SQL Programming

Última atualização a *12 de junho de 2020*

Batch

Define-se por um grupo de uma ou mais instruções SQL que constituem uma unidade lógica. São **delimitadas** pelo terminador **Go**.

> Não são permitidos ';' no final das instruções Batch!

```
-- Bloco Batch
GO [n];
-- n é opcional e indica o número de vezes a executar a batch
```

Compilação

As instruções são compiladas antes de executadas, pelo que um erro sintático gera erro antes da execução de qualquer instrução. Os erros de *runtime*, como o nome indica, só são detetados durante a execução e por isso não anula as instruções executadas previamente.

Contexto

Para mudar a base de dados a utilizar na Batch, utiliza-se `USE <dbName>`.

No final da sua execução, todas as variáveis locais, tabelas temporárias e cursores criados são eliminados.

Variáveis

As variáveis podem ser declaradas com ou sem inicialização e pode-lhe ser atribuído um valor posteriormente.

```
DECLARE @<varname> <type> [= <value>];
SET @<varname> = <value>;
```

> Se não forem inicializadas, terão o valor NULL.

Podem ainda ser realizadas **operações aritméticas** sobre elas. `+, -, *`

```
SET @<varname> <op>= <number> SELECT @<varname>;
```

A sua utilização é feita com recurso ao operador **Select**.

```
SELECT @<varname1>,@<varname2>;
-- Output: <value1> <value2>
```

Podem-lhe ser atribuídos valores de consultas. Caso haja mais do que um tuplo, irá ficar com o valor do último.

```
SELECT @<varname>=<attribute> FROM <table> WHERE id=1;
-- A variável vai ficar com o valor do atributo selecionado do tuplo com id=1
```

Estas podem ainda ser utilizadas nas operações de consulta de tabelas.

```
-- Filtrar  atributos
SELECT * FROM <table> WHERE <attr>=@<varname>;
```

Output

Para imprimir mensagens na consola, recorre-se ao comando **PRINT**.

```
PRINT "Some text:" + STR(@<varname>);
-- JAVA: System.out.println("Some text:" + <varname>.toString());
```

Instruções de controlo de fluxo

> As operações condicionais geralmente são de comparação de valores (>, <, >=, <=) ou de verificação de existência EXISTS()

Blocos de instruções

```
BEGIN
-- Bloco de instruções
END
-- JAVA: { // Bloco de instruções... };
```

> Em todas as instruções de controlo de fluxo, caso as instruções a realizar em cada condição sejam mais do que uma, estas têm de ser inseridas num bloco de instruções.

Instruções condicionais

```
IF <boolean>
    -- Instruções
[ELSE]
```

Ciclos

```
WHILE <boolean>
    -- Instruções
```

Alternativas

```
@<varname>/>attr> =
CASE @<varname>/<attr>
    WHEN <n> THEN <value>
    WHEN <n> THEN <value>
    WHEN <n> THEN <value>
    ELSE <valueDefault>
END
```

Tabelas temporárias

São iguais às restantes tabelas, com exceção da sua persistência, que é limitada, podendo assumir dois tipos.

