

Datawarehouse

Filipe Fidalgo

ffidalgo@ipcb.pt

T-SQL

O Transact SQL é a extensão ao SQL-92, padronizada e publicada pela ANSI. Normalmente quando falamos em “extensão” ao SQL, imaginam-se apenas comandos SQL DML e DDL utilizados em aplicativos tipo OLTP. Porém o T-SQL vai muito além: é uma poderosa linguagem não apenas para manipular dados, mas capaz de executar rotinas administrativas relacionadas a segurança, integridade e manutenção do SQL Server.

Tipos de Dados MSSQL

	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

Tipos de Dados mais usados do MSql Server (existem mais ...)

Variáveis – Locais e Globais

O SQL Server suporta dois tipos de variáveis: locais e globais.

As variáveis locais são válidas dentro da transacção onde foram declaradas. Para a declaração de variáveis locais usamos a palavra chave Declare, seguida pelo nome da variável e pelo tipo de dado. Uma variável local deve ter como prefixo o caractere @.

Exemplo:

```
DECLARE @Nome nvarchar(30)
```

Podemos declarar várias variáveis no mesmo comando, separando-as com virgula:

```
DECLARE @Nome nvarchar(30), @idade INT
```

Para atribuir um valor à variável podemos utilizar Set ou Select:

--Exemplo 1

```
DECLARE @Nome varchar(30), @Idade int
```

```
SET @Nome = 'Filipe'
```

```
SELECT @Idade = 30
```

```
SELECT @Idade AS 'SET E SELECT'
```

Variáveis – Locais

```
DECLARE @Nome varchar(30), @Idade int
SET @Nome = 'Filipe'
SELECT @Idade = 30
SELECT @Idade AS 'SET E SELECT'
SELECT @Nome AS 'SET E SELECT'
SELECT @Idade
SELECT @Nome
```

Utilizando Select podemos atribuir valores a diversas variáveis num mesmo comando:

--Exemplo 2

```
DECLARE @Nome varchar(30), @Idade int
SELECT @Nome = 'FILIPE', @Idade = 30
SELECT @Nome, @IDADE AS 'SELECT'
```

Variáveis locais não podem ter um valor por Default.

Variáveis – Globais

As variáveis globais têm com prefixo @@ e não podem ser definidas pelo utilizador.

A sua função é retornar informação do sistema. Podemos obter o valor de uma variável global usando select:

```
SELECT @@Error  
ou print  
PRINT @@Error
```

Podemos ainda atribuir uma variável global a uma variável local. Isto é importante quando precisamos tratar o valor de uma variável global que pode ser alterado diversas vezes dentro da um mesma transacção.

Ao atribuirmos o valor a uma variável local, não corremos o risco de perder o valor desejado.

```
DECLARE @erro int  
DELECT @erro = @@Error --Erro provocado Delect em vez de Select  
PRINT @erro
```

Variáveis – Globais

Uma variável global pode ser utilizada como condição numa consulta:

```
SELECT @@CONNECTIONS AS 'CONEXÕES'
```

Vejamos algumas das variáveis globais do SQL Server:

Variável Global	Descrição
@@CONNECTIONS	Retorna o número de ligações e tentativas desde o início do SQL Server.
@@ERROR	Retorna o código do ultimo erro ocorrido
@@IDENTITY	Retorna o ultimo valor Identity inserido
@@ROWCOUNT	Retorna o número de linhas do último comando executado.
@@VERSION	Retorna diversas informações a respeito do servidor, como versão, data e CPU

Comentários e Print

Comentários

O T-SQL suporta dois tipos de comentários:

- O comentário até o final da linha, especificado pelos caracteres --
- e os comentários de bloco, iniciados por /* e terminados com */, por exemplo:

-- Este é um comentário de uma única linha /* Este é um comentário que pode ter diversas linhas */

Print

Retorna uma mensagem definida pelo utilizador.

PRINT 'Este é um exemplo do uso de print'

Podemos imprimir ainda variáveis locais ou globais:

--Exemplo

PRINT @@Version

Controle de Fluxo

O verdadeiro poder do T-SQL está no controle de fluxo, sem o qual todas as regras de negocio teriam de ser controladas pelo cliente. Normalmente os SGDB não têm uma linguagem para controle de fluxo tão poderosa como algumas linguagens de programação, mas proporcionam os recursos necessários para a implementação das regras de negócio.

IF...ELSE

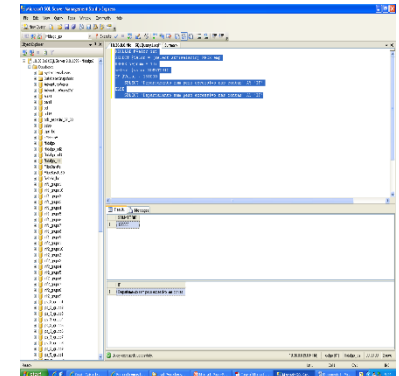
IF...ELSE oferece controle de fluxo condicional, a sintaxe é descrita a seguir:

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

Controle de Fluxo – If ...Else

No exemplo seguinte, é atribuída a uma variável ao total dos salários de um determinado departamento, a seguir num comando IF...Else verifica-se se a soma é superior a 130000 unidades, imprimindo mensagens distintas para cada caso:

```
DECLARE @Valor int
SELECT @Valor = (select sum(salario) FROM emp
WHERE depnum = 10)
select @valor
IF @Valor > 130000
    SELECT 'Departamento com peso excessivo nas contas' AS 'IF'
ELSE
    SELECT 'Departamento sem peso excessivo nas contas' AS 'IF'
```



Caso exista mais de um comando a ser executado numa determinada condição, podemos usar um bloco Begin...end.

Controle de Fluxo – CASE

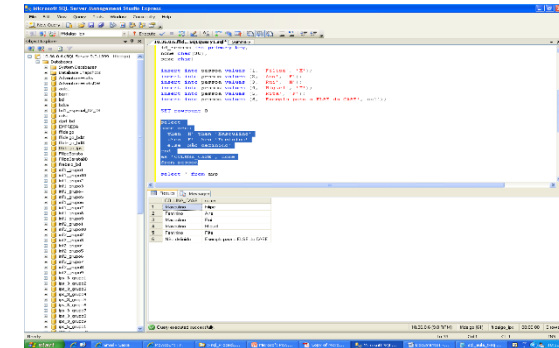
O comando Case permite-nos substituir determinados valores em função de uma lista de condições.

Permite ainda especificar um valor a ser retornado se nenhuma condição for verdadeira.

É bastante útil para tornar a saída mais legível para o utilizador final.

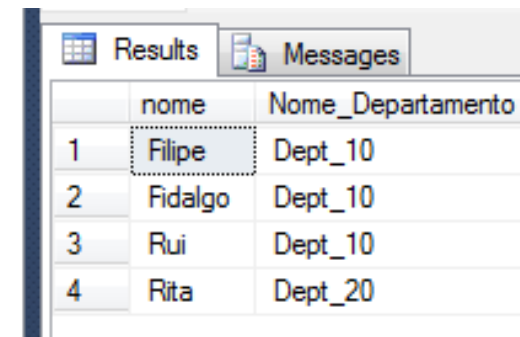
Por exemplo, se existir um campo Sexo numa tabela onde o armazenamento tiver sido feito com os valores char 'M' e 'F', .., estes podem facilmente ser substituídos por Masculino ou Feminino numa consulta.

```
select  
case sexo  
  when 'M' then 'Masculino'  
  when 'F' then 'Feminino'  
  else 'Não definido'  
end  
as 'COLUNA_CASE', nome  
from pessoa
```



Exercício:

- Mostre o nome do empregado juntamente com a informação:
caso trabalhe no Dept:10, deverá ser “DEPT_10”
caso trabalhe no Dept:20, deverá ser “DEPT_20” ...

