

# Base de Dados - SQL Programming

Base de Dados - 2024/25

Carlos Costa

# Índice

- Script e Batch
- Cursor
- Stored Procedure
- User Defined Function
- Trigger

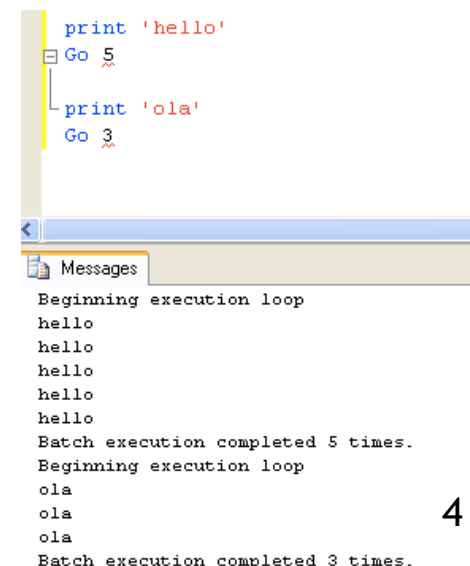
Baseado em SQL Server (T-SQL)

# Script & Batch

# Batch

- Definição: Grupo de uma ou mais instruções SQL que constituem uma unidade lógica.
- Um **erro sintáctico** numa instrução provoca a falha de toda a batch.
- Um **erro de runtime** não anula instruções SQL prévias (nessa batch).
- Não são transações\*.
- São delimitadas pela terminador **GO**.
  - GO não é enviada para o servidor
  - “GO n” – executa a batch n vezes

\* vamos ver mais à frente



```

print 'hello'
Go 5
print 'ola'
Go 3
  
```

Messages

```

Beginning execution loop
hello
hello
hello
hello
hello
hello
Batch execution completed 5 times.
Beginning execution loop
ola
ola
ola
Batch execution completed 3 times.
  
```

# Batch - Utilização

- Terminada a batch, são eliminadas todas as variáveis locais, tabelas temporárias e cursores criados.
- Algumas instruções são únicas na batch.
  - i.e. só existe essa instrução
  - Exemplo:
    - CREATE PROCEDURE
    - CREATE DEFAULT
    - CREATE RULE
    - CREATE TRIGGER
    - CREATE VIEW
- Para mudar de base de dados:
  - USE <dn\_name>;

# Script

- Trata-se de um ficheiro de texto contendo uma ou mais batches delimitadas por GO.

Por exemplo: `EmployeeManipulation.sql`

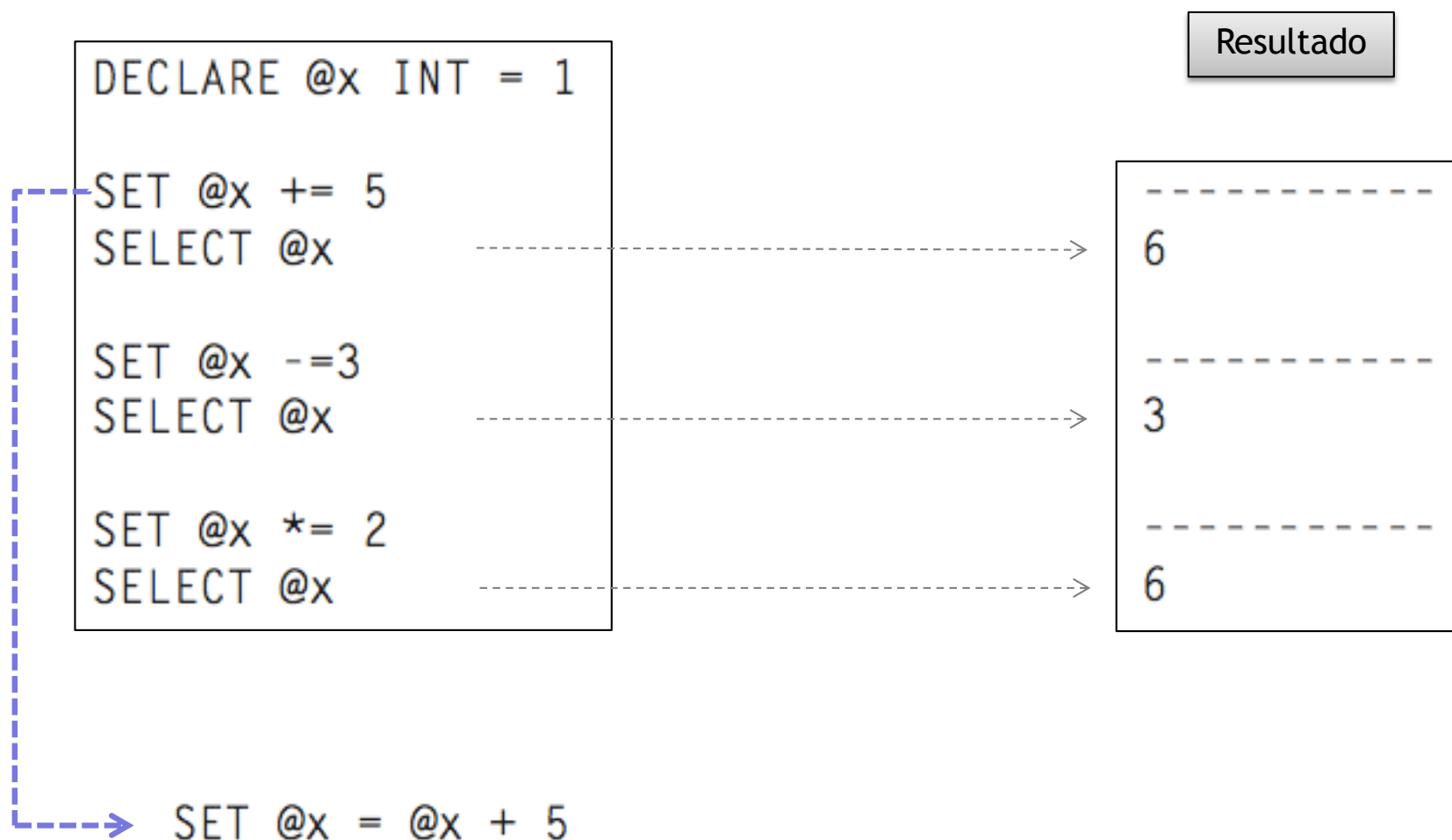
- As batch são executadas em sequência.