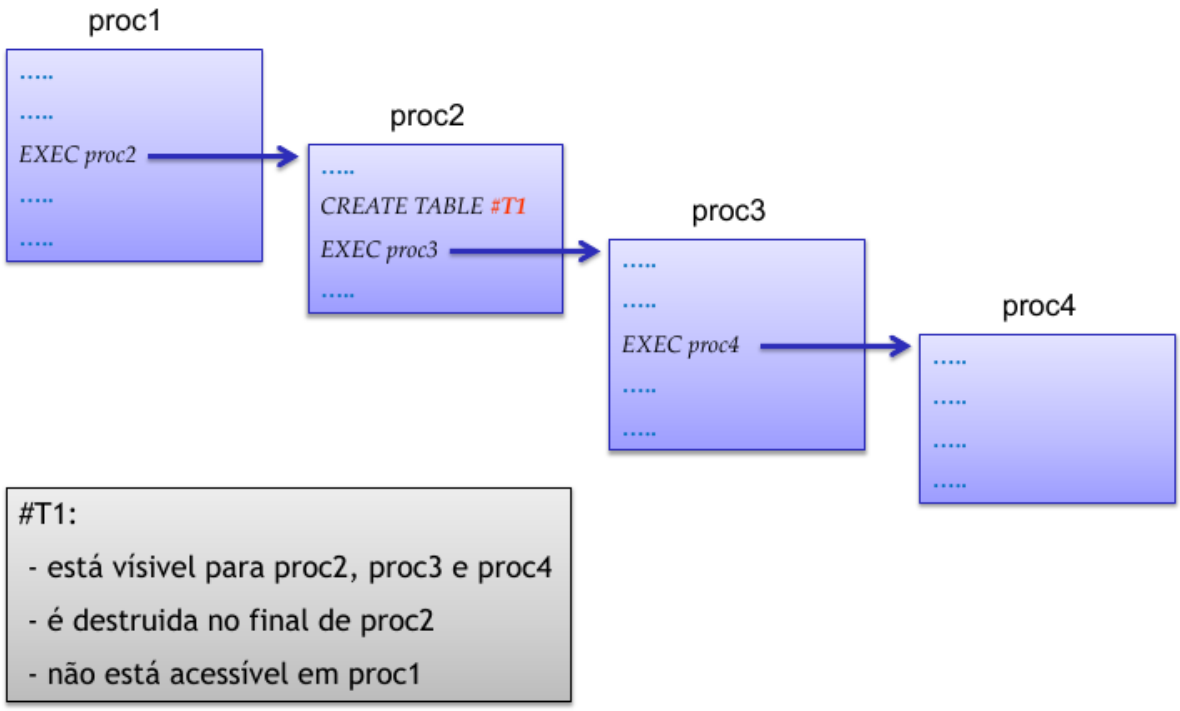
Locais

São criadas na BD ***tempdb***, sendo visíveis apenas na sessão e nível (e *inner levels*) em que são criadas. O seu nome é precedido de um cardinal (#).

```
CREATE TABLE #<name> (...);
```

Se não forem eliminadas pela operação DROP durante a execução, serão quando a função termina.

No caso de uma batch ad-hoc (*query editor*), a tabela só é eliminada quando a sessão for encerrada, podendo ser utilizada por *batches* diferentes (com GO pelo meio).



#T1:
- está vísivel para proc2, proc3 e proc4
- é destruida no final de proc2
- não está acessível em proc1

Globais

São criadas na BD ***tempdb***, mas são visíveis para outras sessões (com acesso total). O seu nome é precedido de dois cardinais (##).

```
CREATE TABLE ##<name> (...);
```

Se não forem eliminadas pela operação DROP durante a execução, serão apenas quando a última sessão se desconecta.

> Em alternativa pode criar-se uma tabela na BD **tempdb**, que será eliminada apenas quando o SQL Server é reiniciado.

Tabelas como variáveis

O seu contexto é semelhante ao das tabelas locais, mas não estão visíveis em *inner levels*. Podem em alternativa ser passadas como parâmetros.

Têm ainda restrições adicionais, sendo apenas permitidas chaves primárias, defaults, nulls e unique.

Apesar de serem declaradas como variáveis, continuam a ser definidas na ***tempdb***.

```
-- Declaração
DECLARE @<tablevarname> TABLE (...);
-- Operações de manipulação
INSERT INTO @<tablevarname> (<attr>, ...) VALUES (<value>, ...);
SELECT * FROM @<tablevarname>;
```

Script

É um ficheiro de texto com extensão `.sql` que contém uma ou mais *batches*. Estas são executadas em sequência.

Cursor

É uma ferramenta *server side* que permite percorrer sequencialmente os tuplos retornados por uma operação de consulta.

```
-- Declaração
DECLARE <cursorName> CURSOR [<options>] FOR SELECT ...;
-- Abertura (para obter dados)
OPEN <cursorName>;
-- Avançar para a próxima linha
FETCH [<direction>] <cursorName> [INTO @<varname>, ...];
-- Fechar
CLOSE <cursorName>;
-- Desalocar memória
DEALLOCATE <cursorName>;
```

Para analisar o estado do cursor, em T-SQL existe a função `@@fetch_status`, que retorna `0: Ok`, `-1: Fim dos registos`, `-2: Tuplo não disponível`.