A screenshot of a SQL query results window. The window has two tabs: 'Results' and 'Messages'. The 'Results' tab is active, showing a table with two columns: 'nome' and 'Nome_Departamento'. The table contains four rows of data.

	nome	Nome_Departamento
1	Filipe	Dept_10
2	Fidalgo	Dept_10
3	Rui	Dept_10
4	Rita	Dept_20

Controle de Fluxo – CASE

Outro exemplo, usando uma comparação na clausula do when

**select nome, salario,
case**

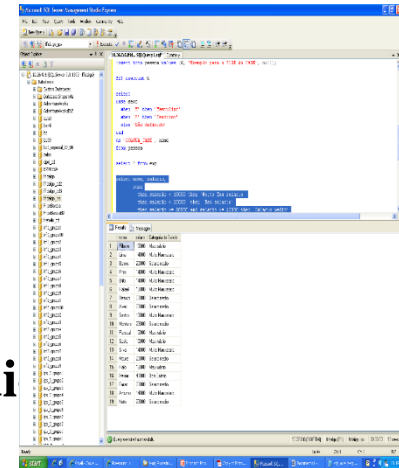
when salario < 150 then 'Muito Mau salario'

when salario < 250 then 'Mau salario'

when salario >= 250 and salario <= 300 then 'Salario medi

else 'Bom Salario' end 'Categoria do Salario'

from emp



Exercício:

- 1) Altere o exemplo anterior, indicando que nos casos “Muito Mau Salario”, verifique se trabalha no Dept: 10.

Nesses caso a mensagem a mostrar deverá ser:

“Muito Mau salario - Trabalha no Dept_10”

Caso não trabalhe, a mensagem deverá ser a mesma.

Results		Messages	
	nome	salario	Categoria do Salario
1	Filipe	100	Muito Mau salario - Trabalha no Dept_10
2	Fidalgo	200	Mau salario
3	Rui	300	Salario medio
4	Rita	1000	Bom Salario

```
select nome, salario,
       case
         when salario < 150 then

           case
             when dept=10 then
               'Muito Mau salario - Trabalha no Dept_10'
             else
               'Muito Mau salario'
           end
         when salario < 250 then 'Mau salario'
         when salario >= 250 and salario <= 300 then 'Salario medio'
         else 'Bom Salario' end 'Categoria do Salario'
from emp
```

Controle de Fluxo – While

O While é um ciclo de repetição com a seguinte sintaxe:

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

Break e continue, para e reiniciam o ciclo, respectivamente.
O ciclo while normalmente é utilizado dentro de cursores.

Vamos então ver primeiro o uso de cursores.

Cursores

Os Cursores são utilizados quando precisamos tratar cada linha de uma dada consulta de forma individual.

Para isso utilizamos Cursores, que não é nada mais que um select em que o resultado é retornado linha a linha.

1º declarar cursor

```
declare @Mostra_Salario_Empregado Cursor  
for select Nome, salario from emp
```

De notar que precisamos também de definir as variáveis de saída

```
declare @Nome char(50)  
declare @Salario int
```

Cursores

2º Abrir o cursor

Open @Mostra_Salario_Empregado

Neste momento a consulta já foi feita e está armazenada, vamos agora analisar a primeira linha do cursor:

3º Analisando registos do cursor:

Vamos usar as duas variáveis que foram criadas.

```
fetch next from @Mostra_Salario_Empregado  
into @Nome, @Salario /*A ordem é importante*/
```

Já temos o primeiro registo vamos construir um ciclo e tratar cada registo de forma individual.

Cursorres

while (@ @fetch_status = 0) /* @ @Fetch_Status é uma variável global do SQL server que indica se um cursor retornou algum registro após o fetch*/

Begin

/*Aqui fazemos os tratamentos necessários*/

if exists (...)

/*Depois temos de saltar para o proximo registro*/

fetch next from @Mostra_Salario_Empregado into @Nome,

@Salario

end

end /*While*/

Cursorres

`/*Finalizar e desalocar a memória associada ao cursor*/`

```
close @Mostra_Salario_Empregado  
deallocate @Mostra_Salario_Empregado
```

Os cursores são muito úteis, mas seu uso deve ser evitado, já que eles consomem muitos recursos.
Devemos usá-los apenas se for estritamente necessário.

Controle de Fluxo – While + Cursores

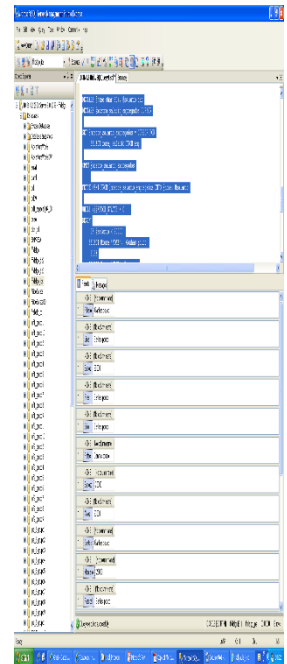
```
DECLARE @nome char(50), @salario int
DECLARE @mostra_salario_empregados CURSOR
```

```
SET @mostra_salario_empregados = CURSOR FOR
SELECT nome, salario FROM emp
```

OPEN @mostra_salario_empregados

FETCH NEXT FROM @mostra_salario_empregados INTO
@salario

```
WHILE (@@FETCH_STATUS = 0)
BEGIN
    IF @salario < 20000
        SELECT @nome, 'Ganhas pouco'
    ELSE
        SELECT @nome, @salario
```



Exercícios - Cursores

1- Crie um cursor para a tabela “EMP”. Leia desse cursor o nome e a função, bem como o departamento onde trabalha.
Para os empregados que trabalham no departamento 10, imprima os nomes e a função, juntamente com o texto “trabalha no departamento Consultoria”.
Caso contrário, basta imprimir o nome e a função.

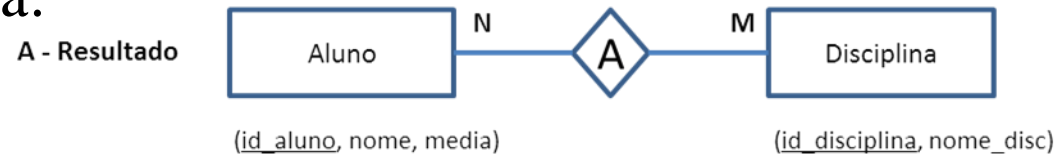
2- Crie um cursor para a tabela “EMP”. Leia desse cursor o numero de empregado, salario e o departamento onde trabalha.
Para os empregados que:

- trabalham no departamento 10, aumente o salário em 10%;
- trabalham no departamento 20, aumente o salário em 20%;
- trabalham no departamento 30, aumente o salário em 30%;
- trabalham no departamento 40, aumente o salário em 40%;

Construa também um bloco em T-Sql que faça o inverso.

Exercícios - Cursores

- 1- Crie um bloco em T-Sql que imprima a tabuada do 5.
- 2 – Crie um bloco em T-Sql que calcule o fatorial de um dado valor.
- 3- Considere a seguinte estrutura:



E a tabela: create table certificado (coluna varchar(500));
Escreva um bloco em T-Sql, que para um dado identificador de um aluno (id_aluno), escrever na tabela “Certificado” a seguinte informação:

- o nome do aluno;
- o nome da disciplina, a data da realização do exame escrito e a nota, para cada exame realizado pelo aluno;
- a media do aluno;
- data em que esse certificado foi criado.

Ex:

Filipe Fidalgo

Bases de Dados I 01-Jan-2010 15
Base de Dados II 02-Fev-2010 16

Media: 16 valores
Criado em: 15-Dez-2010

Gerar Erros

O comando **Raiserror** permite gerar um erro definido pelo utilizador. Recebe os seguintes argumentos:

- **msg_id | msg_str**: Permite que seja passado um ID do erro previamente gravado na tabela de sistema *sysmessages* (veremos em seguida como adicionar um erro personalizado) ou ainda uma *String* informativa sobre o erro. Caso se utilize uma string, o id do erro será 50000.
- **Severity**: Informa sobre a gravidade do erro:
 - um valor de 0 a 18 pode ser utilizado por qualquer utilizador;
 - de 19 a 25 apenas por membros do papel *Sysadmin*;
 - Ao usarmos uma gravidade acima de 20, a ligação é encerrada.
- **State**: Utilizado para indicar a origem do erro, pode assumir um valor entre 1 e 127.
- **WITH**: Permite definir algumas opções:
 - Log: grava o erro no log do sistema. Esta opção é obrigatória com uma gravidade acima de 20;
 - NOWAIT: Retorna a mensagem imediatamente para o usuário;
 - SETERROR: define o valor da variável global @@error com o ID do erro ou com 50000, independente da gravidade do erro.

Gerar Erros

Neste exemplo, é gerado um erro simples de baixa gravidade:

```
Raiserror('Este erro é de baixa gravidade',15,1)
```

Um erro com gravidade 25 e gravação em log:

```
Raiserror('Este erro é grave',25,1) WITH LOG
```

Gerar Erros - Personalizados

Usando o procedure de sistema *sp_addmessage* podemos definir nossos próprios erros.

