusando-os para execuções repetidas. Isso significa que a função definida pelo usuário não precisa ser reanalisada e reotimizada em cada utilização resultando em tempos de execução mais rápidos.
- As funções CLR oferecem uma vantagem de desempenho significativa sobre funções Transact-SQL para tarefas de computação, manipulação de cadeias de caracteres e lógica de negócios. Funções Transact-SQL são mais adequadas à lógica intensiva de acesso a dados.
- **Elas podem reduzir o tráfego de rede:** Uma operação que filtra dados com base em alguma restrição complexa que não pode ser expressa em uma única expressão escalar pode ser expressa como uma função. A função pode ser invocada, então, na cláusula WHERE para reduzir o número ou linhas enviadas ao cliente.

Componentes de uma função definida pelo usuário

Funções definidas pelo usuário podem ser escritas em Transact-SQL ou em qualquer linguagem de programação .NET. Todas as funções definidas pelo usuário têm a mesma estrutura de duas partes: um cabeçalho e um corpo. A função obtém zero ou mais parâmetros de entrada e retorna um valor escalar ou uma tabela.

O cabeçalho define:

- O nome da função com o nome do esquema/proprietário opcional
- O nome do parâmetro de entrada e tipo de dados
- Opções aplicáveis ao parâmetro de entrada
- O tipo de dados e nome opcional do parâmetro de retorno
- Opções aplicáveis ao parâmetro de retorno
- O corpo define a ação ou lógica, a função a ser executada. Ou contém:
 - Uma ou mais instruções Transact-SQL que executam a lógica de função
 - Uma referência a um assembly .NET

As funções definidas pelo usuário são modificadas usando **ALTER FUNCTION** e descartadas usando **DROP FUNCTION**.

Funções definidas pelo usuário são de **valor escalar** ou de **valor de tabela**.

- Funções serão de **valor escalar** se a cláusula RETURNS tiver especificado um dos tipos de dados escalares. Funções com valor escalar podem ser definidas usando várias instruções Transact-SQL.
- Funções serão de **valor de tabela** se a cláusula RETURNS tiver especificado TABLE.

A sintaxe de uma function é bastante extensa, portanto ela será omitida. Para maiores detalhes consulte o link indicado no início do capítulo. Iremos diretamente aos exemplos.

Exemplo de valor escalar:

A função abaixo tem o objetivo de aplicar uma porcentagem sobre um valor qualquer.

```
create function AplicaAumento
(
    @p_valor decimal(10,2),
    @porcentagem_aumento decimal(10,2)
)
Returns decimal(10,2)

As
Begin
    return @p_valor + ( @p_valor * @porcentagem_aumento/100)
end
```

Para executar a função, devemos anteceder seu nome com **dbo**.

```
select dbo.AplicaAumento(1000, 10) as "Novo Valor"
```

	Novo Valor
1	1100.00

Ou então, podemos utilizá-la em uma instrução SQL que acessa dados de uma tabela:

```
select func_id, func_nome, func_salario as "salario_Anterior",
       dbo.AplicaAumento(func_salario, 10) as "Novo_Salario"
from funcionarios
where func_id <= 5
```

	func_id	func_nome	salario_Anterior	Novo_Salario
1	1	Maria Silva	5000.00	5500.00
2	2	Ana	4780.00	5258.00
3	3	Carla Silva	7000.00	7700.00
4	4	Antonio	1000.00	1100.00
5	5	Nestor Silva	700.00	770.00

Exemplo de valor de tabela:

```
create function fnc_RetornaTabelaFuncionarios
(
    @p_salario_inicial decimal(10,2),
    @p_salario_final decimal(10,2)
)
Returns table
as
Return
(
    select * from funcionarios
    where func_salario between @p_salario_inicial and @p_salario_final
)
```

Exemplo de execução:

```
select *
from dbo.fnc_RetornaTabelaFuncionarios(500, 1000)
where gerente_id = 1
```

	func_id	func_nome	gerente_id	setor_id	func_salario	func_dataNasc
1	4	Antonio	1	1	1000.00	1985-08-09 00:00:00.000
2	5	Nestor Silva	1	1	700.00	1949-11-05 00:00:00.000
3	7	Anderson Silva	1	2	500.00	1977-07-09 00:00:00.000
4	9	João	1	2	980.00	1981-02-03 00:00:00.000

Observe que, além do filtro que a function já aplica à tabela funcionários, referente ao salário, é possível também aplicar outros filtros na cláusula WHERE, como foi aplicado com relação ao campo gerente_id.