É ainda importante destacar que ao fechar um cursor este pode voltar a ser reaberto, pelo é importante desalocar a memória que lhe está alocada caso não necessitemos de o voltar a utilizar.

Opções

As principais opções para a criação de cursores são:

- **Static** `+ rápida`
  - Cursor opera sobre uma cópia da BD na ***tempdb***, sobre a qual são feitas as eventuais alterações, que não têm qualquer efeito sobre os dados originais.
- **Keyset**
  - O cursor cria uma relação na ***tempdb*** apenas com os atributos necessários para identificar os tuplos, atuando sobre uma relação que resulta do *JOIN* interno dessa relação com a original.
  - Atualizações e eliminações de tuplos são vistos pelo cursor (eliminados têm *status* -2).
  - A cada linha fazem operação de *SELECT*, por isso devem ser evitados caso os dados tenham várias origens.
- **Dynamic**
  - O cursor itera sobre a relação original, sendo por isso visíveis todas as alterações.
  - Não há tratamento especial para as alterações, pois as linhas eliminadas deixam de existir e por isso não são processadas e caso seja adicionada uma nova esta será processada pelo cursor quando chegar ao seu índice.
- **Fast forward** `alta performance`
  - Este cursor é *read-only*.

AR: A alternativa

A alternativa são os *set based query*, assentes em álgebra, que apesar de mais complexos de desenhar, são **bastante mais rápidos** de executar.

> Os cursores podem ser vistos como pesca com linha e as operações *set based* como pesca com rede.

Os cursores continuam a ser uma melhor solução quando...

- ... é necessário realizar um procedimento sobre cada tuplo da relação;
- ... é necessário fazer somas cumulativas;
- ... é necessário relacionar um tuplo com os seus vizinhos.

Stored procedures

É um ficheiro que resulta da compilação de uma Batch, pelo que a sua execução deixa de envolver compilação.

> Pode ter parâmetros de entrada e valor de retorno, que pode ser um *status* ou um conjunto de registos (tuplos).

Os procedimentos são guardados na *cache* a primeira vez que são executados, tornando a sua execução mais rápida.

Assim, o processo que na escrita de SQL *ad hoc* passa **sempre** por `Validação da sintaxe > Compilação > Execução > Retorno` divide-se num processo de criação com `Validação da sintaxe > Compilação`, executado apenas uma vez e nas seguintes limita-se a `Execução > Retorno`.

Destacam-se as seguintes atributos:

Extensibilidade Cria uma abstração da BD, através da sua encapsulação.

Performance A sua execução é a mais rápida possível nas operações sobre a BD.

Usabilidade É mais fácil escrever código uma vez e consumi-lo várias vezes do que estar constantemente a criar código *ah hoc* (para um fim específico)

Integridade É menos propício a conter erros de integridade e mais fácil de testar

Segurança Aceder às tabelas exclusivamente através deles é um procedimento padrão

Criar, atualizar e eliminar procedures

```
CREATE PROC[EDURE] <name> [ @<varName> <datatype> ] [= default] OUTPUT [, ..., n]
AS
-- SQL statements
GO
```

```
-- Exemplo: CREATE Storage Procedure with parameters + RETURN
CREATE PROC GroupLeader_Members @Emp_Code varchar(10) = null
AS
    IF @Emp_Code is null
    BEGIN
        PRINT 'Please enter Employee Code!'
        RETURN
    END
    SELECT * FROM Employees
    WHERE EMP_EMP_ID = (SELECT EMP_ID FROM Employees
                        WHERE Emp_Code = @Emp_Code)
    ORDER BY Emp_Name
```

A edição substitui o procedimento existente pelo novo.

```
ALTER PROC[EDURE] <name> [ @<varName> <datatype> ] [= default] OUTPUT [, ..., n]
AS
-- SQL statements
```

```
DROP PROC[EDURE] <name>;
```

Tipos de procedures

Existem procedimentos de **sistema**, cujo nome começa por `sp_` e são criados na BD ***master***, podendo por isso ser utilizados por qualquer BD.