# Variáveis

- Declaração:
  - `DECLARE @x varchar(10) = 'Ola'`
  - `DECLARE @min_range int, @max_range int`
- Atribuição de um valor:
  - `SET @x = 'Kabung'`
  - `SET @min_range = 0, @max_range = 100`
- Atribuição de um valor numa instrução SELECT:
  - `SELECT @price = price FROM titles`  
`WHERE title_id = 'PC2091';`

# Variáveis - Operações Aritméticas

## Adição, Subtração e Multiplicação





# Batches e Variáveis - Exemplos

## Batch Initialization and Scope

### Resultado

```
DECLARE @Test INT ,
        @TestTwo NVARCHAR(25);
SELECT @Test, @TestTwo;
```

```
SET @Test = 1;
SET @TestTwo = 'a value';
SELECT @Test, @TestTwo ;
GO
```

```
SELECT @Test AS BatchTwo, @TestTwo;
```

```
-----
NULL          NULL
(1 row(s) affected)
```

```
-----
1             a value
(1 row(s) affected)
```

```
Msg 137, Level 15, State 2, Line 2
Must declare the scalar variable "@Test"
```

# Utilização de Variáveis em Consultas

```

DECLARE @TempID VARCHAR(5),
        @TempCustomerName VARCHAR(30);

SELECT @TempID = CustomerID,
       @TempCustomerName = ContactName
FROM Customers
ORDER BY CustomerID;

SELECT @TempID, @TempCustomerName;
    
```

Select to Variable

```

-----
WOLZA Zbyszek Piestrzeniewicz
(1 row(s) affected)
    
```

último tuplo

Use Variable in WHERE

```

DECLARE @TempID VARCHAR(5) = 'BERGS';

SELECT ContactName
FROM Customers
WHERE CustomerID = @TempID;
    
```

```

ContactName
-----
Christina Berglund
(1 row(s) affected)
    
```

# PRINT

- Imprimir mensagem na consola  
PRINT string
- Outras Linguagens de Programação
  - Java: System.out.print
  - C#, VB.NET: Console.WriteLine

## -- Exemplos

```
PRINT 'ola';
```

```
DECLARE @Temp int = 5;  
PRINT 'TEMP value: ' + STR(@Temp);
```

# Instruções de Controlo de Fluxo

- BEGIN ... END
- IF ... ELSE
- CASE ... WHEN
- WHILE

## BEGIN ... END

- Define um bloco de instruções
  - `block_of_statements`
- Outras Linguagens de Programação
  - C#, Java, C: `{ ... }`
  - Pascal, Delphi: `BEGIN ... END`

# IF ... ELSE

IF Boolean\_expression

statement | block\_of\_statements

[ELSE

statement | block\_of\_statements ]

## -- Exemplos

```
IF (SELECT ytd_sales FROM titles WHERE title_id='PC1035') > 5000
  PRINT 'Year-to-date sales are greater than $5,000 for PC1035.'

IF EXISTS(SELECT * FROM [ORDER] WHERE Closed = 0)
  BEGIN
    PRINT 'Process Orders';
    PRINT 'BLA..BLA';
  END
ELSE
  PRINT 'BLE..BLE';
```

# WHILE

WHILE Boolean\_expression

SQL\_statement | block\_of\_statements |

[BREAK] | [CONTINUE]

## -- Exemplos

```
WHILE (SELECT AVG(royalty) FROM roysched) < 25
BEGIN
    UPDATE roysched SET royalty = royalty * 1.05;
    IF (SELECT MAX(royalty) FROM roysched) > 27
        BREAK;
    ELSE
        CONTINUE;
END;
```

```
DECLARE @i as int = 1;
WHILE @i < 100
BEGIN
    IF (@i % 2) = 0
        print str(@i) + ' - Par';
    ELSE
        print str(@i) + ' - Impar';
    SET @i += 1;
END;
```

# CASE ... WHEN

CASE input\_expression

WHEN when\_expression THEN result\_expression

[WHEN when\_expression THEN result\_expression...n]

[ELSE else\_result\_expression ]

END

## -- Exemplo

```
SELECT OrderID, CustomerID ,  
       EmployeeName =  
       CASE EmployeeID  
         WHEN 1 THEN 'Mario'  
         WHEN 2 THEN 'Julio'  
         WHEN 3 THEN 'Vasco'  
         WHEN 4 THEN 'Sousa'  
         WHEN 5 THEN 'Rui'  
         ELSE 'desconhecido'  
       END  
FROM [Orders]
```

	OrderID	CustomerID	EmployeeName
1	10248	VINET	Rui
2	10249	TOMSP	desconhecido
3	10250	HANAR	Sousa
4	10251	VICTE	Vasco
5	10252	SUPRD	Sousa
6	10253	HANAR	Vasco
7	10254	CHOPS	Rui
8	10255	RICSU	desconhecido
9	10256	WELLI	Vasco



# Tabelas Temporárias

- Há situações em que necessitamos de criar tabelas de uso temporário.
- Criam-se da forma usual e têm as mesmas características que as “normais” excepto a persistência.
- Dois Tipos:
  - Temporárias Locais
  - Temporárias Globais
- Tabelas como Variáveis