Também é possível criar uma tabela nova para ser retornada na function. Ex:

```
create function fnc_func_setor (@setor int)
Returns @tbl_func_Setor Table
(
    func_id int,
    func_name varchar(50),
    setor_id int,
    setor_nome varchar(50)
)
as
begin
    insert @tbl_func_Setor
    select funcionarios.func_id, funcionarios.func_nome,
           setores.setor_id, setores.setor_nome
    from funcionarios
    inner join setores on setores.setor_id = funcionarios.setor_id
    where funcionarios.setor_id = @setor

    Return
end
```

```
select * from dbo.fnc_func_setor(1)
```

	func_id	func_name	setor_id	setor_nome
1	1	Maria Silva	1	Faturamento
2	2	Ana	1	Faturamento
3	3	Carla Silva	1	Faturamento
4	4	Antonio	1	Faturamento
5	5	Nestor Silva	1	Faturamento

Controle de Transação

Uma transação é uma unidade lógica de trabalho e também uma unidade de recuperação (e ainda uma unidade de concorrência e uma unidade de integridade). As transações têm as propriedades ACID de atomicidade, consistência, isolamento e durabilidade. O gerenciamento de transações é a tarefa de supervisionar a execução de transações de tal modo que se possa, de fato, garantir que elas têm essas importantes propriedades. Com efeito, a função geral do sistema poderia ser definida como a execução confiável de transações.

As transações são iniciadas por `BEGIN TRANSACTION` e terminadas por `COMMIT TRANSACTION` (término bem-sucedido) ou por `ROLLBACK TRANSACTION` (término malsucedido).

- Um `COMMIT` estabelece um ponto de `COMMIT` (as atualizações se tornam permanentes).
- Um `ROLLBACK` rola o banco de dados para trás, até o ponto de `COMMIT` anterior (as atualizações são desfeitas).

Se uma transação não atingir o término planejado, o sistema forçará um `ROLLBACK` (recuperação de transações). Para ser capaz de desfazer (ou refazer) atualizações, o sistema mantém um log de recuperação. Além disso, os registros do log para uma dada transação devem ser gravados no log físico antes de poder ser concluído o processamento de `COMMIT` para essa transação (a regra de registro no log antes da gravação).

Faça o seguinte teste:

```
begin transaction
delete from setores
```

Agora execute a instrução abaixo:

```
select * from setores
```

setor_id	setor_nome

Acalme-se! Apesar de termos apagado TODOS os setores, o fizemos dentro de uma transação. Isso significa que ainda é possível fazer 2 coisas:

- Utilizar o comando `COMMIT` para efetivar (confirmar) o que foi apagado.
- Utilizar o comando `ROLLBACK` para desfazer tudo o que foi feito desde que a transação foi iniciada.

Sendo assim, vamos voltar os registros para a tabela de setores, desfazendo a instrução delete:

```
rollback
```

Agora execute novamente a instrução abaixo e veja que os registros voltaram!!!!

```
select * from setores
```

	setor_id	setor_nome
1	1	Faturamento
2	2	Contabilidade
3	3	RH
4	4	Compras
5	5	Estoque
6	6	Farmácia
7	7	Recepção
8	8	Desenvolvimento
9	9	Engenharia
10	10	CAD

TODAS os inserts, updates e deletes que forem executados dentro de uma transação poderão ser desfeitos ou efetivados.

