



Concepteur Développeur en Informatique

Assurer la persistance des données

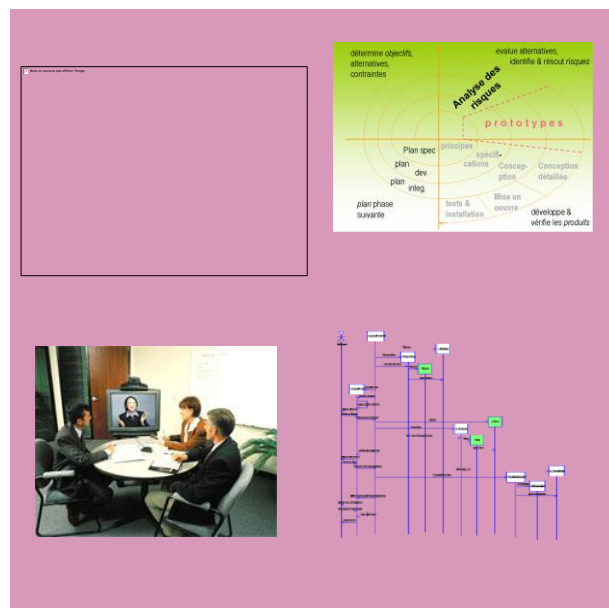
SQL – Programmer le SGBDR

Accueil

Apprentissage

PAE

Evaluation



Module 3 – Développer la persistance

Sommaire

1	Introduction	3
2	Eléments de programmation	3
2.1	Les commentaires	3
2.2	Règles de nommage des objets	4
2.3	Les variables.....	4
2.4	Déclaration et affectation d'une variable.....	5
3	Les structures conditionnelles et itératives	6
3.1	Délimitation des blocs d'instructions.....	6
3.2	Expression de blocs conditionnels	6
3.3	Expression de blocs itératifs conditionnels	9
4	Rappel sur les fonctions intégrées	10
4.1	Fonctions de conversion	10
4.2	Fonctions de traitement de chaînes	11
4.3	Fonctions de manipulation de dates	11
5	Gestion des exceptions	12
5.1	Bloc Try Catch	12
5.2	Mise en œuvre de la gestion des exceptions.....	14
5.3	Récupérer les informations sur les erreurs	15
5.4	Définir des conditions propres d'exception.....	16
6	Programmer des procédures stockées.....	19
6.1	Les différents types de procédures stockées.....	20
6.2	Codification des procédures stockées.....	20
6.3	Définition des paramètres	20
6.4	Exemples de procédures stockées.....	21
6.5	Appel des procédures stockées	23
6.6	Passage des paramètres par position	24
6.7	Passage des paramètres par référence nommée	26
7	Programmation des fonctions	27
7.1	Syntaxe de la création d'une fonction	27
7.2	Fonction Scalaire.....	28
7.3	Fonction table.....	29
8	Programmer la récursivité	30

1 Introduction

Ce support a pour but de vous initier à la programmation des serveurs de bases de données relationnelles. Le système cible retenu est SQL Server et le langage de programmation Transact SQL, langage que vous avez déjà eu l'occasion de mettre en œuvre lors des étapes précédentes consacrées à la manipulation des données à l'aide de requêtes SQL.

Dans un premier temps, nous allons observer les principes de la programmation procédurale sous SQL Server et nous initier à la création de scripts avec le langage procédural T-SQL.

Nous aborderons par la suite la programmation des procédures stockées et des fonctions.

La programmation des transactions au sein de programmes compilés et stockés sur le serveur SGBDR apporte de la rigueur à vos développements et facilite la maintenance à venir de vos applicatifs.

Les programmes qui manipulent les données du SGBD ne doivent pas faire appel directement à des transactions mais utiliser des procédures auxquelles seront passés des arguments en paramètre.

Il faudra donc à tout prix éviter de coder « en dur » au sein des programmes utilisateur les instructions relatives à la manipulation des données.

Cette contrainte illustre l'approche de la programmation par couche qui permet de structurer son code en fonction de la nature des opérations (Affichage, Accès données, Maintenance Données, ...)

Le recours à la programmation de ces transactions dans des procédures stockées permet :

- D'assurer une répartition correcte des traitements côté client et serveur
- De centraliser le code relatif à la manipulation des données et donc de rendre plus aisée la maintenance
- De faciliter la réutilisation de code

Vous devrez aussi vous approprier des notions fondamentales relatives au fonctionnement d'un système transactionnel.