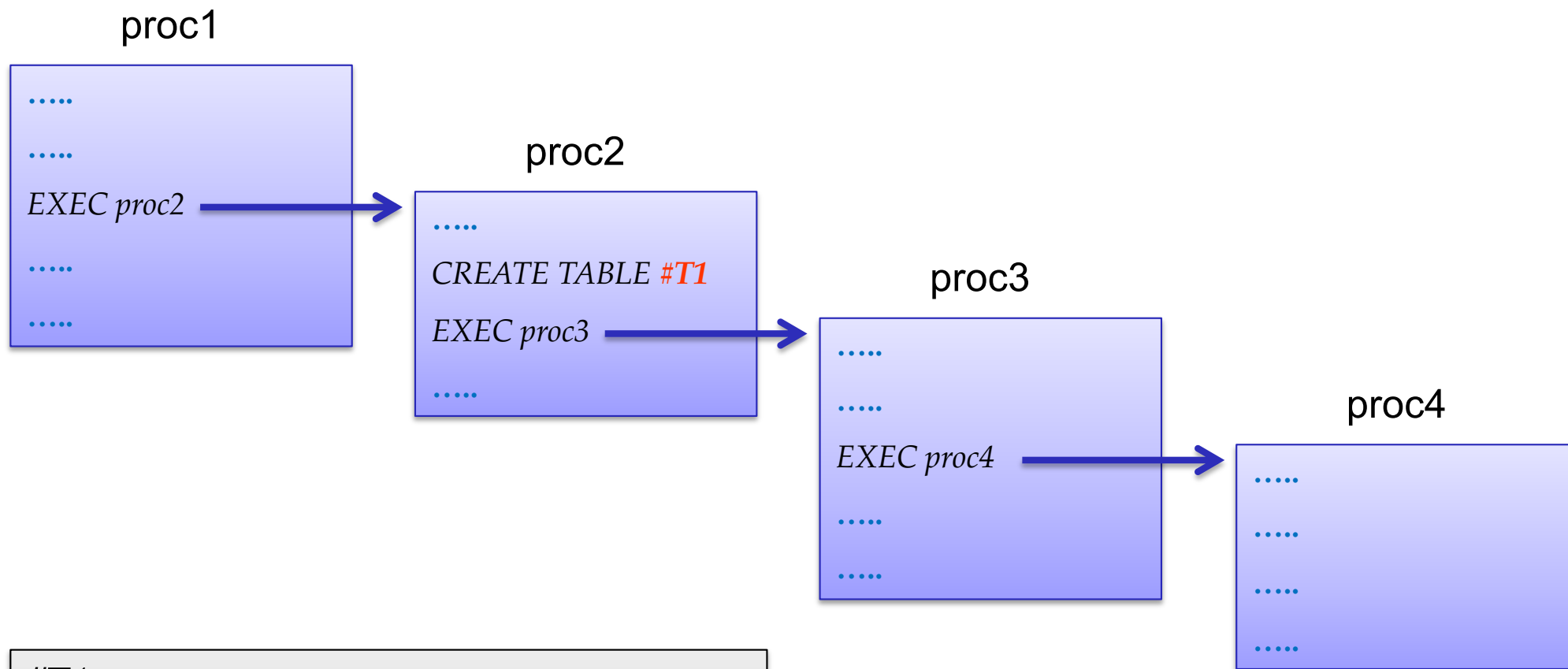
# Tabelas Temporárias Locais

- São sinalizadas com o carácter # antes do nome.
- São criadas na base de dados *tempdb*.
- Estão visíveis
  - Só na sessão que as criou
  - No level em que são criados e todos os inner level (da call stack)
- São eliminadas quando o procedimento ou função termina.
  - Podem ser eliminadas da forma normal (drop)
- No caso de uma batch ad-hoc (query editor) fica visível até encerrar a sessão, mesmo tendo um GO pelo meio.

## -- Exemplo:

```
GO
CREATE TABLE #Hello(
  id      INT          PRIMARY KEY,
  name    VARCHAR(25));
GO
SELECT * FROM #Hello;      -- Está visível (query editor).
```

# T. Locais Temporárias - Call Stack



#T1:

- está visível para `proc2`, `proc3` e `proc4`
- é destruída no final de `proc2`
- não está acessível em `proc1`

# Tabelas Temporárias Globais

- Utilizamos dois **##** antes do nome.
- Similares às local temporary tables (*tempdb*) mas têm um scope maior.
  - Ficam visíveis para outras sessões.
    - **Apropriadas para partilha de dados**
    - **Todos têm full access!!!**
  - São eliminadas quando a última sessão desconecta.

```
-- Exemplo: Verifica se já alguém criou antes de criar
```

```
IF NOT EXISTS(SELECT * FROM tempdb.sys.objects  
              WHERE name = '##TempWork')  
  CREATE TABLE ##TempWork(  
    PK INT PRIMARY KEY,  
    Col1 INT );
```

Alternativa: Criar a tabela normalmente na BD tempdb. Só é eliminada quando o SQL Server reinicializa.

# Tabelas como Variáveis

- São similares a tabelas temporárias locais mas têm um scope mais limitado:
  - Tem o mesmo scope que as variáveis locais
  - Mas não estão visíveis em inner levels da call stack
    - Podem ser passados como parâmetros
- Declaram-se como variáveis
  - Também têm existência na tempdb

## -- Exemplo: Declaração e Utilização

```
DECLARE @WorkTable TABLE (PK INT PRIMARY KEY, Col1 INT NOT NULL);
```

```
INSERT INTO @WorkTable (PK, Col1) VALUES (1, 101);
```

```
SELECT PK, Col1 FROM @WorkTable;
```

PK	Col1
1	101

# Tabelas como Variáveis - Limitações

- Desaparecem quando a batch, procedimento ou função, onde foram criadas, chega ao fim.
- Limitadas em termos de restrições:
  - Não é permitido: chaves estrangeiras e check.
  - Permitido: chaves primária, defaults, nulls e unique.
- Não podem ter objetos dependentes.
  - Chaves estrangeiras ou triggers.

