

# Aspects avancés avec SQL (en T-SQL)

# Introduction

Pourquoi le T-SQL

Finalité : le trigger

Caractéristiques du langage

L'environnement de développement

Réponses du serveur

# Pourquoi le T-SQL ?

---

- Faire des requêtes, c'est bien, les automatiser, c'est mieux !
  - > *Procédures*
  - > *Fonctions*
  - > *Triggers*
  - > ...
- Langage directement utilisable dans les bases de données, sans

utilisation de langages externes

- Souvent dénigré, car méconnu, mais peut réellement alléger les

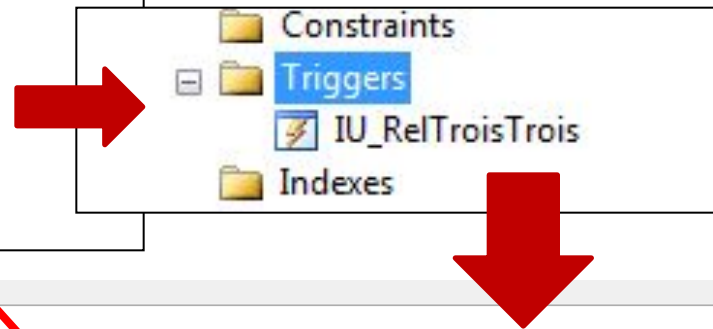
codes externes appelant les BD, optimisant grandement le traitement

# Finalité : le trigger

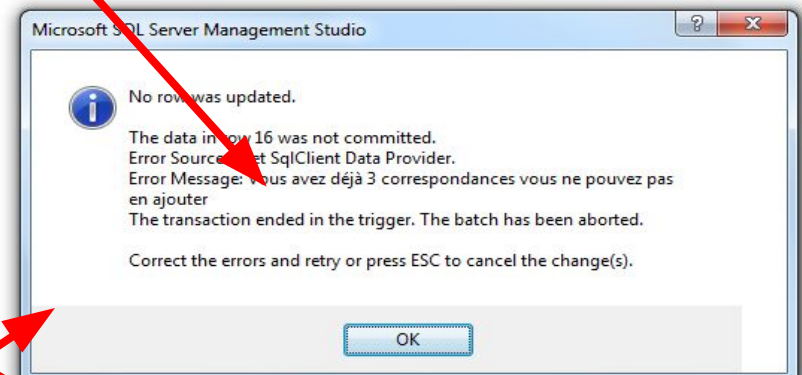
```
CREATE TRIGGER [dbo].[IU_RelTroisTrois] ON [dbo].[Correspondance]
AFTER INSERT, UPDATE
AS
declare
    @nbele integer
BEGIN
    SELECT @nbele = count(*) FROM Correspondance c , inserted i WHERE c.Type_Form_Loc = i.Type_Form_Loc

    if(@nbele>3)
    begin
        RAISERROR 13000 'Vous avez déjà 3 correspondances vous ne pouvez pas en ajouter'
        ROLLBACK
    end

    else
    begin
        COMMIT
    end
end
END
```



FORMA1100\SQLT....Correspondance	
Class_Tarificati...	Type_Form_Loc
1	Travel
1	Business
2	Business
2	Transport
1	Classic
2	Classic
3	Travel
3	Classic
3	Business
4	Transport
4	Classic
4	Travel
5	Transport
5	Business
5	Travel
2	Travel
* NULL	NULL



# Caractéristiques du langage (1/4)

- Langage construits par « blocs ». La structure « **BEGIN ... END** » viendra délimiter les instructions telles que « IF », « While », « GOTO » et « WAITFOR »

```
CREATE TRIGGER AD_RelTroisTrois ON Correspondance
AFTER DELETE
AS
declare
    @nbele integer
BEGIN
    /*Vérification pour l'update et l'insert*/
    SELECT @nbele = count(c.Class_Tarification_Code_Classe_Tarif) FROM Correspondance c , DELETED
    WHERE c.Class_Tarification_Code_Classe_Tarif = d.Class_Tarification_Code_Classe_Tarif

    if (@nbele < 3)
        begin
            RAISERROR 13000 'Si vous supprimez cette entrée, vous serez à un nombre inférieur
            à trois correspondances ==> INTERDIT'
            ROLLBACK
        end
    else
        begin
            COMMIT
        end
    end
END
```

# Caractéristiques du langage (2/4)

---

- En T-SQL, un script est considéré comme une procédure anonyme à part entière. Il n'est donc pas nécessaire d'être dans un bloc procédural pour déclarer des variables, exécuter des boucles, etc.
- Les points-virgules et autres annotations habituelles de programmation ne sont pas obligatoires, mais ne créent pas d'erreur et permettent de délimiter tout de même les instructions
- Donner une en-tête à un ensemble d'instructions permet de créer des objets stockés dans la base de données (procédures, fonctions, triggers)

# Caractéristiques du langage (3/4)

---

```
CREATE DATABASE DB_TEST
GO

USE DB_TEST
GO

CREATE TABLE TABLE_TEST (
...
)
GO
```

- La commande GO permet de forcer immédiatement l'exécution de certaines instruction dont l'exécution est lancée par lots (comme c'est le cas dans les procédures, fonctions et triggers, hors manipulations de transactions)  
Sans le GO, le serveur est normalement autorisé à exécuter les ordres qui lui sont proposés d'exécuter simultanément dans l'ordre qu'il le désire

- Le symbole # précède toute table temporaire que l'on voudrait utiliser

```
SELECT CURRENT_TIMESTAMP as DateTime INTO #tempT

SELECT * FROM #tempT
```

# Caractéristiques du langage (4/4)

- La commande « EXEC » permet d'exécuter une commande particulière complexe, construite à partir d'un ensemble d'ordre préalables et stockés dans une variable

```
DECLARE @SQL VARCHAR(10000)

Set @SQL = 'SELECT * FROM ' + @nom_table
          + ' WHERE '        + @liste_des_colonnes
          + ' LIKE ' + '%' + @mot_a_rechercher + '%''

Exec (@SQL)
```

- L'instruction « **SET** » permet de donner une valeur aux variables ainsi que de paramétrer certaines options de la base de données

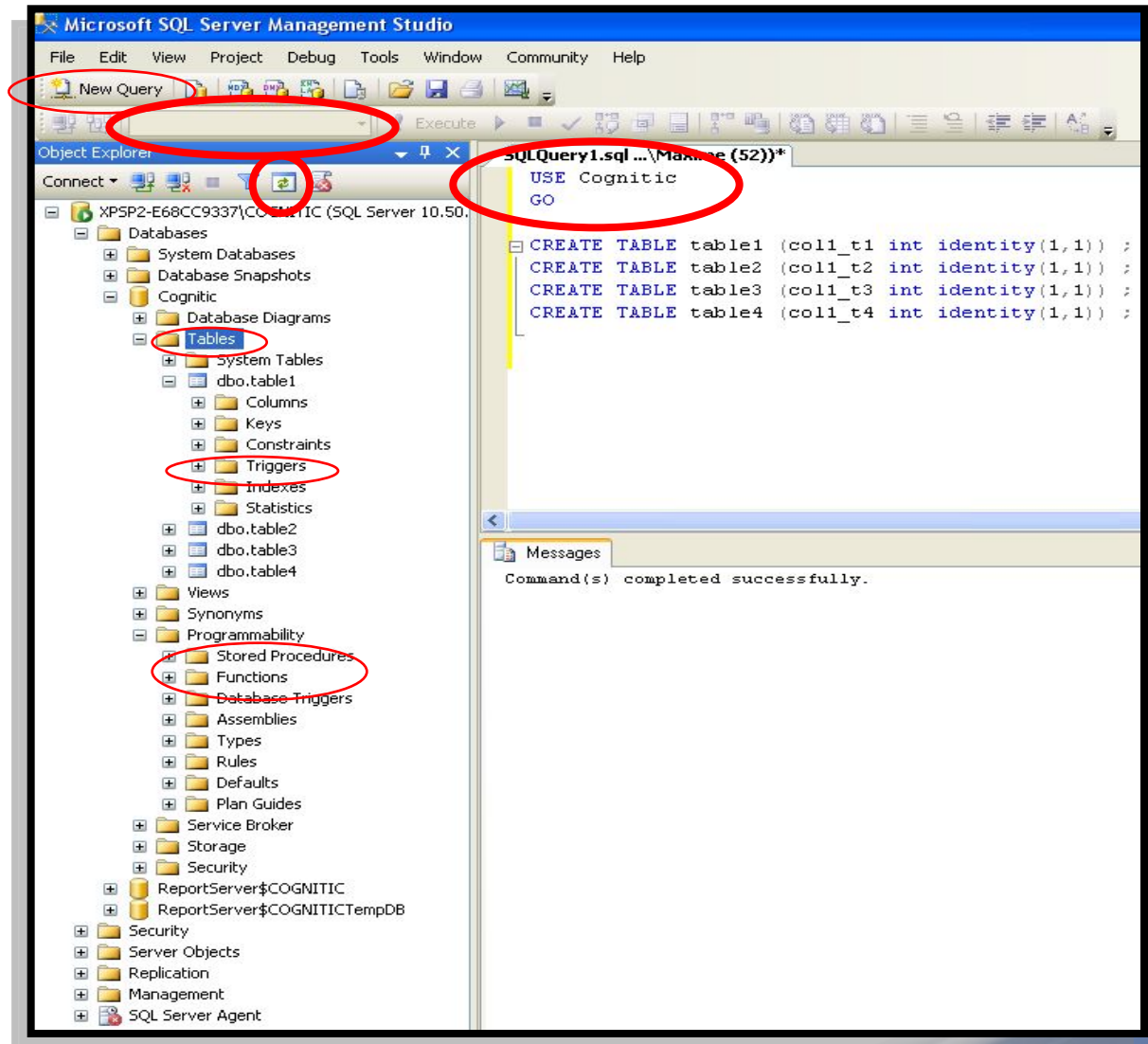
```
SET IDENTITY_INSERT [nom_table] ON/OFF -- Permet d'activer/désactiver l'auto-incrémentation dans la table spécifiée

SET DATEFORMAT {format_de_date}       -- Permet de modifier le format de la date par défaut du système
```