Este procedimento recebe os seguintes argumentos:

- **msg_id**: Um valor inteiro maior que 50000 e menor que 2147483647. Deve ser único, caso já exista uma mensagem com o id especificado o erro não será criado;
- **Severity**: Um valor entre 1 e 25 conforme a descrição da secção anterior;
- **Msg**: É a mensagem de erro, pode ter até 255 caracteres;
- **Language**: Idioma utilizado, quando omitido utiliza o idioma da secção;
- **with_log**: determina se o erro vai ser gravado no log do sistema operativo;
- **Replace**: Determina se o erro deve substituir um erro já existente.

Exemplo de criação de um erro:

```
EXEC sp_addmessage 50001,16,'Esta é uma nova mensagem de erro',null,False,replace
```


Funções

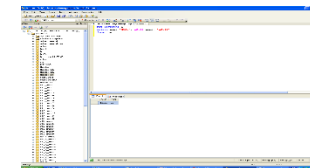
O SQL Server possui dezenas de funções internas, divididas em mais de 10 categorias.

Uma função pode ter a finalidade mais variada: Tratar uma string, retornar algum valor da configuração do SGDB, manipular datas etc. Vamos ver algumas das principais funções do SQL Server nas principais categorias, acompanhados de exemplos e dos respectivos outputs.

Funções de Tratamento de String

ASCII: Retorna o código ASCII do caracter mais à esquerda.

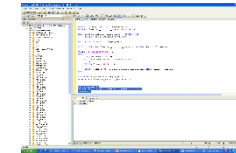
```
set rowcount 1  
select nome 'NOME', ASCII(nome) 'ASCII'  
from emp
```



Funções - String

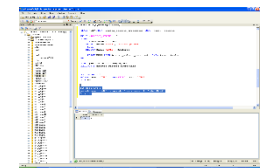
CHAR: Retorna o caracter equivalente ao código ASCII passado como parâmetro:

```
set rowcount 1  
select nome 'NOME', char(ASCII(nome)) 'CHAR'  
from emp
```



CHARINDEX: Retorna a posição Inicial de uma String dentro de outra. Recebe três parâmetros: os caracteres que são procurados, a string onde a busca será realizada, e a posição inicial da busca:

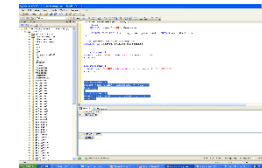
```
set rowcount 1  
select nome 'NOME', charindex('ei', nome, 1) 'CHARINDEX'  
from emp
```



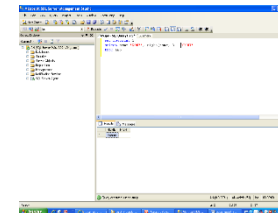
Funções - String

LEFT e RIGHT: Retornam a parte esquerda e direita, respectivamente, de uma string. O segundo argumento é o número de caracteres que devem ser retornados:

```
set rowcount 1  
select nome 'NOME', left(nome, 3) 'LEFT'  
from emp
```



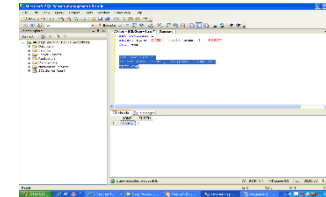
```
set rowcount 1  
select nome 'NOME', right(nome, 3) 'RIGHT'  
from emp
```



Funções - String

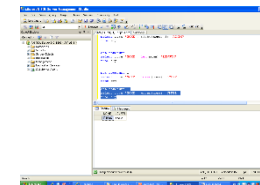
- **LEN:** Retorna o número de caracteres de uma string:

```
set rowcount 1  
select nome 'NOME', len(nome) 'LENGTH'  
from emp
```

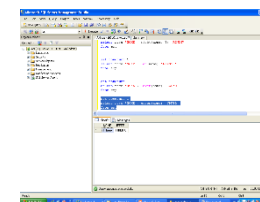


- **LOWER e UPPER:** Converte uma string para letras minúsculas e maiúsculas respectivamente:

```
set rowcount 1  
select nome 'NOME', lower(nome) 'LOWER'  
from emp
```

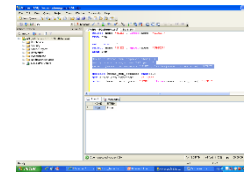


```
set rowcount 1  
select nome 'NOME', upper(nome) 'UPPER'  
from emp
```

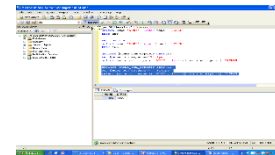


Funções - String

- LTRIM E RTRIM; Removem espaços em branco a esquerda e a direita de uma string, respectivamente.



```
declare @nome_com_espacos char(30)
set @nome_com_espacos='Filipe '
select @nome_com_espacos 'NOME', ltrim(@nome_com_espacos) 'RTRIM'
```

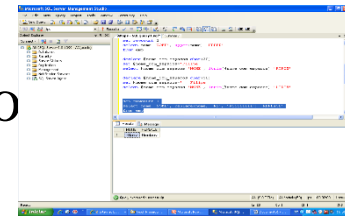


```
declare @nome_com_espacos char(30)
set @nome_com_espacos=' Filipe'
select @nome_com_espacos 'NOME', ltrim(@nome_com_espacos) 'LTRIM'
```

Funções - String

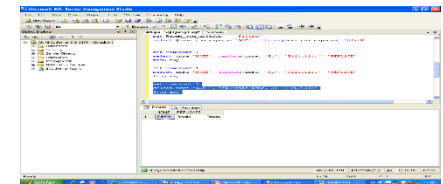
· **REPLACE**: Substitui todas as ocorrências de uma dada String numa outra. Recebe três parâmetros:

- a string onde a procura deverá ser realizada;
- a string que se quer encontrar;
- e a string que deve ser utilizada na substituição



```
set rowcount 1
select nome 'NOME', replace(nome, 'Ri', 'Riiiiiii') 'REPLACE'
from emp
```

· **REPLICATE**: Repete uma string tantas vezes quantas as definidas por um parâmetro de entrada:

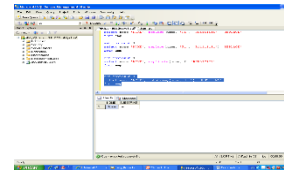


```
set rowcount 1
select nome 'NOME', replicate(nome, 2) 'REPLICATE'
from emp
```

Funções - String

- SUBSTRING: Retorna parte de uma string. Recebe três parâmetros:
 - A string da qual se quer retornar parte;
 - a posição inicial;
 - e o número de caracteres.

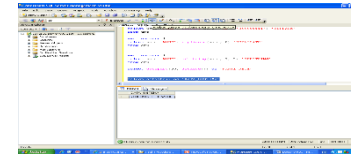
```
set rowcount 1  
select nome 'NOME', substring(nome, 2, 2) 'SUBSTRING'  
from emp
```



Funções – Data e Hora

A função que nos dá a data do sistema é `getdate()`:

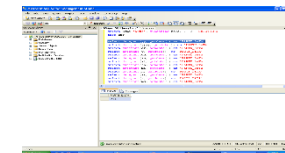
```
select getdate() as 'DATA_SISTEMA'
```



O tipo `DateTime` pode conter informações sobre o ano, mês, dia, hora, minutos etc. A função `DatePart` retorna parte de uma data passada como parâmetro. O parâmetro indica a parte da data desejada. Observe a tabela a seguir:

Parte Data/Hora	Representação	Abreviação
Ano	Year	YY, YYYY
Mês	Month	m
Dia no Ano	dayofyear	dy, y
Dia no Mês	Day	dd, d
Semana no Ano	Week	wk, ww
Hora	Hour	Hh
Minuto	minute	mi, n
Segundo	second	ss, s
Milissegundo	millisecond	ms

```
select datepart(yy, getdate()) as 'PARTE_DATA'
```



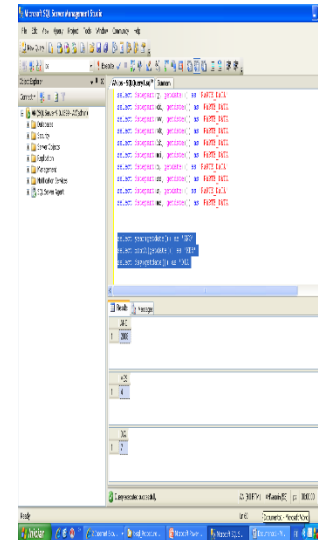
Funções – Data e Hora

A função Getdate() pode ainda ser utilizada em conjunto com outras funções. O SQL Server possui ainda as funções MONTH, YEAR, DAY, que retornam, respectivamente o mês, ano e dia de uma Data. Podemos ter um resultado equivalente a instrução anterior utilizando getdate() em conjunto com a função YEAR:

`select year(getdate()) as 'ANO'`

`select month(getdate()) as 'MES'`

`select day(getdate()) as 'DIA'`



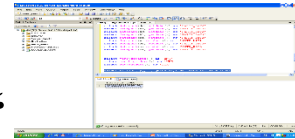
Funções – Data e Hora

DATEADD: Adiciona um intervalo de tempo, passado como parâmetro, à data de sistema. Recebe três argumentos:

- A parte da data a ser incrementada (DatePart),
- o incremento;
- e a data alvo.