# Cursor

# Cursor

- Ferramenta que permite percorrer sequencialmente os tuplos retornados por determinada consulta (SELECT).
- Tipicamente temos duas abordagens:
  - Set based query (AR) versus cursor operation
- Soluções set-based são, em geral, bastante mais rápidas do que cursores.
- Usualmente os utilizadores sentem-se mais confortáveis a pensar em termos de ciclos e ponteiros do que em consultas baseadas em álgebra relacional.
  - Défice de formação em base de dados? Álgebra Relacional?
- Em SQL Server os cursores são server-side.

Analogia da “Pesca”: podemos ver os cursores como pesca à linha e as operações set-based como pesca com rede.



# Cursor - 5 steps

## 1. Declaração

```
-- SQL-92
DECLARE CursorName [CursorOptions] CURSOR
FOR Select Statement;

-- T-SQL
DECLARE CursorName CURSOR [CursorOptions]
FOR Select Statement;
```

## 2. Open

```
-- Open to retrieve data
OPEN CursorName;
```

## 3. Fetch

```
/* Moves to the next row and assigns the values from each column returned by
the cursor into a local variable */
FETCH [Direction] CursorName [INTO @Variable1, @Variable2, ...];
-- T-SQL offers @@fetch_status function to report the state of the cursor after
the last FETCH command (0: OK; -1: Fail, end off record set; -2: Fail, tuple not available)
```

## 4. Close

```
-- Close cursor. Can be opened again (2.Open)
CLOSE CursorName;
```

## 5. DEALLOCATE

```
-- Release cursor.
DEALLOCATE CursorName;
```

# Cursor - Exemplo

**Objetivo: Número de produtos distintos e total absoluto encomendados por cada cliente**

```

DECLARE @custID as nchar(5), @prevCustID as nchar(5), @prodID as int,
        @qty as int, @totalQty as int, @cnt as smallint;

DECLARE C CURSOR FAST_FORWARD
FOR SELECT CustomerID, ProductID, Quantity FROM CustOrderProducts ORDER BY CustomerID;

OPEN c;

FETCH C INTO @custID, @prodID, @qty;

SELECT @prevCustID = @custID, @totalQty = 0, @cnt = 0;

WHILE @@FETCH_STATUS = 0
BEGIN
    if @prevCustID <> @custID
    BEGIN
        PRINT @prevCustID + ' - ' + CAST(@cnt as varchar) + ' - ' + CAST(@totalQty as varchar);
        SELECT @prevCustID = @custID, @totalQty = 0, @cnt = 0;
    END;

    SET @totalQty += @qty;
    SET @cnt += 1;

    FETCH C INTO @custID, @prodID, @qty;
END;

CLOSE C;

DEALLOCATE C;

```

**CustOrdersProducts**

CustomerID	ProductID	Quantity
VINET	11	12
VINET	42	10
VINET	72	5
TOMSP	14	9
TOMSP	51	40
HANAR	41	10
HANAR	51	35

# Exemplo - Implementação Alternativa

## Objectivo:

Implementação Alternativa com Consulta baseada em Álgebra Relacional

```
SELECT CustomerID, count(ProductID) as nprod, sum(Quantity) as totalQty
FROM CustOrderProducts
GROUP BY CustomerID
ORDER BY CustomerID;
```

## Resultados

236 ms

pesca à linha

ALFKI	-	12	-	174
ANATR	-	10	-	63
ANTON	-	17	-	359
AROUT	-	30	-	650
BERGS	-	52	-	1001
BLAUS	-	14	-	140
BLONP	-	26	-	666
BOLID	-	6	-	190
BONAP	-	44	-	980

7 ms

pesca com rede

CustomerID	nprod	totalQty
ALFKI	12	174
ANATR	10	63
ANTON	17	359
AROUT	30	650
BERGS	52	1001
BLAUS	14	140
BLONP	26	666
BOLID	6	190
BONAP	44	980

# When are cursors the best solution?

- **Iterating over a stored procedure:** When a stored procedure must be executed several times, once for each row or value, and the stored procedure can't be refactored into a set-based solution, or it's a system stored procedure, then a cursor is the right way to iteratively call the stored procedure.
- **Iterating over DDL code:** When DDL code must be dynamically executed multiple times, using a cursor is the appropriate solution.
  - Sometimes it's necessary to iterate over multiple rows or columns, generating a dynamic SQL statement for each row or column.
- **Cumulative Totals/Running Sums:** While there are set-based solutions, a cursor is the best-performing solution in these cases because it only has to add the next row's value to the cumulative value.
- **Time-Sensitive Data:** Some time-sensitive problems, depending on the database design, can benefit by using a cursor to determine the duration between events. Like the cumulative totals problem, time-sensitive data requires comparing the current row with the last row. Although there are possible set-based solutions, in some cases I've seen cursors perform better than set-based solutions.

# Cursor - [CursorOptions]

**Static:** Copies all the data into tempdb and the cursor iterates over the copy of the data. Any changes (inserts, updates, or deletes) to the real data are not seen by the cursor. This type of cursor is generally the fastest.

**Keyset:** Only the minimum number of columns needed to identify the rows in the correct order are copied to tempdb. The cursor walks through the data by internally joining the keyset table in tempdb with the real data. Updates and deletes are seen by the cursor, but not inserts. This is the only cursor type that experiences deleted rows as @@fetch\_status = -2, so be sure to test for deleted rows.