Os **locais**, por outro lado e como o nome sugere, são definidos numa BD local, sendo o seu nome livre. No entanto, deve-se privilegiar a normalização com prefixos como `pr_` ou `p_`, por exemplo.

Execução

```
EXEC[UTE] <procName> [@<varName> = <value>];
```

```
-- Exemplos: Execução de Storage Procedure
-- Sem parâmetros de entrada
EXEC dbo.CategoryList;

-- Com um parâmetros de entrada
EXEC Department_Members 'Accounting';

-- Com múltiplos parâmetros de entrada
-- ... por posição
EXEC pr_GetTopProducts 1, 10
-- ... por nome (ordem não interessa)
EXEC GetTopProducts  @EndID = 10, @StartID = 1
```

> No exemplo é devolvido um record set

Retorno

Os procedimentos podem retornar informação que não esteja na forma de record sets. Para tal, devemos inicializar primeiro a variável que receberá o valor do retorno.

```
-- Exemplo: Declaração e utilização de proc. com parâmetro de saída
-- Criação
CREATE PROC dbo.GetProductName (
            @ProductCode CHAR(10), @ProductName VARCHAR(25) OUTPUT)
AS
SELECT @ProductName = ProductName
FROM dbo.Product
WHERE Code = @ProductCode;

-- Utilização
DECLARE @ProdName VARCHAR(25);
EXEC dbo.GetProductName '1001', @ProdName OUTPUT;
PRINT @ProdName;
```

Podemos também retornar um inteiro.

```
-- Exemplo: Storage Procedure with Return
GO
CREATE PROC dbo.IsItOK ( @OK VARCHAR(10) )
AS
IF @OK = 'OK'
  RETURN 0;
ELSE
  RETURN -100;
GO

DECLARE @ret as int;
EXEC @ret=dbo.IsItOK 'OK';
SELECT @ret;
```

Gestão de erros

Para auxiliar o programador na gestão de erros, o SQL dispõe da variável `@@error`, que devolve o *status* da última instrução executada (0 para sucesso) e `@@rowcount`, que indica o número de tuplos afetados também pela última instrução.

Criar erros

Para criar um erro, existem duas sintaxes possíveis.

```
RAISERROR <number> <message>;
-- RAISERROR 3 "Argumento inválido";
RAISERROR (<message>, <gravidade>, <estado>, <argumentosOpcionais>);
-- RAISERROR ("Argumento inválido em %s", 14, 1, "Tabela de teste");
```

A gravidade é definida com base na seguinte tabela:

Severity Code	Description
10	Status message: Does not raise an error, but returns a message, such as a PRINT statement
11–13	No special meaning
14	Informational message
15	Warning message: Something may be wrong
16	Critical error: The procedure failed

Tratamento de erros

```
BEGIN TRY
-- SQL Statements
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber
    ,ERROR_SEVERITY() AS ErrorSeverity
    ,ERROR_STATE() AS ErrorState
    ,ERROR_PROCEDURE() AS ErrorProcedure
    ,ERROR_LINE() AS ErrorLine
    ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

Encriptação

Quando executamos um procedimento, o seu conteúdo é mostrado no terminal do SQL Server. Para isto não acontecer, podemos definir na sua criação que este é encriptado.

```
CREATE PROC[EDURE] <name> [ @<varName> <datatype> ] [= default] OUTPUT [, ...,
n]
WITH ENCRYPTION -- HEREEE
AS
-- SQL statements
GO
```

User Defined Functions (UDF)

As funções definidas pelo utilizador trazem os mesmos benefícios dos procedimentos, pois também são compilados e otimizados, sendo utilizados para **incorporar lógica complexa dentro de uma consulta**.

Também se relacionam com as *views* uma vez que podem ser utilizadas como fontes de dados (FROM ...), tendo como extra o facto de aceitar parâmetros .

> O nome do esquema (*schema*) em que se insere é **obrigatório** na invocação da UDF.

Na sua sintaxe podemos defini-las como sendo **SCHEMA BINDING**, prevenindo assim a alteração ou eliminação dos objetos utilizados na função. Assim, as relações utilizadas por estas deixam de suportar eliminação ou alteração de tuplos existentes, sendo a única operação possível a adição de novos.