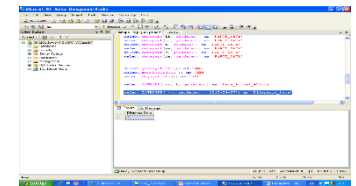
No exemplo seguinte são adicionados dois anos à data actual:

```
select DATEADD( yy, 2, getdate()) as 'Data_Actual_+2Anos'
```



DATEDIFF: Retorna a diferença entre duas datas. Recebe três argumentos:

- a parte da data (DatePart);
- a data inicial
- e a data final:

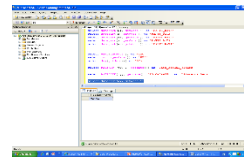


```
select DATEDIFF( yy, getdate(), '2010-04-07') as 'Diferenca_Data'
```

Funções – Data e Hora

DATENAME: Retorna uma descrição textual da data passada como parâmetro.

```
select datename(dw, getdate())
```

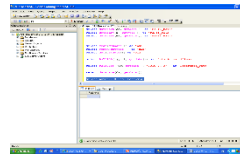


Funções – Matemáticas

Entre as diversas funções matemáticas existentes no SQL Server, vamos também ver apenas as mais importantes:

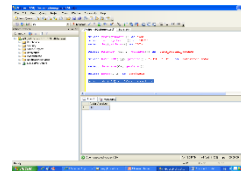
POWER: Retorna um número, primeiro parâmetro, elevado a potência do valor passado como segundo parâmetro:

`select power(2,3) as 'Potência'`



ABS: Retorna o valor absoluto de uma expressão:

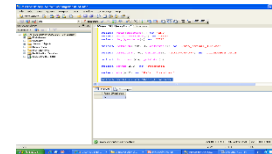
`select abs(-10) as 'Valor_Absoluto'`



Funções – Matemáticas

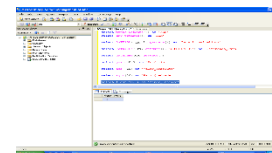
SQRT: Retorna a raiz quadrada de uma expressão:

```
select sqrt(36) as 'Raiz_Quadrada'
```



FLOOR: Retorna o menor valor inteiro, do valor passado como parâmetro:

```
select floor(10.6) as 'Menor_Inteiro'
```

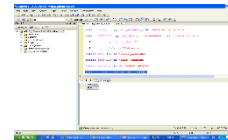


Funções – Sistema

São funções utilizadas para retornar informações diversas sobre o Servidor.

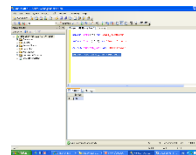
CURRENT_USER: Retorna o usuário corrente da secção.

`select current_user as 'Utilizador'`



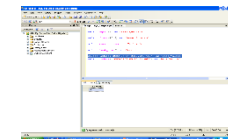
HOST_NAME () : Retorna o nome do servidor:

`select host_name() as 'Servidor'`



· **ISDATE:** Retorna 1 caso uma expressão passada como parâmetro seja uma data válida, caso contrário retorna zero:

`select isdate('2008-04-07 00:27:31:460') as 'Data_Valida'`



Conceitos Stored

- Para além dos objectos já conhecidos, é possível um utilizador guardar (stored) outros elementos, por exemplo procedimentos, funções, ...
- É sobre estes que vamos ver alguns exemplos de utilização.

Tipos de User Define Funções

O SQL Server implementa três tipos de UDFs (User Define Function):

- Scalar valued functions
- Inline table valued functions
- Multi-statement table valued functions

Algumas outras observações a respeito de funções:

- Uma UDF não pode fazer qualquer alteração nos objectos do servidor (por exemplo numa tabela);
- Uma UDF não pode chamar uma função não determinística. Uma função não determinística é qualquer função que retorne um valor diferente a cada chamada, por exemplo: GETDATE() ;
- A chamada de uma função escalar deve ser qualificada com pelo menos o nome do proprietário (owner).

UDF'S

Funções Scalar valued

Uma função definida pelo utilizador do tipo *scalar valued* retorna um único valor como resposta a cada uma das chamadas à função. Uma função é do tipo *scalar valued* se a cláusula de retorno especificar um tipo de dado *scalar* do SQL Server. Por exemplo:

```
CREATE FUNCTION Volume
(@Altura decimal (4,1), @Largura decimal (4,1), @Profund decimal
(4,1))
RETURNS decimal (12,3) -- tipo de dado a retornar.
AS
BEGIN
    RETURN (@Altura * @Largura * @Profund)
END
```

UDF'S

//EXECUTAR FUNÇÃO

Select dbo.Volume(12.2,10.6,10.0) as VOLUME

Select [ffidalgo].[dbo].[Volume](12.2,10.6,10.0)

//OUTRA FORMA DE EXECUTAR FUNÇÃO

DECLARE @returnVolume decimal (12,3);

EXEC @returnVolume = dbo.volume 2.1, 3.2, 4.3;

PRINT @returnVolume;

UDF'S

Funções Table Valued

Para funções do tipo *inline table valued*, a cláusula RETURNS é seguida da palavra TABLE sem a lista de colunas. As funções do tipo *inline table valued* retornam valores apresentados como tabelas e são definidas com uma única instrução SELECT. As colunas, incluindo os tipos de dados retornados pela função, são derivados da lista da instrução SELECT que define a função. Por exemplo:

```
CREATE FUNCTION fn_emp_no_Dep (@dep int)
RETURNS TABLE
AS
RETURN (
    SELECT empnum, nome, depnum
    FROM emp
    WHERE depnum = @dep
)
```

UDF'S

Funções Multi-Statement Table Valued

Se a cláusula RETURNS especificar uma tabela indicando as colunas e tipos de dados, a função será do tipo *multi-statement table valued*. Por exemplo:

```
CREATE FUNCTION fn_lista_Emp_dep_ord_nome (@dep int)
```

```
RETURNS @EmpOrdDep TABLE
```

```
(
```

```
empnum int,
```

```
nome char (20),
```

```
dep int,
```

```
Data_adm datetime
```

```
)
```

```
AS
```

```
BEGIN
```

```
INSERT @EmpOrdDep
```

```
SELECT empnum, nome, depnum, data_adm
```

```
FROM emp
```

```
where depnum=@dep
```

Procedimentos

Existem blocos modulares de código, précompilados e armazenados no servidor como parte de uma BD, conhecidos como stored procedures e triggers.

Cada um destes blocos pode conter instruções de controle de fluxo, comandos DML ou simples consultas. Algumas das vantagens da sua utilização:

Centralização: Qualquer mudança nas regras de negócio implicaria uma actualização em dezenas, às vezes centenas de binários espalhados pela empresa, pela cidade ou pelo mundo, desta forma basta fazer uma única actualização ao nível do servidor;

Segurança: A tarefa do DBA fica facilitada, ao invés de ter que controlar diferentes níveis de privilégios sobre os diferentes objectos, pode definir a segurança a partir de stored procedures;

Trafico de rede: Os stored procedure ou os trigger são armazenados ao nível do servidor, ao cliente basta executá-los, implícita ou explicitamente, com eventuais parâmetros, e receber de volta algum possível retorno.