Keyset cursors, compared to static cursors, write less to tempdb when creating the cursor set, but they must perform most of the cursor SELECT statement for every fetch. Therefore, if the SELECT statement used to define the cursor references several data sources, avoid keyset cursors.

**Dynamic:** The cursor iterates over the original real data. All changes are seen by the cursor without any special handling of the changes. If a row is inserted after the cursor location, then the cursor will see that row when the cursor reaches the new row. If a row is deleted, then the cursor will simply not see the row when it reaches where the row had been.

**Fast\_Forward:** This is the “high-performance” cursor option introduced in SQL Server 2000. Basically, it’s a read-only, forward-only dynamic cursor.

# Stored Procedures

Procedimientos

# Stored Procedure - Definição

- Trata-se de uma **batch armazenada** com um **nome**.
  - Um conjunto de instruções T-SQL que o SQL Server compila num *single execution plan*.
- O SQL Server **não tem de recompilar o código** cada vez que o procedimento é invocado.
- Os procedimento são guardados em **memória cache** na primeira vez em que são executados.
  - Execução mais rápida
- O procedimento pode:
  - Ter parâmetros de entrada
  - Ter valor de retorno (parâmetros de saída, *return success* ou *failure status messages*)
  - Devolver um conjunto de registos (tuplos)

# Stored Procedure - Mais Valias

- **Extensibility**: Using stored procedures is the best means of abstracting, or decoupling, the database. A stored procedure API contract will encapsulate the database and provide it with long-term extensibility.
- **Performance**: A well-written stored procedure is the fastest possible SQL Server code, it keeps the execution of data-centric code close to the data, and it's easier to index tune a database with stored procedures.
- **Usability**: It's easier for application programmers to make a stored procedure call and consume the result than it is to write ad hoc SQL.
- **Data Integrity**: A stored procedure developed by the database developer is less likely to contain data integrity errors, and easier to unit test, than ad hoc SQL code.
- **Security**: Locking down the tables and providing access only through stored procedures is a standard best practice for database development.



# SQL Statement vs. Stored Procedure

## SQL Statement

### First Time

- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

### Second Time

- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

⋮

## Stored Procedure

### Creating

- *Check syntax*
- *Compile*

### First Time

- *Execute*
- *Return data*

### Second Time

- *Execute*
- *Return data*

⋮

# Stored Procedure - Create

## Sintaxe:

```
CREATE PROC[EDURE] procedure_name  
[ @parameter_name data_type] [= default] OUTPUT][,...,n]  
AS  
T-SQL_statement(s)
```

- Única instrução da batch

```
-- Exemplo: CREATE Storage Procedure  
--          devolver um conjunto de registos (record-set)
```

```
go  
CREATE PROCEDURE dbo.CategoryList  
AS  
SELECT ProductCategoryName, ProductCategoryDescription  
FROM dbo.ProductCategory;  
go
```

# Stored Procedure - Create com Parâmetros

-- Exemplo: CREATE Storage Procedure with input parameters

```
CREATE PROC Department_Members @DeptName varchar(50)
AS
    SELECT Dep_Name, COUNT(Emp_ID) NumberOfMember
    FROM Departments D, Employees E
    WHERE D.Dep_ID = E.Dep_ID and Dep_Name = @DeptName
    GROUP BY Dep_Name
```

Devolve um record-set

-- 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
```

Devolve um record-set

?

Nota: Quando temos múltiplos parâmetros de entrada devemos colocar no fim aqueles que têm valor de defeito.

# Stored Procedure - Update e Drop

## Sintaxe:

```
ALTER PROC[EDURE] procedure_name  
[ @parameter_name data_type] [= default] [OUTPUT] [,...,n]  
AS  
T-SQL_statement(s)
```

- Substitui o procedimento existente com o novo código (T-SQL\_statement(s))

## Sintaxe:

```
DROP PROC[EDURE] procedure_name
```

- Elimina um procedimento

```
-- if exists, delete the procedure
```

```
IF Object_Id('Production.ProductList', 'P') IS NOT NULL  
DROP PROCEDURE Production.ProductList;
```

# Stored Procedures - Tipos

- **System** stored procedure:
  - Nome começa com **sp\_**
  - Criados na Master database
  - Podem ser utilizados em qualquer base de dados
    - **Muitas vezes utilizados por sysadmins**
- **Local** stored procedure:
  - São definidos num base de dados local
  - Nome livre mas recomenda-se uma normalização por parte do utilizador
    - **Aumenta a legibilidade**
    - **Exemplos: pr\_ , p\_, ...**

# Stored Procedures - Execução

Sintaxe:

```
EXEC[CUTE] procedure_name [@parameter_name data_type]
```

## -- 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
```

# Stored Procedure – Parâmetros de Saída

- Utilizados para retornar *non-recordset information*.
- Devemos criar previamente a variável que receberá o valor de parâmetro de saída.

-- 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;
```

# Stored Procedures – Return [N]

- Termina incondicionalmente o procedimento e retorna um inteiro
  - tipicamente: success/failure status
- O valor de saída pode ser atribuído a uma variável:  
`EXEC @LocalVariable = StoredProcedureName;`

## -- 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;
```

Também podemos ter um return sem valor de retorno



# T-SQL Error Handling

- T-SQL oferece um conjunto de ferramentas para detecção e tratamento de erros.