2 Éléments de programmation

2.1 Les commentaires

Il est possible d'insérer dans son code des commentaires :

Pour commenter une ligne, il suffira de faire précéder le commentaire de deux tirets –
Pour commenter un bloc, de le délimiter par /* et */

```
/* Conversion d'une valeur
en décimal. Il est possible d'utiliser aussi CAST */

SELECT CONVERT(Decimal(10,3),
sum(({UnitPrice * 1- Discount) * Quantity}) AS "TOTAL CA Net"
FROM [Order Details] -- conversion en décimal d'une valeur
```

Figure 1 : Commentaires

2.2 Règles de nommage des objets

Tous les noms d'objets (table, colonne, variable, etc.) doivent respecter les règles suivantes :

- Le nombre de caractères maximum est de 128
- les caractères qui composent le nom seront de préférence non accentués.
- Il sera aussi préférable d'éviter les caractères spéciaux qui peuvent par ailleurs avoir une utilisation réservée par le langage.
- Les noms doivent commencer par une lettre et ne doivent pas comporter d'espace. Si le nom ne respecte pas ces 2 dernières règles, vous devrez alors le délimiter par des crochets [].

Il n'est pas nécessaire de respecter la casse des caractères mais par convention seuls les mots clefs du langage seront en majuscules.

2.3 Les variables

Les variables sont typées. Leurs principaux types sont :

2.3.1 Types numériques:

INT entier (et ses dérivés SMALLINT, TINYINT, BIGINT)

DECIMAL(11,2) montant à 11 chiffres (décimaux) dont 2 après la virgule.

Il convient de définir la précision (nombre maximal de chiffres) et l'échelle (nombre de chiffres à droite de la virgule). La taille occupée en mémoire dépendra de l'échelle.

REAL réel flottant codé sur 4 octets et FLOAT (double précision) sur 8 octets.

2.3.2 Types caractères :

CHAR(50) : chaîne de caractères de longueur fixe quelle que soit la valeur

VARCHAR(25) : chaîne de caractères de longueur variable ne pouvant contenir plus de 25 caractères.

NCHAR et NVARCHAR sont de même nature que les précédents mais la définition du caractère est en Unicode et stockée sur 2 octets.

Ce type est particulièrement intéressant si vous devez stocker des valeurs de données exprimées en plusieurs langues, comme c'est souvent le cas dans les applications multilingues.

La longueur d'une chaîne, exprimée selon ces types, ne peut excéder 8000 caractères. Pour des valeurs excédant cette taille, il vous faut utiliser les types prévus pour stocker des BLOBs (Binary Large Object). Ces types sont définis comme :

TEXT ou NTEXT : Permet de stocker une chaîne au format ASCII ou Unicode.

IMAGE : Permet de stocker un flux d'octets.

Cette approche permet de stocker des valeurs qui excèdent 2 milliards d'octets. En fait, la valeur réelle stockée au niveau de la table est un pointeur de référence de l'objet stocké sur disque.

2.3.3 Types divers

MONEY : décimal avec symbole monétaire. Il est préférable de ne pas avoir recours à ce type et stocker la valeur de la devise dans une colonne spécifique.

DATETIME, SMALLDATETIME : date et heure. DATETIME s'emploie pour les données comprises entre le 1er janvier 1753 et le 31 décembre 9999 (chaque valeur est stockée dans 8 octets). L'autre type, SMALLDATETIME, s'applique aux dates comprises entre le 1er janvier 1900 et le 6 juin 2079 (chaque valeur est stockée dans 4 octets). Des fonctions spécifiques permettent de manipuler ces types.

TIME pour les valeurs de temps.

BIT : Booléen.

Programmation des SGBDR

TIMESTAMP : Ce type de données présente des nombres binaires automatiquement générés, et dont l'unicité est garantie dans une base de données. **TIMESTAMP** est généralement utilisé en tant que mécanisme d'affectation d'un numéro de version aux lignes des tables. La taille de stockage est de 8 octets. La valeur d'une colonne de ce type varie à chaque opération de modification de la ligne. Une seule colonne par table peut être définie de ce type. Nous verrons par la suite comment utiliser ce type de données.

XML : depuis SQL Server 2005, il est possible de stocker des fragments XML dans un type spécifique. Le type de données xml est un type de données intégré de SQL Server, quelque peu similaire aux autres types intégrés, tels que **INT** et **VARCHAR**. À l'image des autres types intégrés, vous pouvez utiliser le type de données XML comme type de colonne lorsque vous créez une table en tant que type de variable, de paramètre, de retour de fonction ou dans **CAST** et **CONVERT**.