Clientes magros (Thin Client): Como as regras de negócio estão no servidor, as aplicações podem ser menores, exigindo menos recursos do sistema (disco, memória e CPU) e consequentemente hardware de menor custo;

As consultas embutidas dentro de stored procedures podem melhorar o desempenho das consultas repetitivas, já que o SGBD possui o seu plano de execução pré-compilado.

Procedimentos

Os triggers, e os stored procedures são blocos de código armazenados no servidor. Mas enquanto os triggers são executadas implicitamente (pois estão associadas a um comando DML executado sobre uma tabela, podemos dizer que os triggers estão associadas a eventos) os stored procedures, ao contrário, são executadas.

Um stored procedure pode ter o objectivo mais variado: Realizar uma simples consulta ou um cálculo complexo sobre uma regra de negócio.

Tipos de Stored procedures

O SQL Server possui pelo menos 4 tipos de stored procedures:

Locais, são procedures dos utilizadores, utilizadas pelas aplicações; **Temporais**, existentes apenas durante uma sessão;

Sistema, criadas pelo Servidor durante a instalação com funções administrativas

e **Estendidos**, chamadas a partir de uma DLL.

Procedimentos - Temporários

Os Stored procedures temporários são armazenados na Base de Dados TempDB (Existem outros objectos com excepção das views ou functions, podem ser criados de forma temporária).

Existem dois tipos de Stored procedures temporárias:

Locais: Visíveis apenas na sessão corrente, ou seja, para o utilizador que a criou. Este tipo de procedure é excluído automaticamente no fecho da sessão. Para criar um procedure temporário local deve-se adicionar o prefixo # ao nome.

Globais: Visíveis para todas as sessões activas. Este tipo de procedure também é excluído automaticamente no fim da sessão em que foi criado, a não ser que esteja a ser utilizado por outra sessão no momento da exclusão, neste caso, a exclusão ocorrerá ao fim do comando T-SQL que lhe o referenciou. Para criar um procedure temporária global deve-se adicionar o prefixo ## ao nome.

Procedimentos - Temporários

Exemplo da criação de um procedimento temporário local:

```
DROP PROCEDURE LISTA_EMPREGADOS
```

```
CREATE PROCEDURE #LISTA_EMPREGADOS
```

```
AS
```

```
SELECT * FROM emp
```

```
exec #lista_empregados
```


Procedimentos - Temporários

Exemplo da criação de um procedimento temporário global:

```
DROP PROCEDURE LISTA_EMPREGADOS_DEP_10
```

```
CREATE PROCEDURE ##LISTA_EMPREGADOS_DEP_10  
AS  
SELECT * FROM emp  
Where deptnum=10
```

```
exec ##lista_empregados_dep_10
```

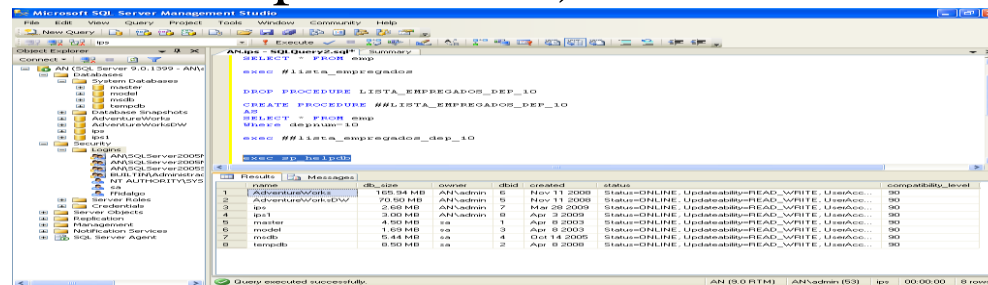
Procedimentos - Sistema

Um aspecto importantíssimo do SQL Server são os System stored procedures (procedures armazenadas de sistema).

Estes procedures são criadas durante a instalação do SQL Server na base de dados Master. A sua função é facilitar a realização de rotinas administrativas, bem como retornar informações sobre o sistema e\ou seus objectos.

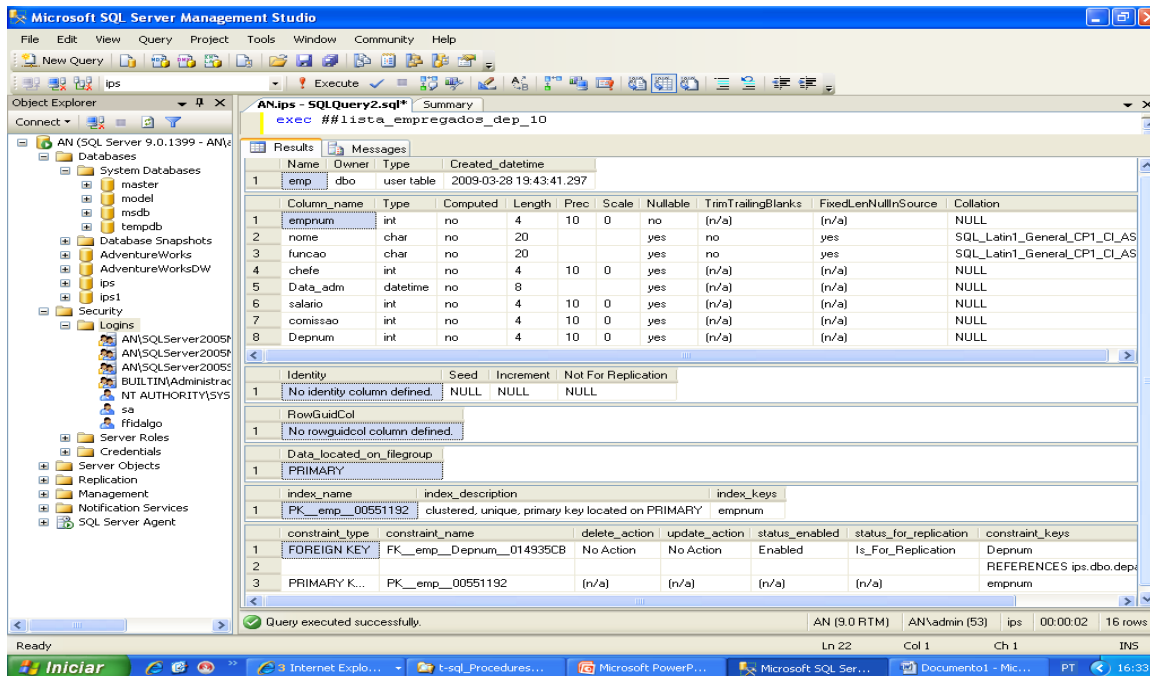
Um System stored procedures tem o prefixo “SP_” por convenção. O SQL Server possui uma grande quantidade de System stored procedures (muitas sem estarem documentadas), por isso vamos estudar algumas a título de exemplo:

sp_helpdb: Mostra informações sobre todos as bases de dados do servidor (se executada sem parâmetros) ou de uma base de dados específica



Procedimentos - Sistema

sp_help: Traz informações sobre um objecto de uma base de dados, como por exemplo, uma tabela



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The left pane shows the 'Object Explorer' with the 'ips' database selected. The right pane shows the 'Results' tab for the query 'exec ##lista_empregados_dep_10'. The query results are displayed in a table format, showing the structure of the 'emp' table.

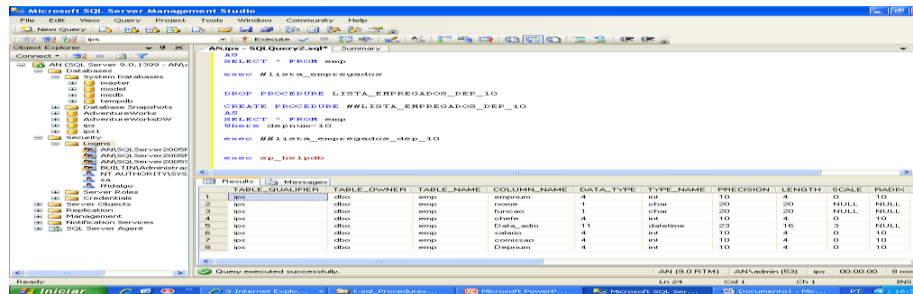
Name	Owner	Type	Created_datetime
emp	dbo	user table	2009-03-28 19:43:41.297

Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
empnum	int	no	4	10	0	no	(n/a)	(n/a)	NULL
nome	char	no	20			yes	no	yes	SQL_Latin1_General_CP1_CI_AS
funcao	char	no	20			yes	no	yes	SQL_Latin1_General_CP1_CI_AS
chefe	int	no	4	10	0	yes	(n/a)	(n/a)	NULL
Data_adm	datetime	no	8			yes	(n/a)	(n/a)	NULL
salario	int	no	4	10	0	yes	(n/a)	(n/a)	NULL
comissao	int	no	4	10	0	yes	(n/a)	(n/a)	NULL
Depnum	int	no	4	10	0	yes	(n/a)	(n/a)	NULL

The bottom section of the results pane shows the table's constraints and indexes. The 'PRIMARY' index is located on the 'empnum' column. There is also a 'FOREIGN KEY' constraint named 'FK_emp_Depnum_014935CB' that references the 'Depnum' column in the 'ips.dbo.dep' table.