Controle de exceção (TRY...CATCH)

Obs: Try Catch só está disponível a partir do SQL 2005. Para ver a versão do seu SQL SERVER, digite:

```
select @@version
```

Try-Catch implementa tratamento de erros para Transact-SQL semelhante ao tratamento de exceções nas linguagens Microsoft Visual C# e Microsoft Visual C++. Um grupo de instruções Transact-SQL pode ser incluído em um bloco TRY. Se ocorrer um erro no bloco TRY, o controle passará para outro grupo de instruções que está incluído em um bloco CATCH.

Sintaxe

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Argumentos

sql_statement : É qualquer instrução Transact-SQL.

statement_block : Qualquer grupo de instruções Transact-SQL em um lote ou incluso em um bloco BEGIN...END.

Uma construção TRY...CATCH captura todos os erros de execução com severidade maior que 10 e que não fechem a conexão do banco de dados. Veja abaixo a tabela de níveis severidade:

Níveis de severidade

A tabela a seguir lista e descreve os níveis de severidade dos erros gerados pelo Mecanismo de Banco de Dados do SQL Server.

Nível de severidade	Descrição
0-9	Mensagens informativas que retornam informações de status ou reportam erros que não sejam severos. O Mecanismo de Banco de Dados não gera erros de sistema com severidades de 0 a 9.
10	Mensagens informativas que retornam informações de status ou reportam erros que não sejam severos. Por razões de compatibilidade, o Mecanismo de Banco de Dados converte a severidade 10 em severidade 0 antes de retornar as informações de erro ao aplicativo de chamada.
11-16	Indica erros que podem ser corrigidos pelo usuário.
11	Indica que um determinado objeto ou entidade não existe.
12	Severidade especial para consultas que não usam bloqueio por causa de dicas de consulta especiais. Em alguns casos, operações de leitura executadas por essas instruções podem resultar em dados inconsistentes, pois os bloqueios não são usados para garantir a consistência.
13	Indica erros de deadlock de transação.
14	Indica erros relacionados à segurança, como uma permissão negada.
15	Indica erros de sintaxe no comando Transact-SQL.
16	Indica erros gerais que podem ser corrigidos pelo usuário.
17-19	Indica erros de software que não podem ser corrigidos pelo usuário. O usuário deve informar o problema ao seu administrador de sistema.
17	Indica que a instrução fez o SQL Server ficar sem recursos (como memória, bloqueios ou espaço em disco para o banco de dados) ou exceder algum limite definido pelo administrador de sistema.