**@@error**: retorna um inteiro com o código de erro da última instrução. 0 - Sucesso

```
-- Exemplo: @@error
```

```
UPDATE Person SET PersonID = 1 Where PersonID = 2;  
Print @@error;      -- Violation of PRIMARY KEY constraint 'PK Person...'  
Print @@error;      -- 0
```

**@@rowcount**: permite saber quantos tuplos foram afectadas por determinada instrução SQL

```
-- Exemplo: @@rowcount
```

```
UPDATE Person SET LastName = 'Johnson' WHERE PersonID = 100;  
IF @@rowCount = 0  
    PRINT 'no rows affected';
```

# T-SQL RAISERROR

- Retorna uma mensagem de erro ao cliente

## Duas Sintaxes:

RAISERROR ErrorNumber ErrorMessage;

```
-- Exemp1: RAISERROR
```

```
RAISERROR 12345 'Nao foi possivel atualizar registro';
```

RAISERROR ( message or number, severity, state, optional arguments )  
WITH LOG;

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

```
-- Exemplo: RAISERROR
```

```
RAISERROR ('Nao foi possivel atualizar registro em %s.', 14, 1, 'Customer');
```

```
Msg 50000, Level 14, State 1, Line 1  
Nao foi possivel atualizar registro em Customer.
```

# T-SQL: Try ... Catch

- Captura e Tratamento de Erros

BEGIN TRY

<SQL code>;

END TRY

BEGIN CATCH

<error handling code>;

END CATCH;

-- Exemplo

```
GO
CREATE PROCEDURE uspTryCatchTest
AS
BEGIN TRY
    SELECT 1/0
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;
GO

EXEC uspTryCatchTest;
```

ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
8134	16	1	uspTryCatchTest	4	Divide by zero error encountered.

(1 row(s) affected)

# Stored Procedures - Cifragem

- SQL Server permite ver a definição (conteúdo) do procedimento:

```
EXEC sp_helptext 'dbo.CategoryList';
```

```
Text
-----
CREATE PROCEDURE  CategoryList
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM  dbo.ProductCategory;
```

- Existe a opção de cifrar o conteúdo do SP:

```
-- Exemplo: Cifrar Storage Procedure criado anteriormente
```

```
ALTER PROCEDURE  dbo.CategoryList
WITH ENCRYPTION
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM  dbo.ProductCategory;
```

```
The text for object 'dbo.CategoryList' is encrypted.
```

# User Defined Functions (UDF)

Funções Definidas pelo Utilizador

# UDF - Vantagens

- Os mesmos benefícios dos Stored procedures
  - São igualmente compilados e otimizados
- Podem ser utilizadas para incorporar lógica complexa dentro de uma consulta.
- Oferecem os mesmo benefícios das vistas pois podem ser utilizados como fonte de dados nas consultas e nas cláusulas WHERE/HAVING.
  - Acresce o facto de aceitar parâmetros, algo impossível em views.
- Criação de novas funções contendo expressões complexas.

## UDF - Tipos

SQL Server suporta 3 tipos de UDFs:

- Escalares
- Inline table-valued
- Multi-statement table-valued functions

# UDF Escalar

## Sintaxe:

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

- Aceitam múltiplos parâmetros.
- Retornam um único valor.
  - Instrução RETURN
- Podem ser utilizados dentro de qualquer expressão T-SQL, incluindo check constraint.



# UDF Escalar - Exemplos

## -- Exemplos: Criação e Utilização de UDF Escalar

```
CREATE FUNCTION dbo.Revenue_Day (@Date datetime) RETURNS money
AS
BEGIN
    DECLARE @total money
    SELECT @total = sum(sali_Quantity * sali_price)
    FROM Sales_Orders s, Sales_Orders_Items si
    WHERE s.sal_number = si.sal_number and year(sal_date) = year(@Date)
        and month(sal_date) = month(@Date) and day(sal_date) = day(@Date)
    RETURN @total
END

GO
SELECT dbo.Revenue_Day(GETDATE())
```

```
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

Nota: O nome da schema (**dbo**) é obrigatório na invocação da UDF

# UDF Escalares - Algumas Limitações

- Determinísticas
  - Os mesmos parâmetros de entrada produzem o mesmo valor de retorno.
  - Não são permitidas funções não-determinísticas dentro das UDF.
    - `newid()`, `rand()`, etc
- Não são permitidos updates à base de dados ou invocação do comando DBCC.
- Em termos de valor de retorno não permite:
  - BLOB (binary large object) - text, ntext, timestamp, image data-type, etc.
  - Table variables
  - Cursores
- Não permite TRY...CATCH ou RAISERROR.
- Recursividade limitada a 32 níveis.

# UDF - Inline Table-valued

## Sintaxe:

```
CREATE FUNCTION function_name  
[ @param_name data_type ] [= default] [ READONLY ][,...,n]
```

**RETURNS TABLE**

AS

```
T-SQL_statement {RETURN SELECT statement}
```

- Similares a vistas
  - Ambas são wrappers para construções SELECT
  - Tem as mais valias das vistas acrescido do facto de suportar parâmetros de entrada.

# UDF Inline Table-valued - Exemplo

-- Exemplos: Criação e Utilização de UDF Inline Table-valued

```
CREATE FUNCTION dbo.AveragePricebyItems (@price money = 0.0) RETURNS Table
AS
```

```
    RETURN (SELECT Ite_Description, Ite_Price
              FROM Items
              WHERE Ite_Price > @price)
```

```
GO
```

```
SELECT * FROM dbo.AveragePricebyItems (15.00)
```

```
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

## Sintaxe:

```
CREATE FUNCTION function_name  
[ @param_name data_type ] [= default] [ READONLY ][,...,n]  
RETURNS @return_variable TABLE <table_type_definition>  
AS  
T-SQL_statement
```

- **Combina** a capacidade das funções **escalares** (conter código complexo) com a capacidade das **inline table-valued** (retornar um conjunto).
- Cria uma *table variable*, introduz-lhe tuplos e retorna-a.
  - Tabela retornada pode ser utilizada num SELECT

# UDF Multi-statement Table-Valued - Exemplo

-- Exemplos: Criação e Utilização de UDF Multi-statement Table-Valued

```
CREATE FUNCTION dbo.AveragePricebyItems2 (@price money = 0.0) RETURNS @table TABLE
    (Description varchar(50) null, Price money null)
AS
BEGIN
    INSERT @table SELECT Ite_Description, Ite_Price
        FROM Items WHERE Ite_Price > @price;
    RETURN;
END;

GO
SELECT * FROM dbo.AveragePricebyItems2 (15.00);
```

```
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();
```