Types dérivés

Il est possible de dériver des types SQL Server afin de créer ses propres types, désignés sous le vocable de Types de données utilisateur.

Cette approche permet de rendre plus explicite les définitions des données et de leur associer des règles qui devront être vérifiées systématiquement lors d'une demande de stockage d'une valeur dans la colonne.

Pour créer un type dérivé, il faut recourir à l'exécution de la procédure stockée **sp_addtype**.

```
-- Le code postal doit être constitué de 5 chiffres  
create rule [RCodepostal] as @CodePostal like '[0-9][0-9][0-9][0-9][0-9]'  
-- Le code postal est dérivé du type caractères  
EXEC sp_addtype N'CodePostal', N'char (5)', N'not null'  
-- La règle est associée au type CodePostal  
EXEC sp_bindrule N'[dbo].[RCodepostal]', N'[CodePostal]'
```

Figure 2 : Exemple de création d'un type de données utilisateur

2.4 Déclaration et affectation d'une variable

Une variable se déclare à l'aide du mot clé **DECLARE**.

Par convention, le nom de la variable locale à la procédure ou au script est préfixé par un **@**.

Il existe aussi des variables dites globales qui représente des valeurs du système. Elles sont alors préfixées de **@@**.

L'opérateur **SET** permet d'affecter une valeur à une variable.

On peut aussi affecter une valeur à une variable par le biais de l'exécution d'une requête de sélection et donc de l'ordre **SELECT**. Il convient de s'assurer que la requête ne renvoie alors qu'une seule ligne afin d'obtenir une valeur scalaire.

L'opérateur **PRINT** permet d'imprimer le contenu d'une variable. A utiliser lors des phases de tests...

L'exemple présenté page suivante montre les principes d'affectation selon les deux procédés et met en œuvre des opérations de conversion de type.