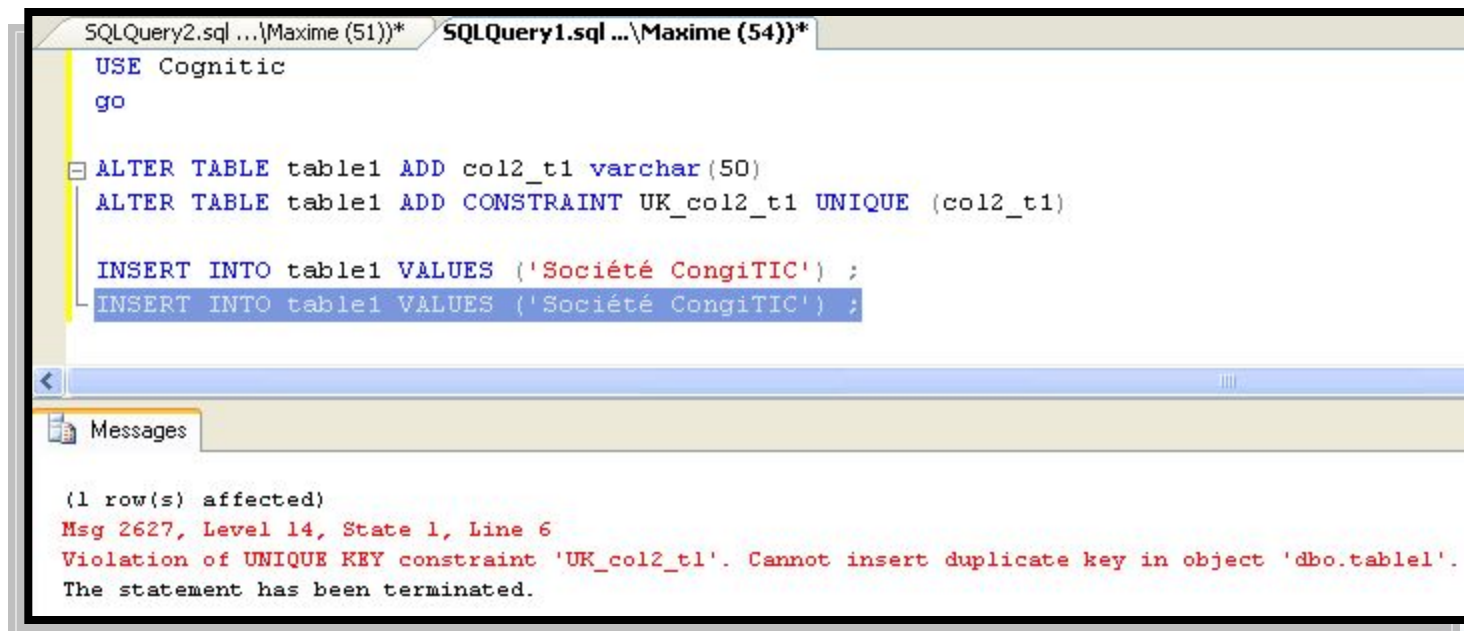
# L'environnement de développement

- SQL Server Management Studio (client Microsoft pour son SQL Server)
- **ATTENTION** à toujours se positionner sur la bonne base de données !
- Utilisation du bouton « **REFRESH** » pour faire apparaître les objets créés
- Les scripts ne sont connus du serveur que lors de leur lecture, ils ne sont pas stockés en tant qu'objets !



# Réponses du serveur (1/2)

- Lorsqu'une requête est envoyée vers le serveur via l'interface d'édition des requêtes (**New Query**), le serveur nous informe du résultat de cette requête, au bas de la page. Un texte en rouge nous indique une erreur. Double-cliquer sur le texte de l'erreur pour surligner immédiatement l'endroit où elle a été rencontrée dans le code.



The screenshot shows the SQL Server Enterprise Manager interface. At the top, there are two tabs: 'SQLQuery2.sql ...\Maxime (51))\*' and 'SQLQuery1.sql ...\Maxime (54))\*'. The active tab is 'SQLQuery1.sql ...\Maxime (54))\*'. The query editor contains the following SQL code:

```
USE Cognitic
go

ALTER TABLE table1 ADD col2_t1 varchar(50)
ALTER TABLE table1 ADD CONSTRAINT UK_col2_t1 UNIQUE (col2_t1)

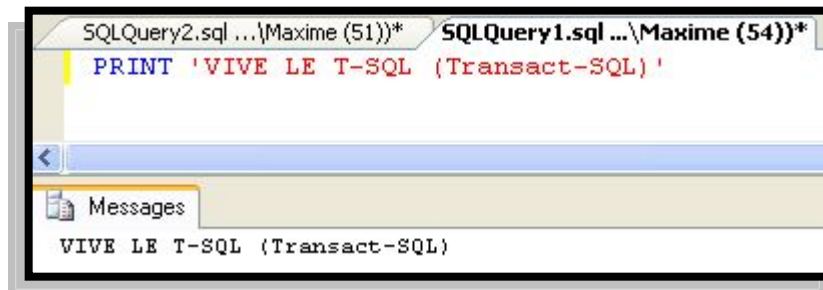
INSERT INTO table1 VALUES ('Société CongiTIC') ;
INSERT INTO table1 VALUES ('Société CongiTIC') ;
```

Below the query editor, there is a 'Messages' pane. It displays the following error message in red text:

```
(1 row(s) affected)
Msg 2627, Level 14, State 1, Line 6
Violation of UNIQUE KEY constraint 'UK_col2_t1'. Cannot insert duplicate key in object 'dbo.table1'.
The statement has been terminated.
```

# Réponses du serveur (2/2)

- Il est possible de demander l'affiche d'un texte par le serveur via l'instruction « **PRINT** »



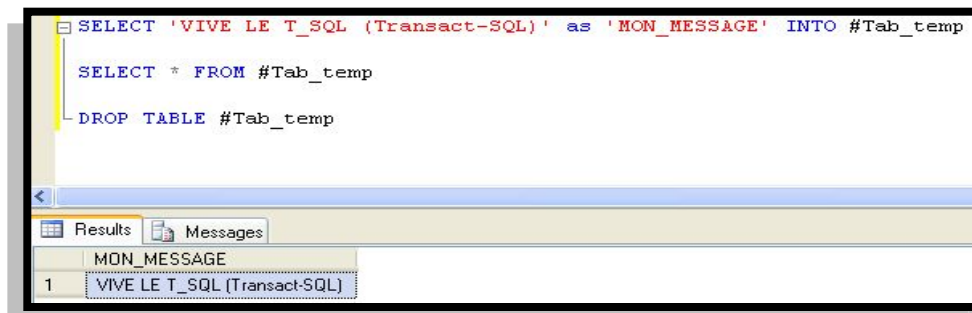
The screenshot shows a SQL query window with the following text:

```
SQLQuery2.sql ...\Maxime (51))* SQLQuery1.sql ...\Maxime (54))*  
PRINT 'VIVE LE T-SQL (Transact-SQL)'
```

Below the query window, the 'Messages' pane displays the output:

```
VIVE LE T-SQL (Transact-SQL)
```

- Une autre manière de faire peut consister également à sélectionner ce que l'on désire afficher à partir d'une table temporaire. Cependant, ces tables ne sont utilisables qu'une seule fois, il est donc nécessaire et recommandé, pour des questions de performance, de les supprimer rapidement. Il est à noter qu'un changement d'utilisateur ou de session supprime ces tables.