# SP versus UDF

- return - zero, single or multiple values
- input/output param
- **cannot** use SELECT/ WHERE/ HAVING statement
- call SP - **OK**
- exception handling - **OK**
- transactions - **OK**
- return - single value (scalar or table)
- input param
- **can** use SELECT/ WHERE/ HAVING statement
- call SP - **NOK**
- exception handling - **NOK**
- transaction - **NOK**

# Trigger



# Trigger - Definição

- Trigger: um tipo especial de stored procedure que é executado em determinadas circunstâncias (eventos) associadas à manipulação de dados.
- SQL Server suporta dois tipos de trigger: DML e DDL. Só vamos tratar de triggers DML:
  - São criados em tabelas (ou vistas) e têm uma ou mais ações associadas (INSERT, UPDATE, DELETE).
- Quando ocorre uma das ações previstas, os triggers são “disparados” (executados).
- Exemplos de uso:
  - Maintenance of duplicate and derived data
  - Complex column constraints
  - Cascading referential integrity
  - Complex defaults
  - Inter-database referential integrity

# Trigger - Conceitos Básicos

- SQL Server triggers são disparados uma vez por cada operação de modificação de dados
  - Não por tuplo afectado - caso da Oracle.
- Ter em atenção que os triggers estendem a duração da transação:
  - Pode criar problemas de locks/blocks em sistemas de elevado desempenho.
  - Compromisso entre integridade dos dados e potencial impacto no desempenho.
- Existem dois tipos de DML triggers que diferem quanto ao propósito, timing e efeito.
  - **instead of**
  - **after**

# SQL Server - Transaction Flow

É importante entender em que parte da transação ocorre cada um dos triggers...

1. IDENTITY INSERT check
2. Null ability constraint
3. Data-type check
4. **INSTEAD OF** trigger execution.

If an INSTEAD OF trigger exists, then execution of the DML stops here.

INSTEAD OF triggers are not recursive. Therefore, if the INSERT trigger executes another DML command, then the INSTEAD OF trigger will be ignored the second time around.

5. Primary-key constraint
6. Check constraints
7. Foreign-key constraint
8. DML execution and update to the transaction log
9. **AFTER** trigger execution
10. Commit transaction

# Transaction Flow - Ideias a reter...

- **AFTER** trigger pode assumir que os dados passaram todos as verificações de integridade de dados.
- **AFTER** trigger ocorre depois de todos os constraints
  - Não pode corrigir eventuais problemas dos dados.
- **AFTER** trigger ocorre antes do *commit*<sup>1</sup> da transação DML. Assim podemos fazer o *rollback*<sup>1</sup> da transação se os dados forem inaceitáveis.
- **INSTEAD OF** trigger - a transação para no ponto 4 e nenhum dos posteriores é executado, incluindo a instrução DML.
  - Exceção: Invocação recursiva do trigger
- **INSTEAD OF** trigger pode “contornar” problemas de integridade referencial mas não de nulidade, tipo de dados e identidade das colunas.

<sup>1</sup> Vamos ver o que isto é mais à frente quando se falar de Transações

# Trigger - Create, Enable/Disable, Drop

## Sintaxe:

-- 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>
```

# Trigger - After

- Podemos ter vários triggers after por tabela.
- Algumas das utilizações possíveis:
  - Processos complexos de validação de dados envolvendo, por exemplo, várias tabelas
  - Assegurar regras de negócios complexas.
  - Efetuar auditorias aos dados.
  - Atualizar campos calculados.
  - Assegurar verificações de integridade referencial definidas pelo utilizador e deletes em cascata
    - Devemos evitar, i.e. privilegiar a integridade referencial declarativa, a menos que não exista outra forma.
    - Exemplo: Especialização - Subcategorias exclusivas. Uma Pessoa só pode ser Aluno ou Professor.

# Trigger - Exemplo de After

-- Exemplos: Criação e teste de um trigger After Insert ou Update.

```
GO
CREATE Trigger highsales ON dbo.[Order Details]
AFTER INSERT, UPDATE
AS
    SET NOCOUNT ON;

    DECLARE @total as real
    SELECT @total = unitprice * (1-discount) * quantity FROM inserted;
    IF @total < 0.99
        BEGIN
            RAISERROR ('Encomenda nao processada. Valor muito baixo', 16,1);
            ROLLBACK TRAN;
            -- Anula a inserção
        END
    ELSE IF @total > 1000
        PRINT 'Log: Encomenda de valor elevado'
GO
```

OrderID	ProductID	UnitPrice	Quantity	Discount
10248	11	14,00	12	0
10248	42	9,80	10	0
10248	72	34,80	5	0
10249	14	18,60	9	0
10249	51	42,40	40	0
10250	41	7,70	10	0
10250	51	42,40	35	0,15
10250	65	16,80	15	0,15
10251	22	16,80	6	0,05



```
INSERT INTO dbo.[Order Details] values (10248, 14, 18.6, 20, 0.15);
```

{1 row(s) affected}

```
INSERT INTO dbo.[Order Details] values (10248, 14, 18.6, 200, 0.15);
```

Log: Encomenda de valor elevado  
{1 row(s) affected}