Procedimentos - Sistema

sp_columns: Mostra informações sobre colunas de uma tabela

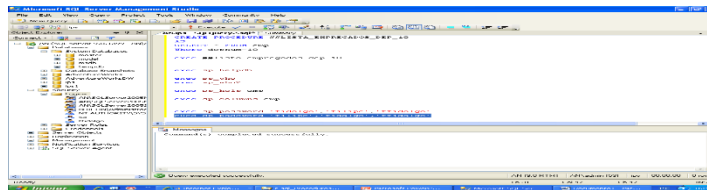


sp_who: Mostra informações sobre utilizadores e processos do servidor. Opcionalmente pode ser passado o nome do utilizador como parametro. **Sp_who2** (não documentada) mostra mais informações

Procedimentos - Sistema

sp_password: Adiciona ou altera a password de um login.
Os argumentos são: password actual, nova password e login.

`exec sp_password 'filipe','fidalgo','ffidalgo'`



Procedimentos - Estendidos

Extended stored procedures são DLL's criadas por linguagem de programação, que podem ser executadas dinamicamente pelo SQL Server.

Normalmente são desenvolvidas para executar tarefas que o SQL Server não pode fazer. Como convenção, possuem o prefixo “XP_” no nome.

Durante a instalação do SQL Server são criados diversos extended procedures.

Posteriormente podem ser adicionados novos através da chamada da procedure de sistema sp_addextendedproc.

Para o utilizador, um Extended stored procedures é tratado da mesma forma que os outros stored procedure, tanto na execução como na passagem de parâmetros.

Vamos ver um exemplo simples de utilização de um extended procedure.

Xp_cmdshell permite executar uma instrução na linha de comando do msdos, a partir do SQL Server, obtendo a mesma saída.

O código seguinte altera data do Sistema Operativo a partir de um comando executado no SQL Server:

```
declare @comando varchar(30)
set @comando='date'+char(13)+'8/4/2008'
exec master..xp_cmdshell @comando
```

Procedimentos - Utilizador

Normalmente este é o tipo de procedimento que mais interessa ao desenvolvimento de software. A sintaxe de criação de um stored procedure é:

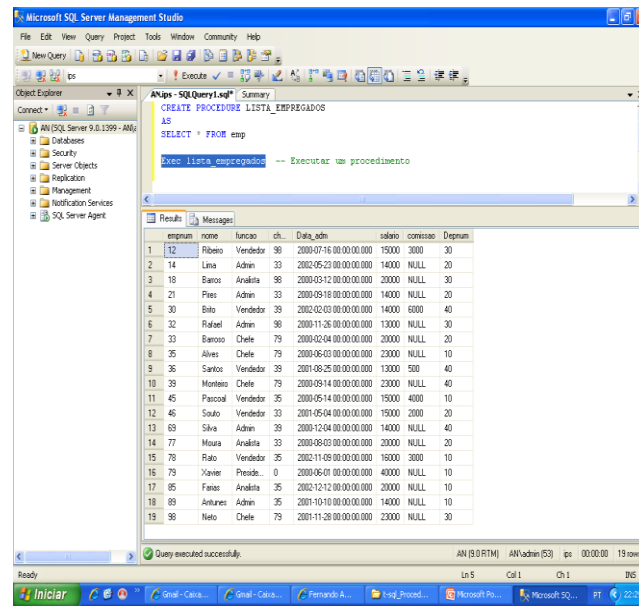
```
CREATE PROC [ EDURE ] procedure_name [ { @parameter data_type  
} [ VARYING ] [ = default ] [ OUTPUT ] ] [ ,...n ]  
[ WITH { RECOMPILE | ENCRYPTION [ FOR REPLICATION ] AS  
sql_statement [ ...n ]  
[ ; number ] | RECOMPILE , ENCRYPTION } ]
```

Procedimentos - Utilizador

Vamos ver um exemplo simples que mostra a listagem referente a uma tabela:

```
CREATE PROCEDURE LISTA_EMPREGADOS
AS
SELECT * FROM emp
```

Exec lista_empregados -- Executar um procedimento



Procedimentos - Utilizador

É boa prática, para evitar a geração de erros, verificar antes da criação de um procedimento, se já existe na base de dados.

Para tal realizamos uma pesquisa na tabela sysobjects com nome Lista_Empregados e tipo P (Stored procedure), em caso afirmativo o procedure é excluído antes da criação

```
IF EXISTS (Select [name] from sysobjects Where name='Lista_Empregados' and  
Type='P')
```

```
drop procedure Lista_Empregados
```

```
GO
```

```
CREATE PROCEDURE LISTA_EMPREGADOS
```

```
AS
```

```
SELECT * FROM emp
```

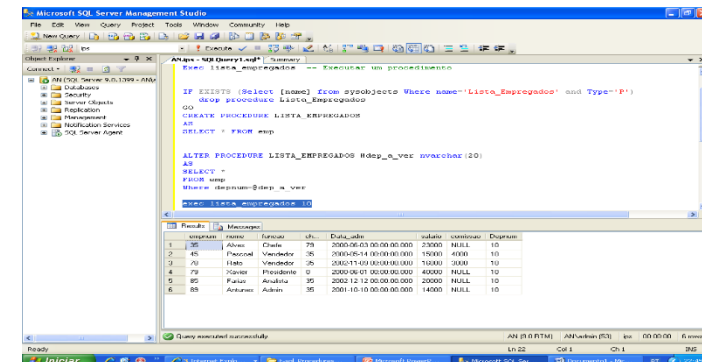
Procedimentos - Utilizador

Vamos agora proceder a algumas alterações ao procedimento, com a finalidade de mostra que estes podem ser alterados depois de criados e podem também, receber parâmetros de entrada (Se receber mais de um, terão de ser separados por uma virgula).

```
ALTER PROCEDURE LISTA_EMPREGADOS @dep_a_ver int
AS
SELECT *
FROM emp
Where depnum=@dep_a_ver
```

Podemos executar os procedimentos de várias formas:

- Indicando os parâmetros por ordem:
exec lista_empregados 10
- Indicando os parâmetros com referência ao nome:
exec lista_empregados @dep_a_ver=10



Procedimentos - Utilizador

Parâmetros Default

Se tentarmos executar o procedimento anterior sem indicarmos o parâmetro de entrada, obtemos um erro, pois este não pode ser omitidos, a não ser que possua um valor Default.

Um valor default pode ser indicado logo após a declaração do parâmetro.

Vamos alterar o procedimento para ter 20 como departamento por Default:

```
ALTER PROCEDURE LISTA_EMPREGADOS @dep_a_ver int = 20  
AS  
SELECT *  
FROM emp  
Where depnum=@dep_a_ver
```

Procedimentos - Utilizador

Argumento With Encryption

A tabela de sistema syscomments possui o texto de todas os Procedimentos e Triggers (e de diversos outros objectos). Cada base de dados possui uma tabela syscomments.

Através desta tabela podemos recuperar o texto usado na criação do objecto (Como veremos mais adiante). Caso haja a necessidade de ocultar o texto do procedimento ou trigger, pode usar-se o argumento With Recompile no momento da criação do objecto. Isto criptografa o texto do objecto, tornando-o ilegível.

Nem mesmo o administrador da base de dados consegue recuperar texto de um objecto criptografado. With Encryption é uma excelente alternativa de ocultar uma regra de negócio da aplicação.

Procedimentos - Utilizador

Para criptografar um procedimento basta usar o argumento with encryption após o nome do procedimento:

IF EXISTS (Select [name] from sysobjects Where name='Lista_Empregados' and Type='P')

drop procedure Lista_Empregados

GO

CREATE PROCEDURE LISTA_EMPREGADOS with Encryption

AS

SELECT * FROM emp

Argumentos For Replication e NOT FOR REPLICATION

O argumento For Replication especifica que um procedimento não pode ser executado durante a replicação. Já Not For Replication indica que um trigger não deverá alterar a tabela à qual esta associado.