```
CREATE FUNCTION ....
WITH SCHEMA BINDING --HEREEEE
AS
```

A sua invocação é bastante simples.

```
SELECT dbo.FuncName(args...)
```

UDF Escalar

Retornam um único valor.

Caracterizam-se por ser **determinísticas**, ou seja, para os mesmos valores de entrada retornam sempre a mesma saída. Não podem fazer *updates* às bases de dados, nem invocar comandos `DBCC`.

Não permite retornar valores binários (text, timestamp,...), tabelas, nem cursores.

A recursividade está limitada a 32 níveis.

```
CREATE FUNCTION function_name
[ @param_name data_type] [= default] [ READONLY ][,...,n]
RETURNS return_data_type
AS
T-SQL_statement(s)
```

```
CREATE FUNCTION dbo.fsMultiply (@A INT, @B INT = 3) RETURNS INT
AS
BEGIN
     RETURN @A * @B;
END;

GO
SELECT dbo.fsMultiply (3,4), dbo.fsMultiply (7, DEFAULT);

SELECT dbo.fsMultiply (3,4) * dbo.fsMultiply (7, DEFAULT);
```

```
----------- -----------
12          21

-----------
252
```

UDF Inline Table-valued

De forma similar às *views* são caracterizados por *wrapers* para construções SELECT, acrescendo o facto de suportarem parâmetros.

```
CREATE FUNCTION function_name
[ @param_name data_type] [= default] [ READONLY ][,...,n]
RETURNS TABLE
AS
T-SQL_statement {RETURN SELECT statement}
```

```
CREATE FUNCTION dbo.ftPriceList (@Code CHAR(10) = Null, @PriceDate DateTime)
RETURNS Table
AS
     RETURN(SELECT Code, Price.Price
                  FROM dbo.Price JOIN dbo.Product AS P
                       ON Price.ProductID = P.ProductID
                  WHERE EffectiveDate = (SELECT MAX(EffectiveDate)
                                         FROM dbo.Price
                                         WHERE ProductID = P.ProductID
                                               AND EffectiveDate <= @PriceDate)
                        AND (Code = @Code OR @Code IS NULL));

GO
SELECT * FROM dbo.ftPriceList(DEFAULT, '20020220');
```

UDF Multi-statement Table-valued

Combina as UDF escalares com as inline table-valued, pois oferecem a possibilidade de conter código complexo e retornar um conjunto.

Cria uma tabela numa variável, introduz-lhe tuplos e retorna-la, podendo esta depois ser utilizada numa consulta (SELECT).

```
CREATE FUNCTION dbo.ftPriceAvg() RETURNS @Price TABLE (Code char(10), EffectiveDate datetime,
                                                                                Price money)
AS
   BEGIN
         INSERT @Price (Code, EffectiveDate, Price)
                   SELECT Code, EffectiveDate, Price
                   FROM Product JOIN Price ON Price.ProductID = Product.ProductID;

         INSERT @Price (Code, EffectiveDate, Price)
                   SELECT Code, Null, Avg(Price)
                   FROM Product JOIN Price ON Price.ProductID = Product.ProductID
                   GROUP BY Code;
         RETURN;
  END;

GO
SELECT * FROM dbo.ftPriceAvg();
```

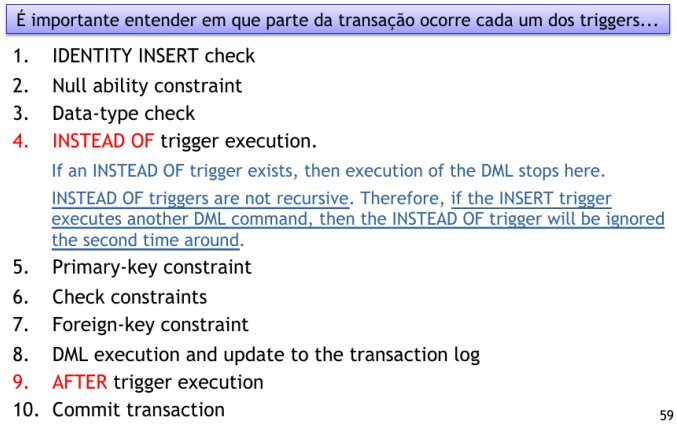
Trigger

Quando manipulamos dados podemos executar determinadas ações quando ocorrem ações previstas. A estas ações chamamos **triggers**.

> São utilizados por exemplo para criar *constraints* ou *defauls* complexos ou assegurar a integridade entre bases de dados.

Estes são disparados uma vez por cada operação de modificação dos dados e foram pensados para eventos que afetam um único tuplo (se afetarem mais do que um geralmente estão associados a situações de mau desempenho).

Existem dois tipos de *triggers*, que diferem quanto ao propósito, timing e efeito.