The screenshot shows a SQL query window with the following text:

```
SELECT 'VIVE LE T_SQL (Transact-SQL)' as 'MON_MESSAGE' INTO #Tab_temp  
  
SELECT * FROM #Tab_temp  
  
DROP TABLE #Tab_temp
```

Below the query window, the 'Results' pane displays the output:

	MON_MESSAGE
1	VIVE LE T_SQL (Transact-SQL)

# Les variables

Définition

Nature des variables

Déclaration des variables

Nom des variables

Type des variables

Affecter une valeur

CONVERT et concaténation

Le type TABLE

# Définition

*Toute donnée ou groupe de données que l'on désire **référencer** une ou plusieurs fois dans le code d'un programme est le plus souvent **stocké dans une « enveloppe » nommée** qui permet de l'appeler (de l'utiliser) plus facilement*

*Cette donnée stockée et nommée est appelé **variable***

*Une variable possède une **nature**, un **nom**, un **type** et a une ou plusieurs **valeurs**, même s'il s'agit de la valeur NULL*

- Remarquons bien que l'absence de valeur n'existe pas, cette absence à l'affichage cache toujours au moins la valeur « NULL » qui n'est pas équivalent à 0, qui est une donnée numérique réelle
- Attention à toujours donnée un nom concis et clair à vos objets ou vos variables, cela rendra le code plus facile à comprendre et plus précis !

```
CREATE TABLE MA_TABLE_UTILISATEURS_DE_MON_SYSTEME_DE_LOCATION_DE_VOITURE (..)  
CREATE TABLE T_USERS_RentACar (..)  
CREATE TABLE URAC (..)
```

# Nature des variables

---

- Variables **SCALAIRES**

>> Une variable est dite de nature scalaire si elle ne peut contenir qu'une et une seule VALEUR (un chiffre, une chaîne de caractères, etc.)

- Variables **COMPOSITES**

>> Une variable est dite composite si elle contient plusieurs valeurs différentes qu'elles soient du même type ou non.

En T-SQL, il s'agira le plus souvent d'une table temporaire

- Variables **CONTENEUR** ou **CURSEUR**

>> Les curseurs sont des mécanismes de mémoire tampons permettant d'accéder aux données renvoyées par une requête et de parcourir les lignes du résultat une par une

- A noter que le T-SQL n'aborde pas la notion de variable scalaire **CONSTANTE**

# Déclaration de variables

---

- Pour pouvoir être utilisée, une variable doit être déclarée, il faut signaler au système que l'on crée un conteneur de ce nom que nous pourrions utiliser
- La variable créée n'existe qu'au moment où elle est déclarée et utilisée. Faire tourner un nouveau script ne permet pas d'utiliser les variables précédemment déclarées
- Toute déclaration de variable commence par l'instruction « **DECLARE** ». Notez cependant qu'une seule instruction « **DECLARE** » peut déclarer d'un coup plusieurs variables séparées par des virgules
- Tout nom de variable utilisateur commencera toujours par le symbole « @ »

```
DECLARE @variable1 INT  
DECLARE @var2 INTEGER, @var3 VARCHAR(50), @var4 DATETIME
```

# Nom des variables

---

- Le nom que vous choisissiez pour vos variables est libre tant qu'il respecte les quelques règles suivantes :
  - >> **Maximum 128 caractères**
  - >> **Ils doivent commencer par une lettre ou un « underscore »**
  - >> **Les caractères spéciaux, les accents et les espaces blancs ne sont pas admis**
- Les noms que vous choisissiez sont insensibles à la casse, comme la plupart des mots utilisés en T-SQL, d'ailleurs
- Comme énoncé précédemment, il est important de garder en mémoire que les noms les plus concis et clair sont les plus utiles !



# Type des variables

---

- Toute variable doit avoir un type
- Les types utilisés sont :
  - >> bit, **int**, smallint, tinyint, **decimal**, numeric, money, smallmoney, float, real, **datetime**, smalldatetime, timestamp, uniqueidentifier, char, **varchar**, text, nchar, nvarchar, ntext, binary, varbinary, image
- Parmi les plus importants mis en gras ci-dessus :
  - INT(eger)** - définit un chiffre entier
  - DECIMAL(x,y)** - définit un chiffre décimal contenant maximum x valeurs dont y après la virgule
  - DATETIME** – une date au format **AAAA-MM-JJ HH:MM:SS.CCC**
  - VARCHAR(X)** – une chaîne de caractères contenant X caractères maximum

# Affecter une valeur (1/2)

---

- Une fois la variable déclarée, il faudra utiliser la commande « **SET** » pour lui affecter une valeur. Utilisez l'opérateur « = » pour associer la variable à la sa nouvelle valeur

```
DECLARE @variable1 INT
DECLARE @var2 INTEGER, @var3 VARCHAR(50), @var4 DATETIME

SET @variable1 = 85
SET @var2 = 500
SET @var3 = 'Jennifer'
SET @var4 = GETDATE()
```

- Par défaut, la valeur d'une variable est « *NULL* »
- Il est possible d'affecter la valeur « *NULL* » à une variable
- Les chaînes de caractères apparaissent entre simples guillemets

# Affecter une valeur (2/2)

---

- Une autre façon d'associer une valeur à une variable est d'utiliser l'opérateur « **SELECT** *@variable = colonne\_table* **FROM** ... »

Cela permet d'affecter à la variable une valeur issue d'une colonne d'une table donnée

Les insertions multiples avec le même **SELECT** sont possibles

```
SELECT variable1 = col1_t1 from table1  
SELECT var2 = COUNT(*), var3 = col2_t1 FROM table1
```

- La clause « **WHERE** » n'est pas obligatoire, cependant la requête ne peut renvoyer qu'une seule ligne de la table, les variables étant de nature scalaire
- Il est bien entendu indispensable que la variable soit du même type que la valeur que l'on désire lui associer

# CONVERT et concaténation (1/2)

---

- **Le CONVERT implicite**

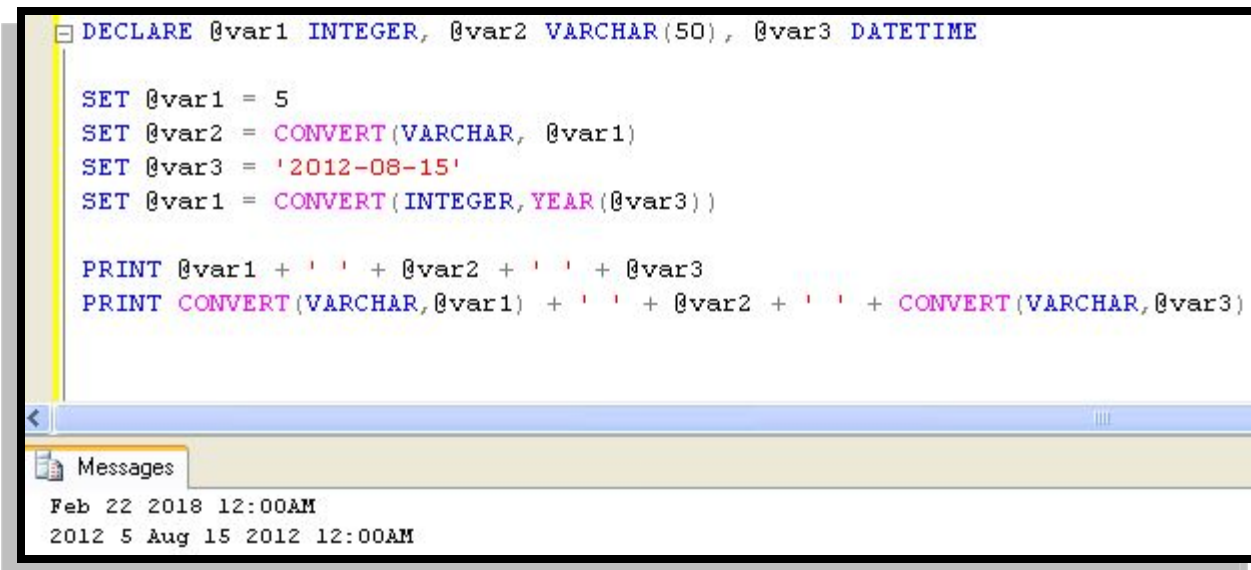
>> Si la valeur que l'on désire affecter à une variable n'est pas du même type que cette variable, SQL Server essayera de modifier le type de cette valeur pour l'adapter à celui de la variable. On dit alors qu'il y a conversion implicite de la valeur

```
CONVERT (TYPE_VERS_LEQUEL_ON_CONVERTI, @variable_a_convertir)
```