Procedimentos - Utilizador

Argumento With Recompile

O SQL Server cria um plano de execução para qualquer bloco de T-SQL antes de sua execução. O plano de execução fica armazenado num local especial de memória denominado procedure Cache. A pré-existência de um plano de execução provoca um ganho em performance significativo. Num procedimento, o plano de execução é compilado no momento da criação. Este plano será re-compilado, para fins de optimização, sempre que o procedimento sofrer alguma alteração (com ALTER PROCEDURE) ou que um objecto utilizado por ele, como uma tabela, for alterado.

Podemos forçar a criação de um novo plano de execução de um procedimento se o este for criado ou executado com a opção With Recompile. Criar um procedimento com a opção With Recompile força a criação de um novo plano de execução cada vez que é executado. O uso do procedimento de sistema sp_recompile, provoca a re-compilação do plano de execução na próxima execução.

Procedimentos - Utilizador

Recursividade e Aninhamento

O aninhamento ocorre quando um procedimento ou triggers provoca a execução de outro procedimento ou triggers na sua zona de execução.

O número máximo de aninhamentos permitidos é de 32.

A recursividade é um tipo de aninhamento, ocorre quando um objecto se chama a si mesmo. Existem dois tipos de recursividade, a Directa e a Indirecta. Na directa o objecto chama-se a si mesmo na própria zona de execução. Na indirecta o objecto é chamado por outro objecto que não ele próprio. Para permitir recursividade Directa de triggers, a opção de configuração `RECURSIVE_TRIGGERS` da base de dados deve estar activa. Esta opção não tem qualquer influência na recursividade Indirecta.

Procedimentos - Utilizador

Execução Automática

Um procedimento pode ser executado de forma automática sempre que o SQL Server for iniciado. Este procedimento não pode ter parâmetros de entrada, deve existir na base de dados Master e ser criada por um membro com o papel sysadmin.

Para inicializar ou cancelar a inicialização automática, deve usar-se o procedimento de sistema `sp_procoption`, que recebe três argumentos:

- Nome do procedimento;
- opção que é sempre `startup`
- e valor.

O código seguinte configura um suposto procedimento de nome teste para inicialização automática:

```
EXEC sp_procoption 'teste',startup,true
```

Para desabilitar a execução automática, basta alterar o último argumento para `False`:

```
EXEC sp_procoption 'teste',startup, False
```

Embora um procedimento de execução automática não possa receber parâmetros, o utilizador pode criar um procedimento que execute um ou mais procedimentos passando os parâmetros necessários.

Procedimentos - Utilizador

Agrupamento

Os procedimentos criadas com o mesmo nome são chamados procedimentos Agrupados. Agrupar procedimentos pode ser útil para fins de organização lógica do código. Um procedimento agrupado recebe um número, a única regra é que o primeiro deve ter o número 1:

```
CREATE PROCEDURE Nome;1
```

```
CREATE PROCEDURE Nome;10
```

A exclusão de qualquer procedimento do grupo provoca a exclusão de todos os outros.

Alterando um Procedimento

A sintaxe para alterar um procedimento é basicamente a mesma utilizada para a criação, substituindo-se a palavra chave Create por Alter.

Excluindo uma ou mais Procedimentos

Para remover uma ou mais procedimentos utilizamos o comando Drop procedure, é possível excluir diversos procedimentos simultaneamente, para isto basta indicar os nomes separadas por virgulas.

Procedimentos - Utilizador

Podemos verificar a existência de um procedimento antes de o tentar excluir, o que evita uma mensagem de erro caso o objecto já não exista na base de Dados:

```
IF EXISTS (Select [name] from sysobjects Where  
name='Lista_Empregados' and Type='P')
```

```
drop procedure Lista_Empregados
```

Renomeando um Procedimento

Para renomear um procedimento utilizamos o procedimento de sistema `sp_rename`, que recebe dois argumentos: O nome do actual do procedimento, e o novo nome:

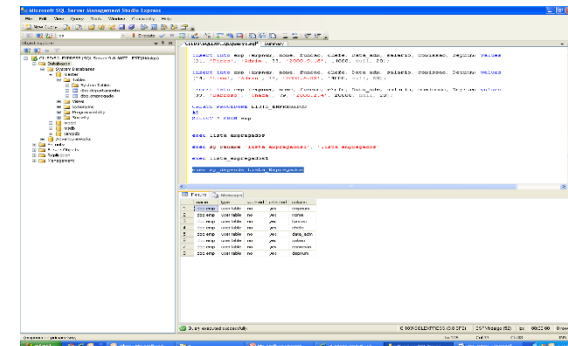
```
exec sp_rename 'lista_empregados', 'lista_empregados1'
```

Procedimientos - Utilizador

Verificando dependências

Antes de excluir um objecto da nossa base de dados, é prática recomendada, verificar quais dependências que este objecto possui. O SQL Serve possui um procedimento de sistema com este objectivo, `sp_depends`, que recebe como parâmetro o nome do objecto. Por exemplo, para verificarmos as dependências do nosso procedimento `Lista_Empregados` anteriormente criado, podemos executar:

Exec sp_depends Lista_Empregados

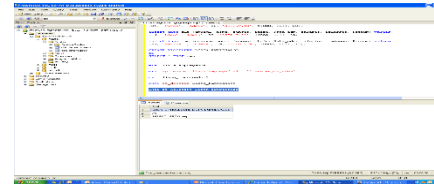


Procedimentos - Utilizador

Obtendo o código fonte de um procedimento

É possível obter o código fonte (DDL) de um procedimento, desde que o mesmo não tenha sido criado criptografado (com a opção **with Encryption**), através do procedimento de Sistema Sp_helptext. Este procedimento recebe como argumento o nome do procedimento do qual desejamos obter o código fonte, por exemplo:

Exec sp_helptext Lista_Empregados



Sp_helptext pode retornar o código não apenas de um procedimento, mas de qualquer objecto armazenado em SYSCOMMENTS, como rules, defaults, UDFs, views e triggers.

Procedimentos - Utilizador

Retorno de dados, parâmetros OUTPUT e RETURN

No SQL Server um procedimento pode retornar valores de 3 maneiras diferentes:

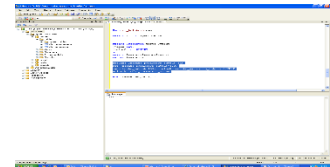
- Um ou mais conjuntos de resultados provenientes de consultas realizada na execução do procedimento;
- Parâmetros OUTPUT: Um parâmetro como os que usamos nos exemplos anteriores é um parâmetro de entrada, é utilizado na execução do procedimento e não retorna nenhum valor. Já um parâmetro de saída (OUTPUT) pode retornar um valor, além de poder ainda ser utilizado para ser passado como um parametro de entrada para outro procedimento. O funcionamento é análogo à passagem de parâmetros por referencia. Definimos um parâmetro de saída utilizando a palavra chave OUTPUT.
- Return: Return é um parâmetro especial, utilizado para retornar um número inteiro. Normalmente é utilizado para tratamento de erros ou para verificar determinadas condições na execução.

Procedimentos - Utilizador

O seguinte procedimento não realiza nada internamente além de uma consulta simples e atribui valores as parâmetros, o único objetivo é facilitar a compreensão das diversas maneiras em que podemos passar parâmetros a um procedimento e obter resultados.

```
create procedure teste_Output  
@valor int,  
@valor2 int OUTPUT  
as  
select @valor2=@valor+@valor2  
return @valor2/2
```

```
declare @recebe_parametro_output int  
set @recebe_parametro_output=150;  
exec teste_output 150, @valor2=@recebe_parametro_output OUTPUT  
print @recebe_parametro_output
```



Exercícios - Procedimento

1- Crie um procedimento que receba como parâmetro de entrada o identificador de um empregado, e mostre o nome do departamento onde trabalha.

2 – Crie um procedimento que receba como parâmetro de entrada um número de departamento, e que mostre quantas pessoas trabalham nesse departamento.

3 - Crie um procedimento que mostre quantas pessoas começaram a trabalhar na empresa no ano “2000”.

...