```
-- Criação
CREATE TRIGGER trigger_name ON <tablename>
AFTER | INSTEAD OF
    [INSERT] [,] [UPDATE] [,] [DELETE]
AS
SQL_Statement

-- Activar | Desactivar
ALTER TABLE <tablename> ENABLE | DISABLE TRIGGER trigger_name
-- ou
ENABLE | DISABLE TRIGGER trigger_name ON <tablename>

-- Eliminar
```

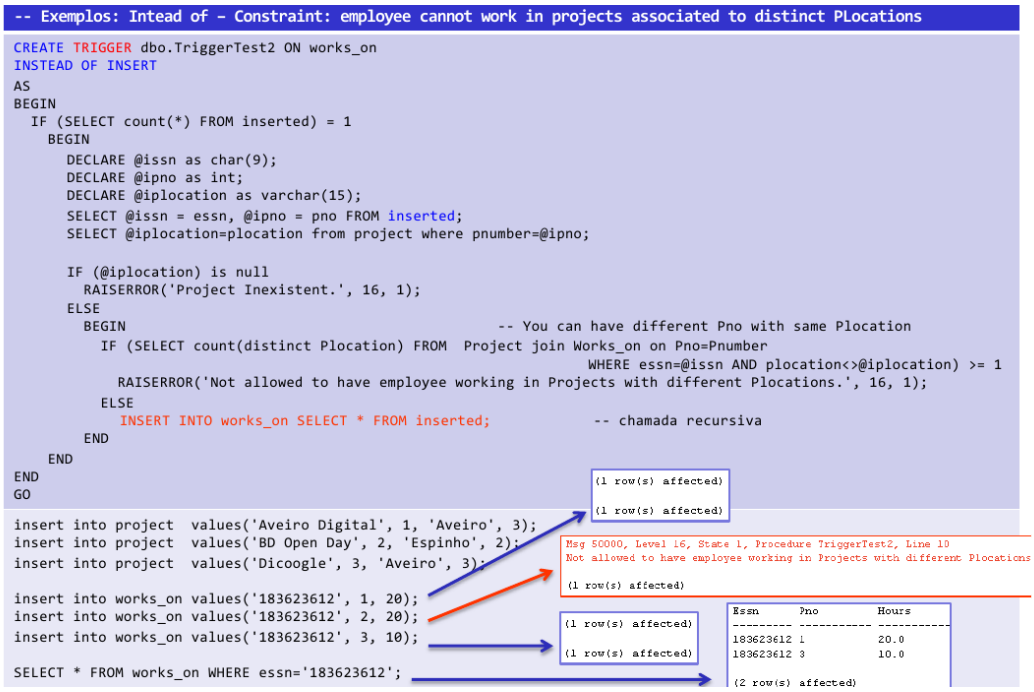
```
DROP TRIGGER trigger_name ON <tablename>
```

INSTEAD OF

Faz com que o *trigger* seja executado no lugar da ação que o gerou (não são recursivos), no ponto 4 do fluxo de execução, pelo que os pontos restantes não são executados. Pode por isto contornar problemas de integridade referencial, mas não de nulidade, uma vez que esta é validada no ponto 2.

Só é permitido um por tabela e a ação associada **não é executada**. Fica assim à responsabilidade do *trigger* efetuar a operação pretendida, ou não.

É utilizada quando sabemos que a ação tem uma elevada probabilidade de não ser permitida e pretendemos que outra lógica seja executada em vez dela.



AFTER

Faz com que o *trigger* dispare apenas no final da ação que o gerou, podendo por isso assumir que os dados passaram todas as verificações de integridade dos dados.

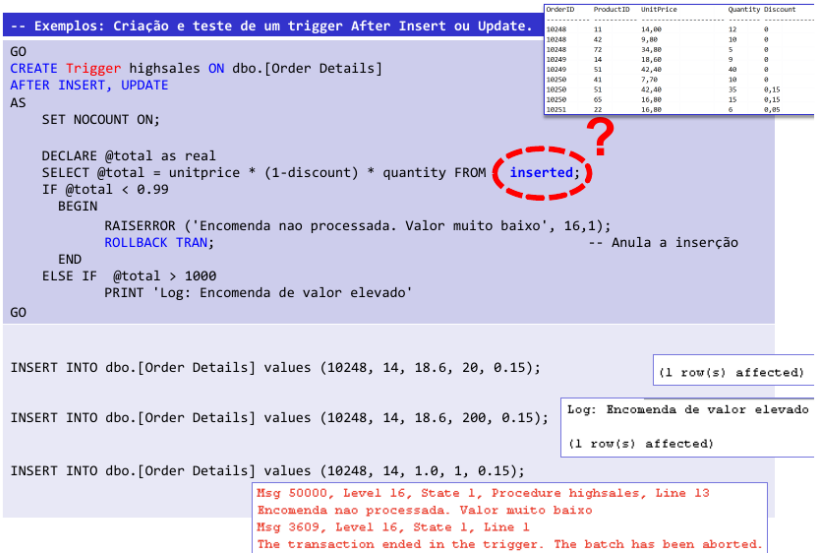
> Por motivos históricos, utilizar AFTER é o mesmo que FOR.

No entanto, uma vez que ocorre no final da ação, não pode corrigir eventuais problemas nos dados .

> Ocorre antes do *commit* da transação DML, pelo que podemos fazer *rollback* da transação se os dados não forem aceitáveis. (Noções abordadas quando forem estudadas transações)

Podem haver vários por tabela.

> As utilizações mais comuns são processos de validação que recorrem a várias tabelas, assegurar regras de negócio complexas, efetuar auditorias, atualizar campos calculados ou assegurar integridade referencial definida pelo utilizador (no entanto deve ser privilegiada a verificação declarativa).



Quais as diferenças?