- Si la conversion implicite n'est pas possible, le système renverra une erreur. Si nous savons que cela est toutefois possible, ou afin d'être sûr que le système ne plantera pas, il est possible (et toujours conseillé) d'effectuer nous-même la conversion de valeur grâce à la fonction CONVERT

# CONVERT et concaténation (2/2)

- La concaténation en T-SQL se fait avec l'opérateur « + »  
Cela peut prêter à confusion puisque cet opérateur est également celui de l'addition. Comme SQL Server privilégiera l'addition dès qu'il rencontre un nombre, il faudra chaque fois convertir tous les termes d'une concaténation qui ne sont pas des chaînes de caractères



```
DECLARE @var1 INTEGER, @var2 VARCHAR(50), @var3 DATETIME

SET @var1 = 5
SET @var2 = CONVERT(VARCHAR, @var1)
SET @var3 = '2012-08-15'
SET @var1 = CONVERT(INTEGER, YEAR(@var3))

PRINT @var1 + ' ' + @var2 + ' ' + @var3
PRINT CONVERT(VARCHAR, @var1) + ' ' + @var2 + ' ' + CONVERT(VARCHAR, @var3)
```

Messages

Feb 22 2018 12:00AM  
2012 5 Aug 15 2012 12:00AM

# Le type TABLE (1/2)

---

- Il est également possible en T-SQL, de récupérer un ensemble de données, issues de l'une ou plusieurs colonnes d'une ou plusieurs tables de la base de données. Ce procédé sera également utile pour récolter des ensembles de données dans nos programmes
- Ces variables de type « TABLE » fonctionnent comme les tables temporaires dont le nom est précédé d'un « # » (voir précédemment), cependant, elles ont plusieurs avantages :
  - >> Elles sont automatiquement effacées après utilisation
  - >> Elles requièrent moins de traitement et d'espace mémoire que les tables temporaires
  - >> Elles sont le seul moyen d'utiliser des ordres DML sur des données temporaires dans les fonctions, procédures et triggers

# Le type TABLE (2/2)

- Exemple d'utilisation d'une variable TABLE temporaire

```
DECLARE @var_tab table (col1_tab1 INTEGER, col2_tab2 VARCHAR(50))

INSERT INTO @var_tab SELECT col1_t1, col2_t1 FROM table1
SELECT * FROM @var_tab
SELECT * FROM table1

DELETE FROM @var_tab

INSERT INTO @var_tab VALUES (1, 'Bonjour !')
INSERT INTO @var_tab VALUES (2, 'Comment')
INSERT INTO @var_tab VALUES (3, 'Allez')
INSERT INTO @var_tab VALUES (4, 'vous ?')

SELECT * FROM @var_tab
```

Results Messages

	col1_tab1	col2_tab2
1	1	Société CongiTIC

	col1_t1	col2_t1
1	1	Société CongiTIC

	col1_tab1	col2_tab2
1	1	Bonjour !
2	2	Comment
3	3	Allez
4	4	vous ?

# Les contrôles conditionnels

Fonctionnement

L'instruction IF

L'instruction CASE

L'expression CASE



# Fonctionnement

---

- Les instructions de contrôle conditionnelles et séquentielles vous permettent de **poser des conditions** dans votre code et du lui imposer de n'exécuter certaines partie que **dans des cas bien précis**. Il s'agit le plus souvent de réagir en fonction du contenu d'une variable ou du résultat d'une expression
- Les outils dont vous disposerez sont **les instructions « IF » et « CASE »**
- La condition d'un IF ou d'un CASE peut être **une variable booléenne, une constante** ou encore **une expression** qui renvoie l'une des valeurs TRUE, FALSE ou NULL
- **Les instructions** exécutées lorsque la condition renvoi la valeur TRUE peuvent être **aussi multiples que diverses**. Il est également possible d'avoir **autant d'instructions qu'on le désire** avant de fermer une conditionnelle ou de passer à la condition suivante

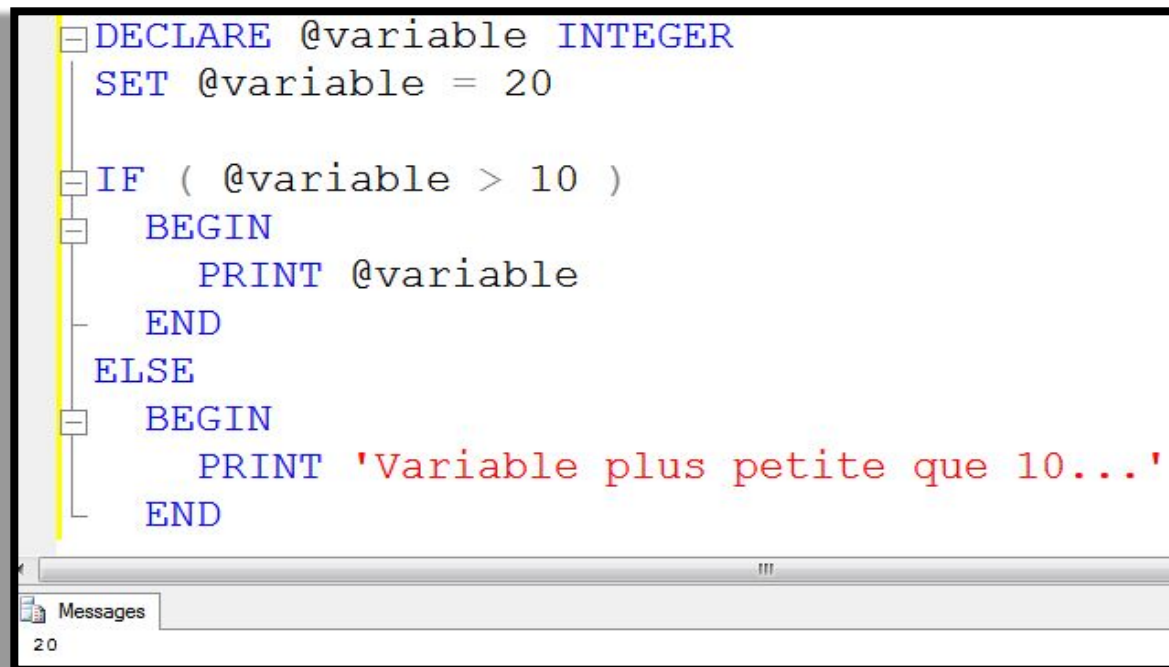
# L'instruction IF (1/2)

---

- Contrairement à certains langages, la condition du IF en T-SQL n'est pas obligatoirement mise entre parenthèses.
- Il est cependant conseillé, surtout si la condition est composée l'une combinaison de plusieurs conditions, de délimiter chaque condition par des parenthèses
- Les instructions exécutées SI la condition est vraie (TRUE) sont placées entre les balises « **BEGIN ... END** » et peuvent être aussi multiples qu'on le désire
- Il n'y a pas de clause « THEN » en T-SQL !

# L'instruction IF (2/2)

- Si la condition est fausse (FALSE) au nulle (NULL), les instructions du IF sont ignorées. Il est alors possible de passer à la clause « ELSE », qui signifie « sinon, par défaut ... ». La clause « ELSE » n'est pas du tout obligatoire



```
DECLARE @variable INTEGER
SET @variable = 20

IF ( @variable > 10 )
BEGIN
    PRINT @variable
END
ELSE
BEGIN
    PRINT 'Variable plus petite que 10...'
END
```

Messages  
20

# L'instruction CASE (1/2)

---

- Fort semblable au IF, **l'instruction CASE** permet de sélectionner une séquence d'instructions à exécuter parmi plusieurs séquences proposées. Le moteur passe en revue les expressions proposées **une à une** et exécute les instructions pour lesquelles **la valeur de l'expression est validée**
- On distinguera **deux types de CASE : le simple et le recherché**. Le premier ne prévoit qu'une égalité strictes entre les valeurs comparées alors que le second permettra des inégalités mais sera plus long à écrire
- Tout comme dans le cas du IF, le CASE (simple ou recherché) peut également contenir un **ELSE** qui ne sera exécuté que si aucune des possibilité du CASE n'est abordée. Même si le **ELSE** n'est pas obligatoire, il est toujours conseillé de le spécifier

# L'instruction CASE (2/2)

```
SELECT
CASE YEAR(BirthDate)
  WHEN 2000 THEN 'Trop jeune'
  WHEN 1990 THEN 'Jeune recrue'
  WHEN 1980 THEN 'Dans la fleur de l'âge'
  WHEN 1970 THEN 'Roule sa bosse'
  ELSE 'Valeur improbable'
END
FROM HumanResources.Employee
WHERE BusinessEntityID = 21
```