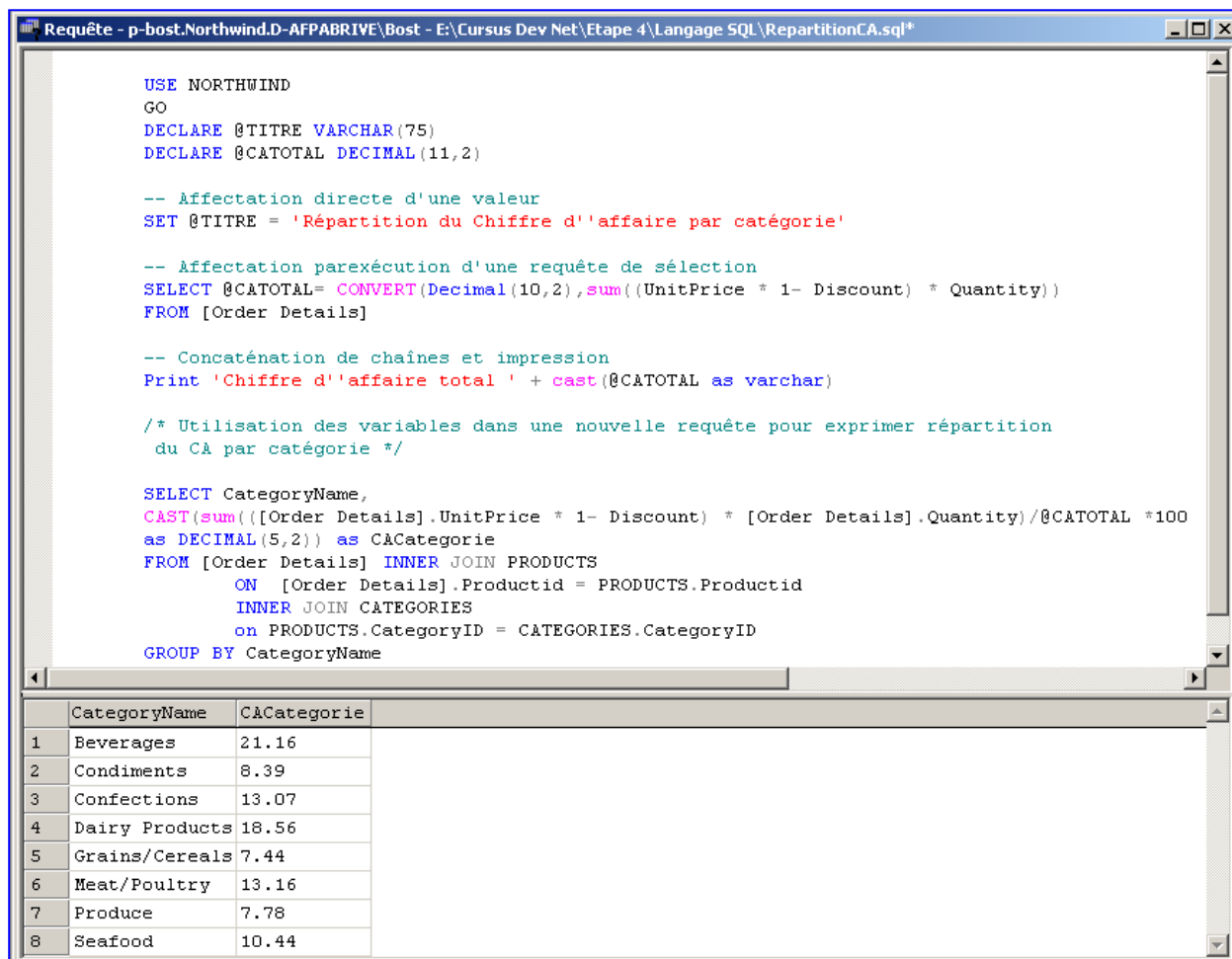


Figure 3 : Utilisation de variables

3 Les structures conditionnelles et itératives

Vous retrouvez dans le langage Transact SQL les structures les plus usuelles d'un langage de programmation qui vous permettront de programmer l'exécution conditionnelle d'instructions ou la répétition de l'exécution de blocs d'instructions.

3.1 Délimitation des blocs d'instructions

Un bloc d'instructions est délimité par les instructions BEGIN et END. Cette structure est utilisée par toutes les autres structures, conditionnelles, itératives, transactionnelles, ...

3.2 Expression de blocs conditionnels

```

IF expression conditionnelle
    Instruction ou Bloc d'instructions
ELSE      facultatif
    Instruction ou Bloc d'instructions
    
```

L'exemple ci-dessous met en œuvre des instructions conditionnées dans le cadre d'une procédure stockée dont le rôle est de supprimer un client référencé dans la base de données.

```
CREATE PROCEDURE PS_Client_Supprimer

/* Procédure de suppression d'un client dans la table Customers
Un client ne peut être supprimé que si aucune commande
ne lui a été associée.
Liste des paramètres :
En entrée :
Identifiant client
En sortie :
Valeur retour 0 pas de suppression, 1 client supprimé */

@CustomerID nchar(5)
AS

IF EXISTS(SELECT OrderID FROM Orders where CustomerId = @CustomerId)
BEGIN
    PRINT('Le client ' + @CUSTOMERID + ' ne peut être supprimé car il a des commandes.')
    RETURN 0
END
ELSE
BEGIN
    PRINT('Le client ' + @CUSTOMERID + ' a été supprimé.')
    DELETE Customers WHERE CustomerID = @CustomerID
    RETURN 1
END
```

Figure 4 : Bloc conditionnel IF

Il est possible d'avoir recours à la structure CASE qui permet l'évaluation successive de différentes conditions au sein d'un même groupe.

La fonction CASE peut être mise en œuvre de deux manières :

- La fonction CASE détermine le résultat en comparant une expression à un jeu d'expressions simples ;
- La fonction CASE détermine le résultat en évaluant un jeu d'expressions booléennes.

Les deux formes prennent en charge un argument ELSE facultatif.

Les deux exemples qui suivent vous permettent de mieux appréhender la syntaxe de ces expressions conditionnelles.

A noter : lorsque vous recourez à cette structure pour réaliser des mises à jour conditionnelles, il faut prendre garde au fait que la colonne devant être modifiée vaudra Null si vous ne lui affectez aucune valeur et ne conservera donc pas sa valeur initiale.

D'où la programmation de l'instruction sur ELSE qui sera exécutée dans le cas où aucune condition n'a été vérifiée dans l'expression des différents cas.

```
USE VOLAVION
update pilote
SET salaireBrut =
    CASE pil#
        WHEN 2 THEN 12500
        WHEN 3 THEN 25000
    ELSE
        SalaireBrut
    END
```

Figure 5 : Utilisation case forme simple


```
USE Volavion
UPDATE pilote
SET salaire =
    CASE
    WHEN salaire < 10000 THEN salaire * 1.10
    WHEN salaire between 10000 and 20000 THEN salaire * 1.08
    ELSE salaire * 1.05
    END
```

Figure 6 : Utilisation case forme complexe

3.3 Expression de blocs itératifs conditionnels

Il n'existe qu'une seule structure qui permet de réaliser des traitements itératifs, la structure WHILE.