	Instead of Trigger	After Trigger
DML statement	Simulated but not executed	Executed, but can be rolled back in the trigger
Timing	Before PK and FK constraints	After the transaction is complete, but before it is committed
Number possible per table event	One	Multiple
May be applied to views?	Yes	No
Nested?	Depends on server option	Depends on server option
Recursive?	No	Depends on database option

Limitações

Num *trigger* não são permitidas as seguintes operações:

- CREATE, ALTER, or DROP database
- RECONFIGURE
- RESTORE database or log
- DISK RESIZE
- DISK INIT

Tabelas lógicas

O SQL Server permite ter acesso a duas tabelas lógicas com uma imagem *read-only* dos dados alterados por operações de inserção ou remoção de dados.

DML Statement	Inserted Table	Deleted Table
Insert	Rows being inserted	Empty
Update	Rows in the database after the update	Rows in the database before the update
Delete	Empty	Rows being deleted

O seu alcance (*scope*) é bastante limitado, não sendo acessíveis aos *stored procedures* invocados por um *trigger*.

Verificar colunas alteradas

Com base nestas tabelas o SQLS disponibiliza duas funções que permitem verificam se colunas foram afetadas.

> update(colName) boolean

```
CREATE Trigger ...
AFTER UPDATE
AS
    IF update(ContactName)
        PRINT '...'
```

> columns_updated()
>
> Retorna um *bitmapped varbinary* representando as colunas alteradas. O seu tamanho depende do número de colunas da tabela. Se uma coluna foi alterada então o seu bit está a *true*. Temos de
> utilizar bitwise operators para determinar quais as colunas alteradas.

Funcionalidades úteis

> sp_helptext triggerName
>
> Permite ler o conteúdo do trigger

> sp_helptrigger tableName
>
> Permite consultar lista dos *triggers* numa determinada relação

---

Resumo feito com base nos *slides* teóricos da unidade curricular de Base de Dados, da autoria do professor Carlos Costa e na consulta da página DEVMEDIA.