- Un **CASE recherché** évalue une liste d'expressions booléennes et, lorsqu'elle trouve une expression **qui renvoie TRUE**, exécute la **séquences d'instructions** associée à cette expression

- L'instruction **CASE simple** évalue une expression et en fonction de son résultat, exécute la liste d'instructions associée

```
SELECT
CASE
  WHEN YEAR(BirthDate) > 2000 THEN 'Trop jeune'
  WHEN YEAR(BirthDate) BETWEEN 1990 AND 2000 THEN 'Jeune recrue'
  WHEN YEAR(BirthDate) BETWEEN 1980 AND 1989 THEN 'Dans la fleur de l'âge'
  WHEN YEAR(BirthDate) BETWEEN 1970 AND 1979 THEN 'Roule sa bosse'
  ELSE 'Valeur improbable'
END
FROM HumanResources.Employee
WHERE BusinessEntityID = 21
```

# L'expression CASE

- Le problème avec les instructions case est qu'elles sont obligatoirement utilisées dans un SELECT et que de ce fait, elles ne nous renvoient aucune information à moins de stocker le résultat de la requête dans une variable de type TABLE ou encore dans une table temporaire
- L'expression CASE va nous permettre de renvoyer l'information dans une variable donnée

```
DECLARE @annee_emp VARCHAR(30)
SELECT @annee_emp = CASE
    WHEN YEAR(BirthDate) > 2000
    THEN 'Trop jeune'
    WHEN YEAR(BirthDate) BETWEEN 1990 AND 2000
    THEN 'Jeune recrue'
    WHEN YEAR(BirthDate) BETWEEN 1980 AND 1989
    THEN 'Dans la fleur de l'âge'
    WHEN YEAR(BirthDate) BETWEEN 1970 AND 1979
    THEN 'Roule sa bosse'
    ELSE 'Valeur improbable'
END
FROM HumanResources.Employee
WHERE BusinessEntityID = 21
```

Messages  
Votre employé est "Dans la fleur de l'âge"

# Boucles et curseurs

Utilisation des boucles

Définition du curseur

Manipulation des curseurs

Exemple de boucle avec curseur

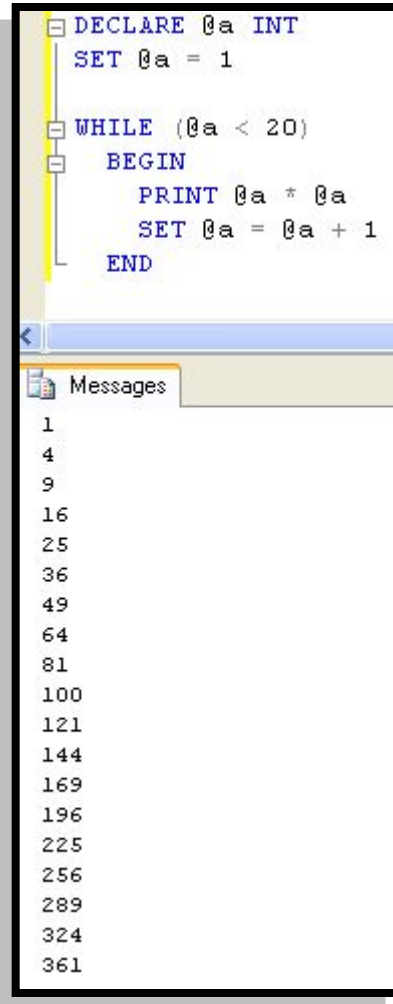
# Utilisation des boucles (1/2)

---

- Les boucles sont utiles lorsque vous désirez **répéter une instruction** un certain nombre de fois, **que vous connaissiez ce nombre ou pas**
- Les boucles peuvent être décomposées en **2 parties** distinctes :
  - >> **Les bornes de la boucle**  
Composées de **mots réservés** qui démarrent et clôturent la boucle ainsi que d'une condition de sortie
  - >> **Le corps de la boucle**  
Il s'agit de **la séquence d'ordres exécutables** compris dans les bornes de la boucle et exécutés à chaque itération
- En T-SQL, seule les boucles « **WHILE** » seront utilisées



# Utilisation des boucles (2/2)



```
DECLARE @a INT
SET @a = 1

WHILE (@a < 20)
BEGIN
    PRINT @a * @a
    SET @a = @a + 1
END
```

Messages

1  
4  
9  
16  
25  
36  
49  
64  
81  
100  
121  
144  
169  
196  
225  
256  
289  
324  
361

- Comme expliqué précédemment, les instructions que l'on désire répéter en boucle à l'intérieur du « **WHILE** » sont indiquées entre les bornes « **BEGIN ... END** »

- Une boucle while s'exécute **TANT QUE** la condition de départ est vraie. Si celle-ci n'est pas vraie au départ de la boucle, son contenu ne s'exécute même pas une fois  
Il est donc nécessaire de donner une valeur de départ à la condition de la boucle et s'assurer que la condition devienne fausse au bout d'un temps

- \* Les commandes « **BREAK** » et « **CONTINUE** » permettent de mettre fin à une boucle directement ou de continuer le traitement normalement. Souvent utilisé avec un IF, bien entendu

# Définition du CURSEUR

---

***Les curseurs** sont des mécanismes de mémoire tampons permettant d'accéder aux données renvoyées par une requête, elle-même pouvant être une jointure de deux ou plusieurs tables, présentant éventuellement plusieurs colonnes*  
*Le gros avantage des curseurs est de permettre de parcourir l'ensemble des lignes qu'il contient, une part une*

- Un curseur doit être déclaré au même titre qu'une variable. Lorsqu'il est déclaré, il ne constitue qu'une référence qui aidera à le générer lorsqu'il sera « **ouvert** » pour la première fois, moment auquel le système réserve alors de l'espace mémoire pour le curseur
- Une fois l'utilisation du curseur terminée, il sera nécessaire de le « **fermer** » ET de « **libérer** » l'espace mémoire qu'il occupait

# Manipulation des curseurs (1/2)

---

- Pour utiliser un curseur préalablement déclarer, il faudra utiliser la commande « **OPEN** *nom\_curseur* »

- Pour extraire une ligne du curseur dans les variables appropriées, nous utiliserons la commande

« **FETCH** *nom\_curseur* **INTO** @varCol1, @varCol2, @varCol3, ... »

A chaque fois que cette commande est utilisée, elle renvoi les données de la ligne suivante du curseur

- Pour mettre fin à l'utilisation du curseur, il faudra le fermer en utilisant la commande « **CLOSE** *nom\_curseur* »
- Fermer un curseur ne libère pas l'espace mémoire qu'il utilisait, au cas où il serait ouvert à nouveau et donc, recréé.  
La commande « **DEALLOCATE** *nom\_curseur* » permettra de supprimer l'espace mémoire que le curseur utilisait

# Manipulation des curseurs (2/2)

---

- Par défaut, le curseur prendra toujours la ligne **SUIVANTE** qu'il contient. Il est possible mais non recommander de remonter d'une ligne dans le curseur, pour des raisons de performance
- Lorsqu'un « **FETCH** » réussit à récupérer une ligne (*c'est-à-dire qu'il n'est pas arrivé à la fin du curseur*) alors la variable globale « **@@FETCH\_STATUS** » vaut « **VRAI** »
- La variable « **@@FETCH\_STATUS** » est bien utile afin de sortir automatiquement d'une boucle « **WHILE** ». Cependant, il faudra penser à faire un « **FETCH** » avant l'ouverture de la boucle, pour que la variable « **@@FETCH\_STATUS** » soit « **VRAI** » au départ.

# Exemple de boucle avec curseur

```
CREATE TABLE #empTemp (id INT, prenom VARCHAR(50), nom VARCHAR(50))

DECLARE @id_emp INT, @prenom_emp VARCHAR(50), @nom_emp VARCHAR(50)

DECLARE CR_employes CURSOR
FOR SELECT TOP 20 [BusinessEntityID], [FirstName], [LastName]
FROM [AdventureWorks2008R2].[Person].[Person]

OPEN CR_employes

FETCH CR_employes INTO @id_emp, @prenom_emp, @nom_emp

WHILE ( @@FETCH_STATUS = 0 )
BEGIN
    INSERT INTO #empTemp VALUES (@id_emp, @prenom_emp, @nom_emp)
    FETCH CR_employes INTO @id_emp, @prenom_emp, @nom_emp
END

CLOSE CR_employes
DEALLOCATE CR_employes

SELECT * FROM #empTemp
```

	id	prenom	nom
1	1	Ken	Sánchez
2	2	Terri	Duffy
3	3	Roberto	Tamburello
4	4	Rob	Walters
5	5	Gail	Erickson
6	6	Jossef	Goldberg
7	7	Dylan	Miller
8	8	Diane	Margheim
9	9	Gigi	Matthew
10	10	Michael	Raheem
11	11	Ovidiu	Cracium
12	12	Th...	...