WHILE expression de la condition
Instruction ou bloc d'instructions

L'exemple suivant illustre la mise en place d'une itération similaire à une boucle FOR.

Vous pouvez éventuellement conditionner l'arrêt ou la poursuite d'un traitement en fonction d'une condition incluse dans la boucle. Attention à ne pas abuser de cette structure qui peut révéler des faiblesses au niveau de la structure logique.

IF *Expression de la condition*
 BREAK
ELSE
 CONTINUE
END

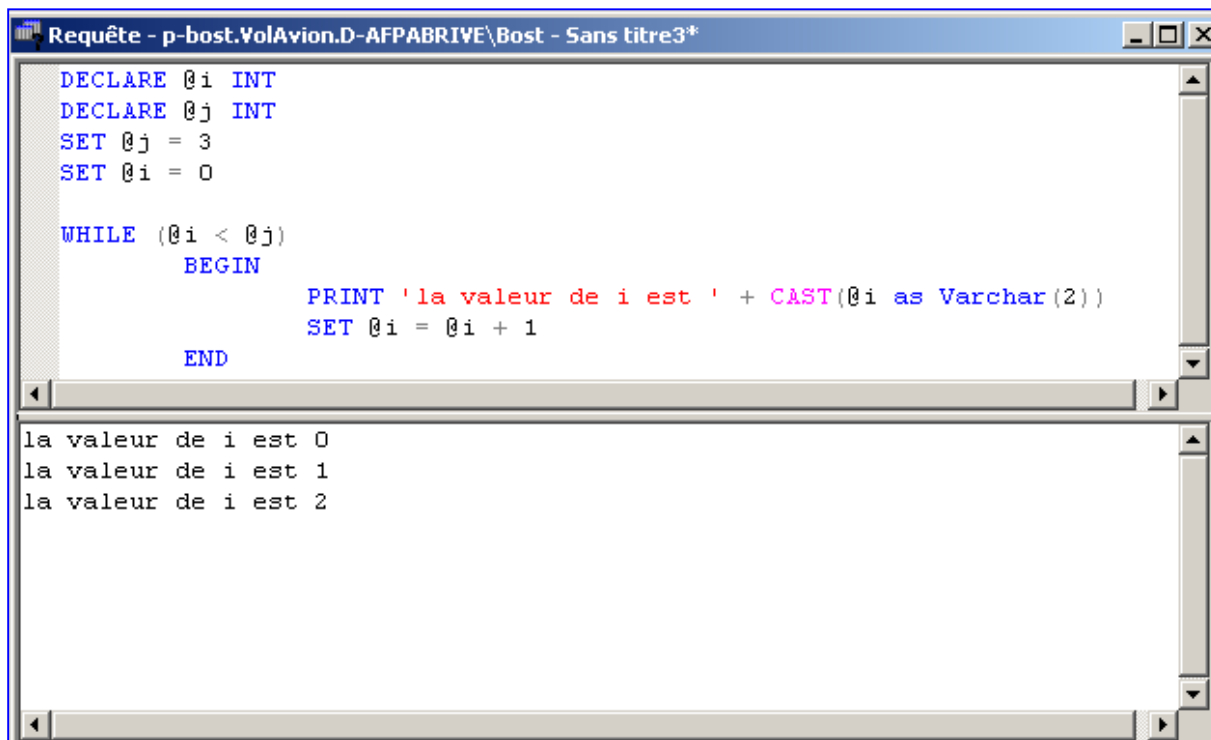


Figure 7 : Illustration d'un bloc itératif d'instructions

4 Rappel sur les fonctions intégrées

4.1 Fonctions de conversion

Certaines conversions ne peuvent être automatiquement réalisées par le système. Nous devons alors réaliser ces conversions de manière explicite au moyen des fonctions de conversion CAST et CONVERT.

Attention aux types d'origine et résultant de la conversion : toutes les combinaisons ne sont pas admises.

CONVERT permet de définir un style pour la donnée convertie alors que CAST ne le permet pas.

La fonction système GETDATE() renvoie la date du jour.

Si je souhaite convertir celle-ci dans un format américain, j'utilise la fonction CONVERT avec le style approprié.