TRIGGERS

Um trigger (gatilho) é muito semelhante a um procedimento, excepto que ao invés de ser executado explicitamente, o gatilho está associado a um evento disparado através de um comando DML.

Muitas vezes as restrições não são suficientes para manter a integridade dos dados, esta é uma das principais utilizações de um Trigger. Um Trigger pode ser usado para controlo de segurança. Vamos analisar os principais conceitos relacionados com triggers e criar um exemplo onde possamos aplicar todos estes conceitos.

Eliminando mensagens de Retorno

Quando a variável SET NOCOUNT tem o valor ON, elimina a mensagem que reporta o número de linhas afectadas por uma transação. É uma boa prática eliminar estas mensagens em triggers, pois caso contrário, poderá ter diversas mensagens informando que “n” linhas foram afectadas quando na verdade executou um único comando DML.

Para retomar o retorno de mensagens, basta executar SET NOCOUNT para OFF.

TRIGGERS

Tipos de Triggers

O SQL Server permite a criação de dois tipos de triggers:

- **AFTER (Insert, Delete, Update):** O Trigger After é o mesmo que For (Podemos usar For em vez de After), o único tipo disponível até a versão 7 do SQL Server. Este tipo de trigger é executado após o evento.
- **INSTEAD OF (Insert, Delete, Update):** triggers Instead Of são executados por substituição da ação original.

Tabelas inserted e deleted

As tabelas inserted e deleted são criados pelo SQL Server durante a execução de um Trigger, e contêm, respectivamente, os dados inseridos ou excluídos da tabela. São residentes apenas em memória, temporárias, e os seus dados não podem ser alterados.

TRIGGERS

Verificar qual o campo que foi alterado

Num trigger disparado durante uma actualização, é interessante saber se um determinado campo foi alterado, ou ainda, que campos foram alterados.

O SQL Server fornece-nos duas funções com este fim:

- IF UPDATE (nome_da_coluna) para saber se uma determinada coluna sofreu alguma alteração;
- e COLUMNS_UPDATED() para saber quais colunas que foram alteradas.

Ordem de Execução

Podemos ter mais do que um Trigger After associado a um mesmo evento de uma mesma tabela. É possível determinar qual trigger que será disparado primeiro (first) ou em último (last). Os restantes (ordem none) serão todos executados entre o primeiro e o último sem uma ordem específica.

Para determinar a ordem de execução de um trigger usamos o procedimento de Sistema sp_settriggerorder, que recebe três argumentos:

- Nome do trigger,
- ordem
- e evento.

Os Triggers Instead Of não podem ter uma ordem de execução definida.

Número de Registros Afectados

Podemos determinar o número de registros afectados por um trigger através da variável global @@ROWCOUNT.

TRIGGERS

Exemplo de Triggers

Embora o SQL Server ofereça óptimos mecanismos de auditoria, podemos utilizar triggers para rastrear qualquer tipo de alteração na nossa base de dados e criar nosso próprio sistema de auditoria.

O Trigger apresentado como exemplo é simples, mas requer algumas explicações. Suponha que o Director de uma empresa, utiliza um sistema desenvolvido por si, e quer saber quais os utilizadores que têm feito alterações nos salários. É necessário saber quando e de onde (pc) esta alteração foi realizada.

Vamos utilizar como cenário a tabela Empregados.

Esta tabela possui um campo Salario, onde esta armazenado o salario do empregado. Temos que então criar um trigger After update, que será chamado de Audita. Vamos precisar ainda de uma tabela para armazenar a auditoria, o trigger deve verificar se esta tabela existe, e em caso negativo, criá-la. A definição da tabela é a seguinte:

TRIGGERS

```
if not exists (select * from sysobjects where name='Audit_1' and xtype='u')
create table audit_1(
codigo int identity(0,1) not null primary key,
utilizador char(30) not null default current_user,
pc char(30) not null default host_name(),
data datetime not null default getdate(),
empnum int not null,
salario_antigo int not null,
salario_novo int not null)
```

Note-se que os campos Utilizador, PC e Data tem como valor default, respectivamente, as funções Current_user, Host_name() e GetDate(). Current_User retorna o utilizador da sessão, Host_name() o pc e GetDate() a data e hora actual.

Estabelecendo estas funções como Default, os seus valores podem ser omitidos no momento da inserção do registo na tabela.

TRIGGERS

Em seguida temos que obter das tabelas Inserted e Deleted o novo e antigo salario do empregado, respectivamente, e ainda o seu número, para sabermos qual salario que foi alterado, para isso é necessário declarar três variáveis locais: @id, @salarionovo e @salarioantigo:

```
DECLARE @id INT, @salarionovo int, @salarioantigo int
```

Depois atribuímos a estas variáveis os valores das tabelas Inserted e Deleted:

```
SELECT  
@id = (SELECT empnum FROM deleted),  
@salarionovo=(SELECT salario FROM inserted), @salarioantigo=(SELECT salario FROM  
deleted)
```

Finalmente inserimos o registro na tabela de auditoria

```
INSERT INTO Audit_1 (Empnum,Salario_Novo,Salario_Antigo) VALUES  
(@id,@salarionovo,@salarioantigo)
```

Um último detalhe: O trigger vai ser disparado sempre que a tabela empregado for alterada, mas só nos interessa auditar registos que tenham o campo salario alterado, para isso, utilizamos a função if update para realizar esta verificação, e só então executar a auditoria:

```
IF UPDATE(UnitPrice) BEGIN
```

TRIGGERS

O trigger completo ficou assim:

```
CREATE TRIGGER Audita
ON emp
AFTER UPDATE AS
IF UPDATE(salario)
BEGIN
SET NOCOUNT ON
if not exists (select * from sysobjects where name='Audit_1' and xtype='u')
create table audit_1(
codigo int identity(0,1) not null primary key,
utilizador char(30) not null default current_user,
pc char(30) not null default host_name(),
data datetime not null default getdate(),
empnum int not null,
salario_antigo int not null,
salario_novo int not null)
DECLARE @id INT, @salarionovo int, @salarioantigo int
SELECT
@id = (SELECT empnum FROM deleted),
@salarionovo=(SELECT salario FROM inserted),
@salarioantigo=(SELECT salario FROM deleted)
Insert Into Audit_1 (Empnum,Salario_Novo,Salario_Antigo) Values(@id,@salarionovo,@salarioantigo)
END
```

Para testar altere alguns salarios e depois veja o conteúdo da tabela

Audit_1

Exemplo - Procedimentos

```
create table utente (  
id_utente int,  
nome char(20),  
localidade char(20),  
constraint tb_utente_pk primary key(id_utente))
```

```
insert into utente values (1,'Filipe', 'CBR')  
insert into utente values (2,'Fidalgo', 'LX')
```

```
select * from utente
```

Exemplo - Procedimentos

```
drop procedure lista_utilizadores
```

```
CREATE PROCEDURE LISTA_UTENTES  
AS  
SELECT * FROM utente
```

```
exec lista_utentes
```


Exemplo - Procedimentos

```
drop procedure insere_utente
```

```
CREATE PROCEDURE insere_utente (@nome char(20), @loc  
char(20))
```

```
AS
```

```
declare @tmpMax int
```

```
select @tmpMax = max(id_utente) from utente
```

```
set @tmpMax = @tmpmax + 1
```

```
INSERT INTO utente (id_utente, nome, localidade)
```

```
VALUES (@tmpMax, @nome, @loc)
```

```
exec insere_utente 'Miguel','PT'
```

Exemplo - Procedimentos

```
ALTER PROCEDURE valida
(
    @user varchar(20) ,
    @pass varchar(20)
)
AS
DECLARE @RES char(10)
IF (SELECT count(*) as count FROM utente WHERE nome =
    @user and localidade=@pass) = 0
    SET @RES = 'Nao Existe'
ELSE
    SET @RES = 'Existe'

select @RES as RES

exec valida 'Rui', 'EV'
```

Exercícios - Triggers

1- Crie um trigger (Atualiza_Media) que por cada nova inserção na tabela resultado, actualize a média na tabela aluno.

2 – Considere a tabela “Quadro_Honra”: (id_aluno, nome_disciplina, nota_maxima)

Crie um trigger chamado “Atualiza_Quadro_Honra”, que por cada nova inserção na tabela resultado, o trigger verifique se a nota inserida ultrapassa a nota máxima da respectiva disciplina na tabela “Quadro_honra” e em caso positivo faça a respectiva atualização.

...