# Fonctions et procédures

Les UDF

Syntaxe

Exemples

Les procédures

Syntaxe et exemple de procédure

Paramètres

Le paramètre OUT(PUT)

Paramètres de type TABLE

# Les UDF

---

*Une UDF, autrement dit « **User Define Function** » ou Fonction Définie par l'Utilisateur est un ensemble de commandes regroupées sous un même nom d'objet, que l'on définit pour des besoins de traitements récurrents au sein des requêtes et du code des procédures stockées ou des triggers. Elle fait donc partie intégrante de la base de données, où elle est **considérée comme un objet** à part entière, au même titre qu'une table, une vue, un utilisateur ou une procédure stockée.*

- Une fonction est un ensemble d'instructions qui remplissent une tâche particulière et surtout, **renvoient une valeur** !
- La valeur renvoyée par la fonction implique que **cette fonction ne peut pas être utilisée seule** ou exécutée directement, elle doit faire partie d'une autre instruction T-SQL
- Il existe **deux grands types de fonctions** : celles renvoyant une valeur et celles renvoyant un ensemble de données (une table donc)

# Syntaxe

---

- La syntaxe d'une fonction est la suivante. Il est à noter le type d'élément renvoyé et le mot-clé « **RETURN** », obligatoires, dans le code de la fonction

```
CREATE FUNCTION [ utilisateur. ] nom_fonction
    ( [ { @parametre1[AS] type [ = valeur_défaut ] } [ , @parametre2 ... ] ] )
    RETURNS type_résultant
    [ AS ]
    BEGIN
        code

        RETURN valeur_résultante
    END
```

- Le « *type\_résultant* » est donc un type classique, le mot-clé TABLE ou encore, une variable de type TABLE que l'on définit à cet endroit
- Il n'est pas possible, au sein du code d'une fonction, d'exécuter un ordre DML de manipulation des tables. Cela devra se simuler via une procédure qui jouera sur des paramètres passés en mode OUTPUT



# Exemples

```
CREATE FUNCTION DateDuJour ()
RETURNS DATETIME
AS
BEGIN
    DECLARE @date DATETIME
    SET @date = CURRENT_TIMESTAMP
    RETURN @date
END
```

```
SELECT dbo.DateDuJour ()
```

```
CREATE FUNCTION FN_JOUR_SEMAINE ()
RETURNS TABLE
AS
RETURN (SELECT 1 AS N, 'Lundi' AS JOUR
        UNION
        SELECT 2 AS N, 'Mardi' AS JOUR
        UNION
        SELECT 3 AS N, 'Mercredi' AS JOUR
        UNION
        SELECT 4 AS N, 'Jeudi' AS JOUR
        UNION
        SELECT 5 AS N, 'Vendredi' AS JOUR
        UNION
        SELECT 2 AS N, 'Samedi' AS JOUR
        UNION
        SELECT 2 AS N, 'Dimanche' AS JOUR)

SELECT *
FROM dbo.FN_JOUR_SEMAINE ()
```

Results Messages

N	JOUR
1	Lundi
2	Dimanche
3	Mardi
4	Samedi
5	Mercredi
6	Jeudi
7	Vendredi

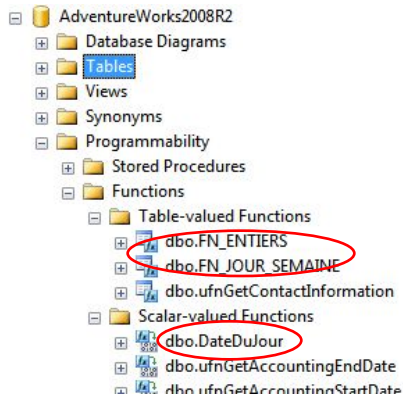
```
CREATE FUNCTION FN_ENTIERS ()
RETURNS @integers TABLE
    (N int PRIMARY KEY NOT NULL)
AS
BEGIN
    DECLARE @N INT
    SET @N = 0

    WHILE @N < 10
    BEGIN
        INSERT INTO @integers VALUES (@N)
        SET @N = @N + 1
    END
    RETURN
END

SELECT * FROM dbo.FN_ENTIERS ()
```

Results Messages

N
0
1
2
3
4
5
6
7
8
9



# Les Procédures

---

*Une procédure est également un objet de la base de données, regroupant un ensemble de commandes qui effectuent une tâche particulière le plus souvent, récurrente.*

*La différence fondamentale entre les procédures et les fonctions est qu'une procédure ne renvoi aucune valeur (du moins pas classiquement...)*

- Une procédure ne renvoyant aucune valeur peut donc être appelée comme une commande dans le code. Elle peut être exécutée seule, sans interagir avec d'autres éléments du code
- En créant des paramètre dits « de sortie » il est possible de simuler en quelques sortes un retour de valeur pour une procédure également

# Syntaxe et exemple de procédure

```
CREATE PROCEDURE [ utilisateur. ] nom_procedure
( [ { @parametre1[AS] type [ = valeur_défaut ] } [ , @parametre2 ... ] ] )

[ AS ]

BEGIN

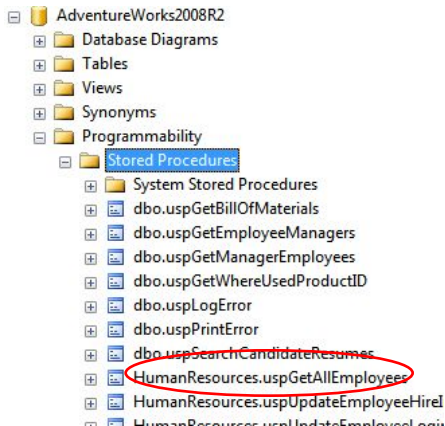
    code

END
```

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ( 'HumanResources.uspGetAllEmployees', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.uspGetAllEmployees;
GO
CREATE PROCEDURE HumanResources.uspGetAllEmployees
AS
    SET NOCOUNT ON;
    SELECT LastName, FirstName, Department
    FROM HumanResources.vEmployeeDepartmentHistory;
GO

HumanResources.uspGetAllEmployees;
-- EXEC (UTE) HumanResources.uspGetAllEmployees;
```

Results		
LastName	FirstName	Department
1 Mikovsky	Jan	Production
2 McAskil-White	Katie	Production
3 Hines	Michael	Production
4 Mirchandani	Nitin	Production
5 Decker	Barbara	Production
6 Chen	John	Production
7 Hesse	Stefen	Production
8 Kim	Shane	Production
9 McK...	Y...	Production



# Paramètres (1/2)

---

***Un paramètre** est une valeur éventuellement échangée entre une fonction ou une procédure et le programme appelant*

- On distingue deux types de paramètres

>> **Les paramètres FORMELS**, qui désignent le nom générique par lequel les paramètres sont référencés dans la déclaration de la fonction ou procédure

>> **Les paramètres RÉELS**, qui représentent la valeur qui est réellement échangée entre le programme appelant et la fonction ou la procédure.

# Paramètres (2/2)

- Les ***paramètres formels*** sont **définis dans le code de création** de la fonction ou de la procédure et n'ont de signification que dans ce contexte précis
- Les ***paramètres formels*** ont un **type** qui **ne peut avoir de contrainte** comme la taille de la chaîne de caractère (VARCHAR2) ou du nombre réel (NUMBER), par exemple
- Un ***paramètre formel*** et son ***paramètre réel*** correspondant doivent être **de même type**

```
CREATE PROCEDURE PrintBis @param_formel VARCHAR(100)
AS
BEGIN
    PRINT @param_formel
END
```

```
EXECUTE PrintBis 'Ceci est un paramètre RÉEL !'
```

Messages  
Ceci est un paramètre RÉEL !

# Les paramètres OUT(PUT)

- Les **paramètres OUTPUT** sont des paramètres d'une fonction ou procédure qui permettent de renvoyer une valeur au programme appelant, même s'il s'agit d'une procédure
- Pour récupérer la valeur renvoyée, il nous faudra une variable du même type, passée en paramètre de la procédure ou fonction
- Dans le cas d'une procédure, **vous** DEVREZ utiliser la commande EXECUTE pour utiliser une procédure qui demande un paramètre en mode OUTPUT

```
CREATE PROCEDURE Production.uspGetList @Product varchar(40)
, @MaxPrice money
, @ComparePrice money OUTPUT
, @ListPrice money OUT
AS
SET NOCOUNT ON;
SELECT p.[Name] AS Product, p.ListPrice AS 'List Price'
FROM Production.Product AS p
JOIN Production.ProductSubcategory AS s
ON p.ProductSubcategoryID = s.ProductSubcategoryID
WHERE s.[Name] LIKE @Product AND p.ListPrice < @MaxPrice;