Pour plus d'informations, voir l'aide de Transact SQL à l'index CONVERT

```
SELECT CONVERT(DATETIME,GETDATE(),102) AS "Date au format américain"
```

	Date au format américain
1	2004-11-28 14:57:29.530

Je souhaite que le CA net soit converti et présenté dans un decimal de 10 de long avec 3 chiffres derrière la virgule.

```

SELECT CONVERT(Decimal(10,3),sum((UnitPrice * 1- Discount) * Quantity)) AS "TOTAL
CA Net"
FROM [Order Details]
    
```

	TOTAL CÀ Net
1	1351747.630

4.2 Fonctions de traitement de chaînes

Quelques exemples dans ce tableau car elles sont nombreuses ...

Voir aide à l'index fonctions, chaîne

LEFT, RIGHT	Extraire des caractères à gauche ou à droite
UPPER, LOWER	Mettre en majuscules ou minuscules
LTRIM, RTRIM	Suppression des espaces à gauche ou à droite
SUBSTRING	Extraction d'une sous chaîne
REVERSE	Inversion d'une chaîne (miroir...)
LEN	Longueur d'une chaîne
ASCII	Valeur ascii d'un caractère
NCHAR	Renvoie le caractère Unicode fonction de la valeur donnée
REPLACE	Remplacement d'une occurrence de chaîne par une autre

Liste des noms des pilotes formatés.

Le premier caractère de gauche est mis en majuscules

Les autres caractères en minuscules

```
SELECT Upper(Substring(Nom,1,1)) + Substring(Nom,2,Len(Nom)-1)
FROM PILOTE
```

	{Aucun nom de colonne}
1	Serge
2	Jean
3	Roger
4	Robert
5	Michel
6	Bertrand
7	Hervé
8	Luc

Remplacement de l'occurrence Toulouse par Ville Rose dans l'attribut Ville de la Table Pilote.

```
SELECT REPLACE(VILLE,'Toulouse','Ville Rose')
FROM PILOTE
WHERE VILLE LIKE 'TOUL%'
```

	{Aucun nom de colonne}
1	Ville Rose

4.3 Fonctions de manipulation de dates

Fonctions intégrées permettant de manipuler des valeurs de type DATETIME.

DATEADD	Ajout d'un intervalle de temps à une date
DATEDIFF	Intervalle de temps entre deux dates
DATEPART	Extraction d'une partie de date

Programmation des SGBDR

DATENAME	Chaîne représentant une partie de date
DAY,MONTH,YEAR	Renvoie d'une partie de date
GETDATE,GETUTCDATE	Date du système

Quelques exemples :

Ajout de 3 jours à la date de naissance du Pilote

```
SELECT DATEADD(DAY,3,DateNaissance) as "Date + 3 jours", DateNaissance  
FROM Pilote
```

	Date + 3 jours	DateNaissance
1	1954-07-03 00:00:00.000	1954-06-30 00:00:00.000
2	1954-06-28 00:00:00.000	1954-06-25 00:00:00.000
3	1954-06-08 00:00:00.000	1954-06-05 00:00:00.000

Nombre de jours entre la date de naissance et la date du jour.

```
SELECT NOM,DATEDIFF(DAY,DateNaissance,GETDATE()) as "Nombre jours depuis  
Naissance"  
FROM Pilote
```

	NOM	Nombre jours depuis Naissance
1	Serge	18414
2	Jean	18419
3	Roger	18439

Extrait de portions de la date de naissance avec DatePart et DateName

```
SELECT DATEPART(MONTH,DateNaissance),  
DATENAME(MONTH,DateNaissance)  
FROM PILOTE
```

5 Gestion des exceptions

Comme tout langage de programmation vous disposerez de la possibilité d'intercepter les exceptions et de générer éventuellement des exceptions sur des conditions particulières survenant lors de l'exécution de votre programme.

Les erreurs du code Transact-SQL peuvent être traitées à l'aide d'une construction TRY...CATCH semblable aux fonctionnalités de gestion des exceptions des langages Microsoft Visual Basic et Microsoft Visual C#.

5.1 Bloc Try Catch

Une construction TRY...CATCH comprend deux parties : un bloc TRY et un bloc CATCH. Lorsque le système détecte une condition d'erreur dans une instruction Transact-SQL incluse dans un bloc TRY, le contrôle est transmis à un bloc CATCH où l'erreur peut être traitée.