18	Indica um problema no software Mecanismo de Banco de Dados, mas a instrução conclui a execução e a conexão com a instância do Mecanismo de Banco de Dados é mantida. O administrador de sistema deve ser informado sempre que uma mensagem com nível de severidade 18 ocorrer.
19	Indica que um limite do Mecanismo de Banco de Dados não configurável foi excedido e que o processo em lotes atual foi encerrado. Mensagens de erro com nível de severidade 19 ou maior pararam a execução do lote atual. Erros de severidade 19 são raros e devem ser corrigidos pelo administrador de sistema ou por seu principal provedor de suporte. Contate seu administrador de sistema quando uma mensagem com severidade de nível 19 ocorrer. Mensagens de erro com nível de severidade de 19 a 25 são gravadas no log de erros.
20-25	<p>Indique problemas de sistema que são erros fatais, ou seja, a tarefa do Mecanismo de Banco de Dados que está executando uma instrução ou um lote que não está mais em execução. A tarefa registra informações sobre o que aconteceu e, depois, é encerrada. Na maioria dos casos, a conexão do aplicativo com a instância do Mecanismo de Banco de Dados também pode ser encerrada. Se isso acontecer, dependendo do problema, é possível que o aplicativo não consiga se reconectar.</p> <p>Mensagens de erro nesse intervalo podem afetar todos os processos que acessam dados no mesmo banco de dados e indicar que um banco de dados ou objeto está danificado. Mensagens de erro com nível de severidade de 19 a 25 são gravadas no log de erros.</p>
20	Indica que uma instrução encontrou um problema. Como o problema afetou apenas a tarefa atual, é improvável que o banco de dados tenha sido danificado.
21	Indica que foi encontrado um problema que afeta todas as tarefas no banco de dados atual, mas é improvável que o banco de dados tenha sido danificado.
22	<p>Indica que a tabela ou o índice especificado na mensagem foi danificado por um problema de software ou hardware.</p> <p>Erros de severidade de nível 22 raramente ocorrem. Se acontecer, execute o DBCC CHECKDB para determinar se outros objetos no banco de dados também foram danificados. O problema pode ser apenas no cache do buffer e não no próprio disco. Nesse caso, reiniciar a instância do Mecanismo de Banco de Dados corrige o problema. Para continuar trabalhando, você deve reconectar-se à instância do Mecanismo de Banco de Dados; caso contrário, use o DBCC para corrigir o problema. Em alguns casos, pode ser necessário restaurar o banco de dados.</p> <p>Se a reinicialização da instância do Mecanismo de Banco de Dados não corrigir o problema, é porque o problema está no disco. Às vezes, destruir o objeto especificado na mensagem de erro pode resolver o problema. Por exemplo, se a mensagem informar que a instância do Mecanismo de Banco de Dados encontrou uma linha com comprimento 0 em um índice não-clusterizado, exclua o índice e crie-o novamente.</p>
23	<p>Indica que a integridade do banco de dados inteiro está em risco por um problema de software ou hardware.</p> <p>Erros de severidade de nível 23 raramente ocorrem. Se um acontecer, execute o DBCC CHECKDB para determinar a extensão do dano. O problema pode ser apenas no cache e não no próprio disco. Nesse caso, reiniciar a instância do Mecanismo de Banco de Dados corrige o problema. Para continuar trabalhando, você deve reconectar-se à instância do Mecanismo de Banco de Dados; caso contrário, use o DBCC para corrigir o problema. Em alguns casos, pode ser necessário restaurar o banco de dados.</p>
24	Indica uma falha de mídia. O administrador de sistema pode ter que restaurar o banco de dados. Também pode ser necessário contatar o seu fornecedor de hardware.

Um bloco TRY deve ser seguido imediatamente por um bloco CATCH associado. A inclusão de qualquer outra instrução entre as instruções END TRY e BEGIN CATCH gera um erro de sintaxe.

Se não houver erros no código incluído em um bloco TRY, quando a execução da última instrução no bloco TRY for concluída, o controle passará para a instrução imediatamente posterior à instrução END CATCH associada.

Se houver um erro no código incluído em um bloco TRY, o controle passará para a primeira instrução do bloco CATCH associado. Se a instrução END CATCH for a última instrução de um procedimento armazenado ou gatilho, o controle voltará para a instrução que chamou o procedimento armazenado ou acionou o gatilho.

Quando o código no bloco CATCH for concluído, o controle passará para a instrução imediatamente posterior à instrução END CATCH. Os erros interceptados por um bloco CATCH não são retornados ao aplicativo que o chamou. Se qualquer parte das informações de erro deve ser retornada ao aplicativo, o código no bloco CATCH deverá fazê-lo usando mecanismos como conjuntos de resultados SELECT ou as instruções RAISERROR e PRINT.

As construções TRY...CATCH podem ser aninhadas. Um bloco TRY ou um bloco CATCH pode conter construções TRY...CATCH aninhadas. Por exemplo, um bloco CATCH pode conter uma construção TRY...CATCH incorporada para tratar erros encontrados pelo código de CATCH.

Os erros encontrados em um bloco CATCH são tratados como erros gerados em qualquer outro lugar. Se o bloco CATCH contiver uma construção TRY...CATCH aninhada, qualquer erro no bloco TRY aninhado passará o controle para o bloco CATCH aninhado. Se não houver nenhuma construção TRY...CATCH aninhada, o erro voltará para o chamador.