SET @ListPrice =
(SELECT MAX(p.ListPrice)
FROM Production.Product AS p
JOIN Production.ProductSubcategory AS s
ON p.ProductSubcategoryID = s.ProductSubcategoryID
WHERE s.[Name] LIKE @Product AND p.ListPrice < @MaxPrice);

SET @ComparePrice = @MaxPrice;
```

```
DECLARE @ComparePrice money, @Cost money
EXECUTE Production.uspGetList '%Bikes%', 700,
@ComparePrice OUT,
@Cost OUTPUT
IF @Cost <= @ComparePrice
BEGIN
PRINT 'These products can be purchased for less than
$'+RTRIM(CAST(@ComparePrice AS varchar(20)))+'. '
END
ELSE
PRINT 'The prices for all products in this category exceed
$'+ RTRIM(CAST(@ComparePrice AS varchar(20)))+'. '
```

The screenshot shows the SQL Server Enterprise Manager interface. The 'Results' tab is active, displaying a table with two columns: 'Product' and 'List Price'. The table contains 11 rows of data. The 'Messages' tab is also visible, showing a message that says: 'These products can be purchased for less than \$700.00.'

Product	List Price
1 Road-750 Black, 58	539.99
2 Mountain-500 Silver, 40	564.99
3 Mountain-500 Silver, 42	564.99
4 Mountain-500 Silver, 44	564.99
5 Mountain-500 Silver, 48	564.99
6 Mountain-500 Silver, 52	564.99
7 Mountain-500 Black, 40	539.99
8 Mountain-500 Black, 42	539.99
9 Mountain-500 Black, 44	539.99
10 Mountain-500 Black, 48	539.99
11 Mountain-500 Black, 52	539.99



# Paramètres de type TABLE

---

- Il est bien entendu possible de passer en paramètre une variable de type table contenant des données, pour, par exemple, faire une insertion multi-lignes

```
CREATE TYPE LocationTableType AS TABLE
( LocationName VARCHAR(50)
, CostRate INT );
GO
```

```
CREATE PROCEDURE usp_InsertProductionLocation
    @TVP LocationTableType READONLY
AS
SET NOCOUNT ON
INSERT INTO [AdventureWorks2012].[Production].[Location]
    ([Name]
    , [CostRate]
    , [Availability]
    , [ModifiedDate])
SELECT *, 0, GETDATE()
FROM @TVP;
GO
```

# Transactions

Loi ACID

Caractéristiques

Concurrence d'accès aux données

SET TRANSACTION

Les verrous



# Loi ACID

---

*Toute transaction répond obligatoirement à la loi ACID, ce qui signifie qu'elle est :*

- **ATOMIQUE** :: *Chacun des changements d'état de la transaction arrive ou aucun n'arrive*
- **COHÉRENTE** :: *Une transaction est un changement d'état réussi ou annulé ; l'ensemble des actions qu'elle comprend respectent les contraintes d'intégrité de cet état*
- **ISOLÉE** :: *Chaque transaction s'exécute une par une*
- **DURABLE** :: *Si une transaction est validée, elle l'est de manière permanente et elle survit à tout incident ultérieur*

# Caractéristiques (1/2)

---

- En T-SQL, la valeur « **AUTOCOMMIT** » (validation automatique) est obligatoirement à ON en permanence. Cela signifie que si on ne l'énonce pas explicitement, tout ordre de modification de la base de données (DDL ou DML) sera une transaction à part entière
- Cliquer sur le bouton « **Execute** » de la console implique autant de « **COMMIT** » (validation) qu'il y a d'ordre dans le script. Un ordre qui échoue met fin à l'exécution des ordres qui suivent et fait un « **ROLLBACK** » (annulation) du dernier ordre
- Afin d'éviter un « **COMMIT** » automatique de chacun des ordres d'un script séparément, il faudra commencer l'ensemble de ces ordres par la commande « **BEGIN TRANSACTION** » et les terminer par un « **COMMIT** » ou un « **ROLLBACK** »

# Caractéristiques (2/2)

---

- Il est possible de créer des points de sauvegarde afin de ne pas devoir revenir au début de la transaction et de faire plutôt un « **ROLLBACK TRANSACTION nom\_point\_sauvegarde** »

```
☐ BEGIN TRANSACTION  
  
    DROP TABLE [Order Details]  
  
    SAVE TRANSACTION point_sauvegarde1  
  
    DROP TABLE [CustomerCustomerDemo]  
  
    ROLLBACK TRANSACTION point_sauvegarde1
```

# Concurrence d'accès aux données

---

- **Plusieurs sessions utilisateurs** peuvent être démarrées en parallèle. Cela implique que plusieurs transactions peuvent **entrer en concurrence** pour accéder à la même information d'une base de données. Il est par conséquent primordiale que SQL Server puisse **gérer l'accès aux données** afin de préserver leur intégrité
- Deux type de manipulations sont disponibles afin de gérer la concurrence d'accès aux données :
  - >> **SET TRANSACTION** : défini la transaction dans un mode de lecture spécifique, qui gèrera lui-même les verrous et la visibilité des données modifiées par la transaction
  - >> Les « **TableLocks** » : ce sont des verrous que l'on peut soi-même imposer sur un ensemble de données afin qu'il ne soit pas accédé avant la fin de la transaction en cours

# SET TRANSACTION (1/2)

---

## SET TRANSACTION ISOLATION LEVEL

```
{ READ UNCOMMITTED  
| READ COMMITTED  
| REPEATABLE READ  
| SNAPSHOT  
| SERIALIZABLE  
}
```

- L'ordre « **SET TRANSACTION** » permet de modifier la visibilité et l'accès aux données des tables de la base de données durant une transaction donnée

- **Read UNCOMMITTED** – *Niveau le plus bas (0)*  
Permet la lecture des objets d'une BD, que ceux-ci soient en cours de modification ou non, ou même s'ils changent d'état lors de l'analyse de la BD. La transaction ignore donc les verrous posés par d'autres transactions, bien qu'elle ne puisse les outrepasser. On parlera de données fantômes dans ce cas

- **Read COMMITED** – *Mode par défaut (Niveau 1)*

Spécifie que les verrous partagés sont maintenus durant la lecture des données pour éviter des lectures incorrectes. Les données peuvent néanmoins être modifiées avant la fin de la transaction, ce qui donne des lectures non renouvelées ou des données fantômes.

# SET TRANSACTION (2/2)

---

- **REPEATABLE Read** – *Niveau 2*

Des verrous sont placés dans toutes les données utilisées dans une requête, afin d'empêcher les autres utilisateurs de les mettre à jour. Toutefois, un autre utilisateur peut ajouter de nouvelles lignes fantômes dans un jeu de données par un utilisateur ; celles-ci seront incluses dans des lectures ultérieures dans la transaction courante.

- **SERIALIZABLE** – *Niveau 3*

Place un verrou sur une plage de données, empêchant les autres utilisateurs de les mettre à jour ou d'insérer des lignes dans le jeu de données, jusqu'à la fin de la transaction. Il s'agit du niveau d'isolation le plus restrictif parmi les quatre niveaux disponibles. **Utilisez cette option uniquement lorsque cela s'avère nécessaire**, car la concurrence d'accès est moindre. Cette option a le même effet que l'utilisation de l'option HOLDLOCK dans toutes les tables de toutes les instructions SELECT d'une transaction.

# Les verrous

---