Une fois que le bloc CATCH a traité l'exception, le contrôle est transféré à la première instruction Transact-SQL qui suit l'instruction END CATCH. Si l'instruction END CATCH représente la dernière instruction d'une procédure stockée ou d'un déclencheur, le contrôle

Programmation des SGBDR

est retourné au code qui a appelé la procédure stockée ou le déclencheur. Nous verrons ce point lors de la programmation de ces objets sur le serveur.

Les instructions Transact-SQL du bloc TRY suivant l'instruction qui génère une erreur ne sont pas exécutées.

En l'absence d'erreurs dans le bloc TRY, le contrôle est transmis à l'instruction située immédiatement après l'instruction END CATCH associée.

Si l'instruction END CATCH est la dernière instruction d'une procédure stockée ou d'un déclencheur, le contrôle est transmis à l'instruction qui a appelé la procédure stockée ou le déclencheur.

Un bloc TRY commence par l'instruction BEGIN TRY et finit par l'instruction END TRY. Vous pouvez spécifier une ou plusieurs instructions Transact-SQL entre les instructions BEGIN TRY et END TRY.

Un bloc TRY doit être immédiatement suivi d'un bloc CATCH. Un bloc CATCH commence par l'instruction BEGIN CATCH et finit par l'instruction END CATCH.

Dans Transact-SQL, chaque bloc TRY est associé à un seul bloc CATCH.

5.2 Mise en œuvre de la gestion des exceptions

Chaque construction TRY...CATCH doit être contenue dans un lot, une procédure stockée ou un déclencheur unique. Par exemple, vous ne pouvez pas insérer un bloc TRY dans un lot et le bloc CATCH associé dans un autre lot. Le script suivant génère une erreur :

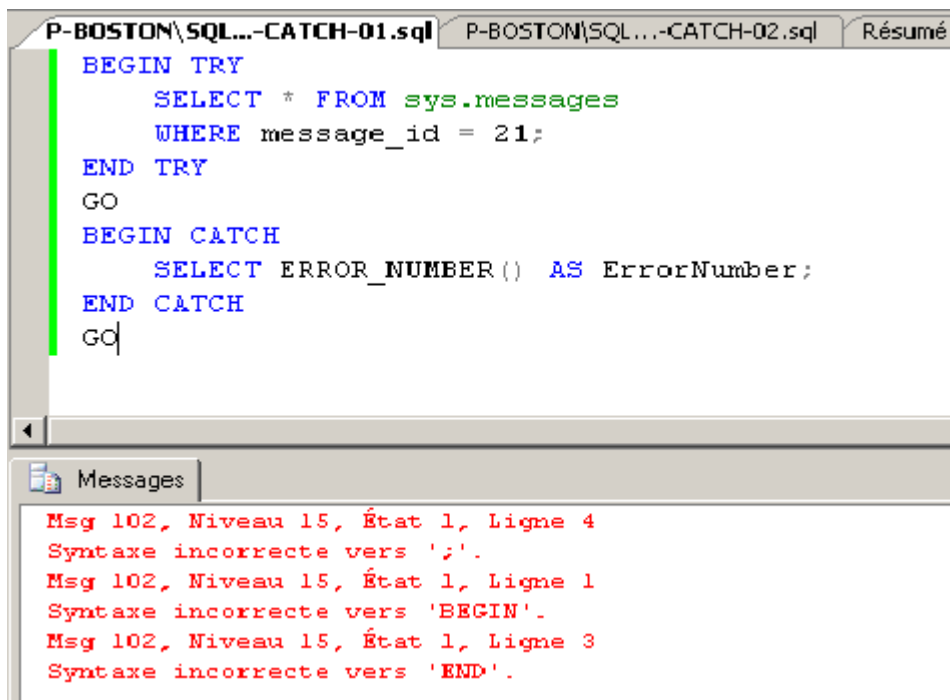


Figure 8 : Try Catch - Bloc invalide / 2 lots

La construction valide serait :