```
INSERT INTO dbo.[Order Details] values (10248, 14, 1.0, 1, 0.15);
```

Msg 50000, Level 16, State 1, Procedure highsales, Line 13  
Encomenda nao processada. Valor muito baixo  
Msg 3609, Level 16, State 1, Line 1  
The transaction ended in the trigger. The batch has been aborted.

# Trigger - Instead of

- Apenas um por tabela (vista).
- NÃO É EXECUTADA a ação associada (Insert, Update, Delete).
  - Fica à responsabilidade do trigger efetuar a operação pretendida (ou não).
- Devemos utilizar este tipo de trigger quando sabemos que a ação (instrução DML) tem uma elevada probabilidade de ser *rolled back* e pretendemos que outra lógica seja executada em vez (*instead of*) dela.
  - Exemplos:
    - Uma instrução tenta fazer update de uma view non-updatable
    - Uma instrução tenta apagar um tuplo mas pretendemos que este passe para uma tabela de arquivo.



# Trigger - Exemplo 1 de Instead of

-- Exemplos: Criação, Teste e Eliminação de um trigger Instead of Insert

```
GO
CREATE TRIGGER dbo.TriggerTest ON dbo.dependent
INSTEAD OF INSERT
AS
    PRINT 'Insert Action Canceled';
GO
```

```
INSERT INTO dependent VALUES('21312339', 'Catia Pereira', 'F', null, null);
GO
```

Insert Action Canceled

```
SELECT * FROM dependent WHERE essn= '21312339';
GO
```

Essn	Dependent_name	Sex	Bdate	Relationship
-----				
(0 row(s) affected)				

```
DROP Trigger dbo.TriggerTest;
```

# Trigger - Exemplo 2 de Instead of

-- Exemplos: Instead of - Constraint: employee cannot work in projects associated to distinct PLocations

```
CREATE TRIGGER dbo.TriggerTest2 ON works_on
INSTEAD OF INSERT
AS
BEGIN
    IF (SELECT count(*) FROM inserted) = 1
    BEGIN
        DECLARE @issn as char(9);
        DECLARE @ipno as int;
        DECLARE @iplocation as varchar(15);
        SELECT @issn = essn, @ipno = pno FROM inserted;
        SELECT @iplocation=plocation from project where pnumber=@ipno;

        IF (@iplocation) is null
            RAISERROR('Project Inexistent.', 16, 1);
        ELSE
            BEGIN
                -- You can have different Pno with same Plocation
                IF (SELECT count(distinct Plocation) FROM Project join Works_on on Pno=Pnumber
                    WHERE essn=@issn AND plocation<>@iplocation) >= 1
                    RAISERROR('Not allowed to have employee working in Projects with different PLocations.', 16, 1);
                ELSE
                    INSERT INTO works_on SELECT * FROM inserted;
                    -- chamada recursiva
            END
        END
    END
END
GO
```

```
insert into project values('Aveiro Digital', 1, 'Aveiro', 3);
insert into project values('BD Open Day', 2, 'Espinho', 2);
insert into project values('Dicoogle', 3, 'Aveiro', 3);
```

{1 row(s) affected}

```
insert into works_on values('183623612', 1, 20);
insert into works_on values('183623612', 2, 20);
insert into works_on values('183623612', 3, 10);
```

Msg 50000, Level 16, State 1, Procedure TriggerTest2, Line 10  
Not allowed to have employee working in Projects with different PLocations.

{1 row(s) affected}

```
SELECT * FROM works_on WHERE essn='183623612';
```

Essn	Pno	Hours
183623612	1	20.0
183623612	3	10.0

# Inserted e Deleted - Logical Tables

- O SQL Server permite ter acesso a duas tabelas lógicas com uma imagem read-only os dados afectados:

- Inserted
- Deleted

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

- Estas tabelas tem um scope muito limitado
  - Stored procedures invocados pelo trigger não as vêem
- A maioria dos triggers implementados não foram pensados para eventos que afectam vários tuplos.
  - Na prática, estas situações acabam por estar associadas a situações de mau desempenho dos triggers.

# Trigger - Colunas Alteradas

- O SQL Server disponibiliza duas funções que nos permitem saber quais as colunas (potencialmente) afectadas pela instrução DML:
  - `update(<columnname>)`
    - Retorna true se determinada coluna for alterada.

```
CREATE Trigger detectcontactupdate ON dbo.[Customers]
AFTER UPDATE
AS
    IF update(ContactName)
        PRINT 'Mudou a pessoa de contacto do cliente.'
```

- `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.

# Triggers - Limitações

- Instruções não permitidas num trigger:
  - CREATE, ALTER, or DROP database
  - RECONFIGURE
  - RESTORE database or log
  - DISK RESIZE
  - DISK INIT

# Trigger - Funcionalidades Úteis

- Ver conteúdo do trigger
  - `sp_helptext <trigger name>`

```

Text
-----
CREATE TRIGGER dbo.TriggerTest2 ON works_on
INSTEAD OF INSERT, UPDATE
AS
BEGIN
    IF (SELECT count(*) FROM inserted) = 1
    BEGIN
        DECLARE @issn as char(9);
        DECLARE @ipno as int;
        DECLARE @iplocation as varchar(15);
        SELECT @issn = essn,
               @ipno = pno FROM inserted;
        SELECT @iplocation=plocation from project where pnumber=@ipno;

        IF (@iplocation) is null
            RAISERROR('Project Inexistent.', 16, 1);
        ELSE
            BEGIN
                -- You can have different Pno with same Plocation...

                IF (SELECT count(distinct Plocation) FROM Project join Works_on on Pno=Pnumber WHERE essn=@issn AND plocation<>@iplocation) >= 1
                    RAISERROR('Not allowed to have employee working in Projects with different Plocations.', 16, 1);
                ELSE
                    INSERT INTO works_on SELECT * FROM inserted;
            END
        END
    END
END

```

- Listar triggers de uma tabela
  - `sp_helptrigger <table name>`

trigger_name	trigger_owner
TriggerTest2	dbo

(1 row(s) affected)

# Resumo

- Script e Batch
- Cursor
- Stored Procedure
- User Defined Function
- Trigger