```
INSERT INTO nom_table WITH (MODE_VERROUILLAGE) (...) VALUES ...  
UPDATE nom_table WITH (MODE_VERROUILLAGE) SET ...  
DELETE FROM nom_table WITH (MODE_VERROUILLAGE) WHERE ...  
SELECT ... FROM nom_table WITH (MODE_VERROUILLAGE) WHERE ...
```

- **NOLOCK**

Dans une commande SELECT uniquement, permet de faire abstraction des verrous posés. Cela peut donner des informations fantômes

- **TABLELOCK**

Créer un verrou sur la table afin d'y empêcher toute modification tant que l'ordre n'est pas terminé

- **HOLDLOCK**

Peut être rajouté aux verrous existants afin de demander à ce que le verrou soit maintenu jusqu'à la fin de la transaction. Si tous les objets se retrouvent en HOLDLOCK dans une transaction, cela correspond au niveau d'isolation SERIALIZABLE

## Liste de l'ensemble des verrous sur le site MSDN

[http://msdn.microsoft.com/fr-fr/library/aa213026\(v=sql.80\).aspx](http://msdn.microsoft.com/fr-fr/library/aa213026(v=sql.80).aspx)

# Triggers

Définition

Syntaxes

INSERTED et DELETED

Fonction UPDATE

ROLLBACK et COMMIT

RAISERROR et @@ERROR

Exemples de triggers DML

Exemples de triggers DDL et LOGON



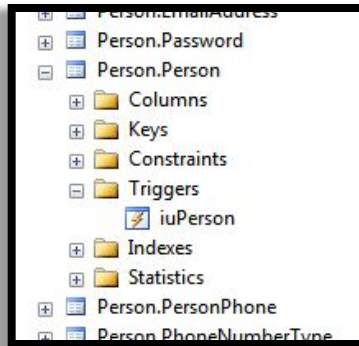
# Définition

---

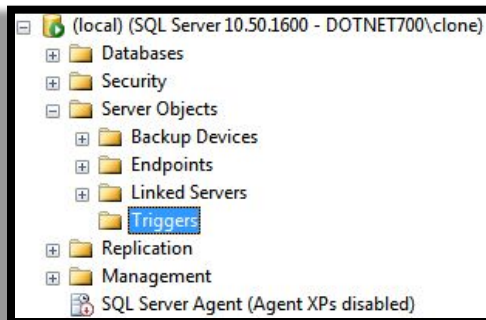
Un **trigger** est un programme stocké contenant une séquence d'instructions qui se déclenchent **automatiquement** lorsqu'un ordre particulier (DML, par exemple) sur une table de la base de données

- Le T-SQL gère des « **Triggers DML** », c'est-à-dire les triggers liés aux ordres INSERT, UPDATE, DELETE, liés à une table ou une vue, les « **Triggers DDL** » appliqués aux ordres CREATE, ALTER, DROP, GRANT, DENY, REVOKE ou UPDATE STATISTICS, et les « **Triggers Logon** », déclenchés lors de l'authentification au serveur, avant l'établissement des sessions utilisateurs
- Les Triggers T-SQL ne sont prévus que pour se déclencher après un ordre SQL (**FOR/AFTER**) ou à la place de cet ordre (**INSTEAD OF**). Les Triggers BEFORE peuvent être simulés, mais n'existent pas tels quels

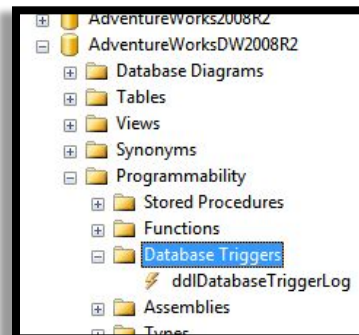
# Syntaxes



```
CREATE TRIGGER nomTrigger
ON nomTable
AFTER {INSERT|DELETE|UPDATE}
AS
BEGIN
    SET NOCOUNT ON;
    -- CODE
END
```



```
CREATE TRIGGER nomTrigger
ON ALL SERVER
FOR|AFTER LOGON
AS
BEGIN
    -- CODE
END
```



```
CREATE TRIGGER nomTrigger
ON {ALL SERVER|DATABASE}
FOR|AFTER {type_evenement|groupe_evenements}
AS
BEGIN
    -- CODE
END
```

# INSERTED et DELETED (1/2)

---

- SQL Server prévoit de récupérer l'information manipulée par l'utilisateur lors de l'action qui a déclenché le trigger. L'information est récupérée comme suit :

>> **Lors d'un INSERT**, il n'existe que la table **INSERTED** qui contient les nouvelles données insérées

>> **Lors d'un DELETE**, il n'existe que la table **DELETED** qui contient les éléments supprimés

>> **Lors d'un UPDATE**, il existe simultanément les tables **INSERTED et DELETED**. La table **INSERTED** contient les nouvelles données mises à jour et la table **DELETED** contient les anciennes données qui disparaîtront

# INSERTED et DELETED (2/2)

---

- Ces tables contiennent automatiquement **la même structure** que les tables auxquelles elles sont liées : si une mise à jour a lieu sur la table CLIENT, la table INSERTED temporaire aura autant de colonne que la table CLIENT contiendra les lignes mises à jour

```
CREATE TRIGGER Purchasing.LowCredit
ON Purchasing.PurchaseOrderHeader
AFTER INSERT
AS
IF EXISTS (SELECT *
           FROM Purchasing.PurchaseOrderHeader p
           JOIN inserted AS i
           ON p.PurchaseOrderID = i.PurchaseOrderID
           JOIN Purchasing.Vendor AS v
           ON v.BusinessEntityID = p.VendorID
           WHERE v.CreditRating = 5
          )
BEGIN
...
```

# Fonction UPDATE

---

- La fonction « **UPDATE** » est une fonction que l'on peut utiliser **dans les triggers INSERT et UPDATE** et qui permet de vérifier si telle ou telle colonne a subi une mise à jour lors de la requête qui a déclenché le trigger

```
IF [NOT] UPDATE (<colonne>)  
BEGIN  
    <traitement>  
END
```

# ROLLBACK et COMMIT

---

- Le but d'un trigger sera bien souvent de vérifier qu'un ordre est valide et de l'accepter dans la base de données ou le refuser, c'est-à-dire le « **COMMIT** » ou plutôt le « **ROLLBACK** »
- En ce qui concerne le « **COMMIT** », cela se fera automatiquement, il n'est donc pas nécessaire (et même recommander de ne pas le faire) d'explicitement un « **COMMIT** » dans la transaction, cela lèvera une erreur qui annoncera que la transaction s'est terminée dans le trigger alors qu'elle sera validée
- Le « **ROLLBACK** » quant à lui sera toujours nécessaire. Par défaut, le « **ROLLBACK** » annulera le trigger et l'instruction qui l'ont démarré. Pour n'annuler que les ordres du triggers, il faudra créer une transaction explicitement dans le trigger
- Les ordres de validation du trigger peuvent bien entendu être contenu dans une conditionnelle telle que le IF

# RAISERROR et @@ERROR

---

- Lorsque l'on désire afficher un message d'erreur ou réagir par rapport à une erreur levée par le système, il faudra utiliser la commande « **RAISERROR** » ou encore la variable globale « **@@ERROR** »
- Le RAISERROR permet juste l'affichage d'un message d'erreur à l'utilisateur, en plus du message classique fourni par le système et d'un numéro d'erreur. Dans sa forme la plus simple, il s'écrit :  
**RAISERROR** XXXX 'message d'erreur'
- La variable globale « **@@ERROR** » renvoie 0 si aucune erreur n'a été levée par le système. Il est possible de réagir en fonction de la variable et de par exemple, faire un ROLLBACK si l'erreur est survenue

```
IF @@Error <> 0
BEGIN
    RAISERROR 14000 'OPERATION IMPOSSIBLE'
    ROLLBACK TRANSACTION
END
```

# Exemples de triggers DML

---

```
USE AdventureWorks2008R2;
GO
IF OBJECT_ID ('Sales.reminder2','TR') IS NOT NULL
    DROP TRIGGER Sales.reminder2;
GO
CREATE TRIGGER reminder2
ON Sales.Customer
AFTER INSERT, UPDATE, DELETE
AS
    EXEC msdb.dbo.sp_send_dbmail
        @profile_name = 'AdventureWorks2012 Administrator',
        @recipients = 'danw@Adventure-Works.com',
        @body = 'Don''t forget to print a report for the sales force.',
        @subject = 'Reminder';
GO
```

```
CREATE TRIGGER E_CLI_INS
ON PERSON.PersonPhone
FOR INSERT, UPDATE
AS
SELECT CAST(REPLACE(PhoneNumber, '.', '') as DECIMAL(20))
FROM INSERTED
IF @@Error <> 0
ROLLBACK TRANSACTION
```



# Exemples de triggers DDL et LOGON

```
USE AdventureWorks2008R2;
GO
IF EXISTS (SELECT * FROM sys.triggers
           WHERE parent_class = 0 AND name = 'safety')
DROP TRIGGER safety
ON DATABASE;
GO

CREATE TRIGGER safety
ON DATABASE
FOR DROP_SYNONYM
AS
    RAISERROR 15000 'You must disable Trigger "safety" to drop synonyms!'
    ROLLBACK
GO

DROP TRIGGER safety
ON DATABASE;
GO
```

```
USE master;
GO

CREATE LOGIN login_test WITH PASSWORD = '3KHJ6dhx{0xVYsdf' MUST_CHANGE,
    CHECK_EXPIRATION = ON;
GO

GRANT VIEW SERVER STATE TO login_test;
GO

CREATE TRIGGER connection_limit_trigger
ON ALL SERVER WITH EXECUTE AS 'login_test'
FOR LOGON
AS
BEGIN
    IF ORIGINAL_LOGIN() = 'login_test' AND
        (SELECT COUNT(*) FROM sys.dm_exec_sessions
         WHERE is_user_process = 1 AND
              original_login_name = 'login_test') > 3
        ROLLBACK;
END;
```

- Liste des événements DDL auxquels il est possible de lier les triggers sur la base de données :  
<http://msdn.microsoft.com/en-us/library/bb522542.aspx>