As construções TRY...CATCH capturam erros não-tratados de procedimentos armazenados ou gatilhos executados pelo código do bloco TRY. Como alternativa, os procedimentos armazenados ou os gatilhos podem conter suas próprias construções TRY...CATCH para tratar os erros gerados por seu código. Por exemplo, quando um bloco TRY executa um procedimento armazenado e ocorre um erro no procedimento, o erro pode ser tratado das seguintes maneiras:

- Se o procedimento armazenado não contiver sua própria construção TRY...CATCH, o erro retornará o controle para o bloco CATCH associado ao bloco TRY que contém a instrução EXECUTE.
- Se o procedimento armazenado contiver uma construção TRY...CATCH, o erro transferirá o controle para o bloco CATCH do procedimento armazenado. Quando o código do bloco CATCH for concluído, o controle voltará para a instrução imediatamente posterior à instrução EXECUTE que chamou o procedimento armazenado.

A construção TRY...CATCH não pode ser usada em uma função definida pelo usuário.

Recuperando informações de erro

No escopo de um bloco CATCH, as seguintes funções de sistema podem ser usadas para obter informações sobre o erro que causou a execução do bloco CATCH.

- ERROR_NUMBER() retorna o número do erro.
- ERROR_SEVERITY() retorna a severidade.
- ERROR_STATE() retorna o número do estado do erro.
- ERROR_PROCEDURE() retorna o nome do procedimento armazenado ou do gatilho em que o erro ocorreu.
- ERROR_LINE () retorna o número de linha dentro da rotina que causou o erro.
- ERROR_MESSAGE () retorna o texto completo da mensagem de erro. O texto inclui os valores fornecidos para quaisquer parâmetros substituíveis, como cumprimentos, nomes de objeto ou horas.

Essas funções retornarão NULL se forem chamadas fora do escopo do bloco CATCH. As informações de erro podem ser recuperadas com o uso dessas funções em qualquer lugar no escopo do bloco CATCH. Por exemplo, o script a seguir mostra um procedimento armazenado que contém funções de tratamento de erros. No bloco CATCH de uma construção TRY...CATCH, o procedimento armazenado é chamado e as informações sobre o erro são retornadas.

Ex:

```
BEGIN TRY
    select 1 / 0
END TRY
BEGIN CATCH
    declare @erro varchar(100)
    set @erro = 'Erro ao divisor. ' + char(13) +
               'Erro na linha: ' + cast( ERROR_LINE() as varchar) + char(13) +
               'Mensagem de Erro original: ' + ERROR_MESSAGE()
    RAISERROR (@erro , 16,1)
END CATCH
```

Usando TRY...CATCH em uma transação

O exemplo a seguir mostra como um bloco TRY...CATCH funciona dentro de uma transação. A instrução dentro do bloco TRY gera um erro de violação de chave primária.

```
BEGIN TRY
    BEGIN TRANSACTION

    --Alteramos todos os setores iguais a 3 para 4. Isso é permitido.
    update funcionarios
    set setor_id =4
    where setor_id = 3

    --Alteramos o código do setor 3 para 4.
    --Isso NÃO é permitido pois já existe um setor 4.
    --Vai ocorrer um erro de violação da chave primária
    update setores
    set setor_id = 4
    where setor_id = 3

    -- o commit NÃO será executado pois quando ocorrer o erro no
    -- Update anterior, o controle de execução irá para o bloco CATCH
    commit;
END TRY
BEGIN CATCH
    if @@TranCount > 0 -- se alguma coisa foi modificada no banco, o processo é desfeito
        rollback

    -- exibimos uma mensagem de erro personalizada.
    declare @erro varchar(300)
    set @erro = 'Erro ao executar transação! ' + char(13) +
        'Erro na linha: ' + cast( ERROR_LINE() as varchar) + char(13) +
        'Mensagem de Erro original: ' + ERROR_MESSAGE() + char(13) +
        'Todo o processo foi desfeito.'
    RAISERROR (@erro , 16,1)
END CATCH
```