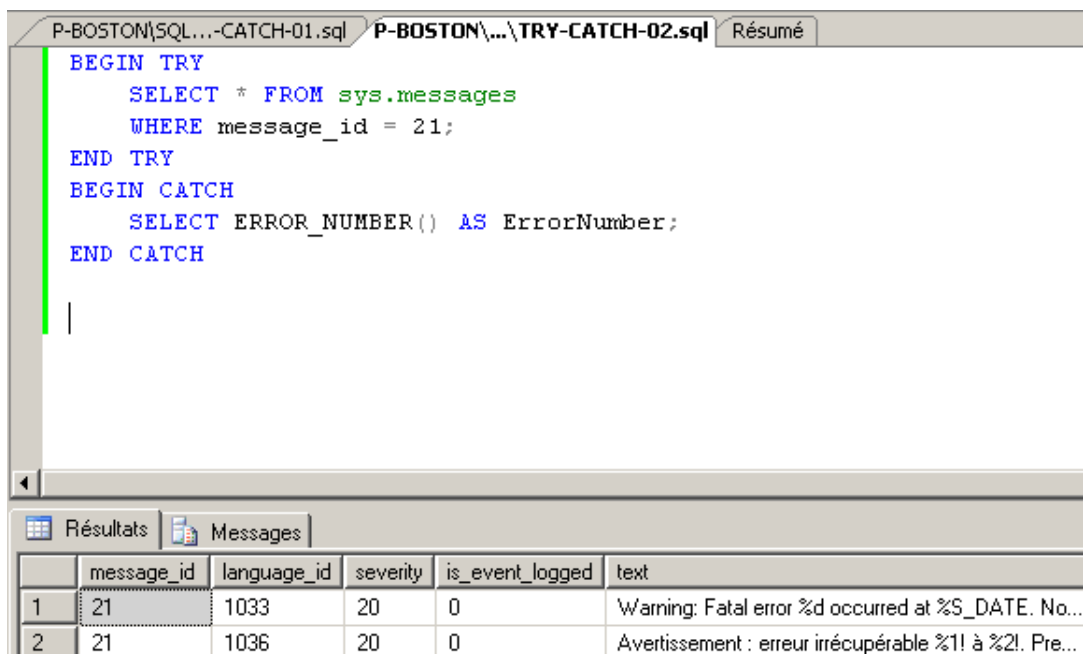


Figure 9 : Try-Catch Bloc valide

Pour tout bloc TRY doit correspondre un et un seul bloc CATCH

Un bloc TRY doit être immédiatement suivi d'un bloc CATCH.

Imbrication des blocs TRY CATCH

Les constructions TRY...CATCH peuvent être imbriquées. Cela signifie que les constructions TRY...CATCH peuvent être insérées dans d'autres blocs TRY et CATCH.

Lorsqu'une erreur se produit dans un bloc TRY imbriqué, le contrôle du programme est transféré au bloc CATCH associé au bloc TRY imbriqué.

Par exemple, vous pouvez vouloir gérer une erreur qui se produit dans un bloc CATCH donné. Ecrivez alors un bloc TRY.....CATCH dans le bloc CATCH spécifié.

Niveau de gravité et gestion des exceptions

Les erreurs dont le niveau de gravité est supérieur ou égal à 20 et qui contraignent le moteur de base de données à fermer la connexion ne sont pas gérées par le bloc TRY...CATCH. Toutefois, le bloc TRY...CATCH gère les erreurs dont le niveau de gravité est supérieur ou égal à 20 tant que la connexion n'est pas fermée.

Les erreurs dont le niveau de gravité est inférieur ou égal à 10 sont considérées comme des avertissements ou des messages d'information et ne sont pas gérées par les blocs TRY...CATCH.

5.3 Récupérer les informations sur les erreurs

Le bloc TRY...CATCH utilise les fonctions de gestion des erreurs suivantes pour capturer les informations d'erreur :

- ERROR_NUMBER() retourne le numéro de l'erreur.
- ERROR_MESSAGE() retourne le texte complet du message d'erreur. Le texte comprend les valeurs fournies pour tous les paramètres substituables, tels que les longueurs, les noms d'objets ou les heures.
- ERROR_SEVERITY() retourne le niveau de gravité.
- ERROR_STATE() retourne le numéro d'état de l'erreur.
- ERROR_LINE() retourne le numéro de ligne de la routine à l'origine de l'erreur.
- ERROR_PROCEDURE() retourne le nom de la procédure stockée ou du déclencheur dans lequel l'erreur s'est produite.

Ces fonctions permettent de récupérer les informations d'erreur de n'importe où dans l'étendue du bloc CATCH d'une construction TRY...CATCH. Elles retournent la valeur NULL si elles sont appelées en dehors de l'étendue d'un bloc CATCH.

Elles peuvent être référencées dans une procédure stockée et utilisées pour récupérer les informations d'erreur lorsque la procédure stockée est exécutée dans le bloc CATCH.

De cette manière, vous n'avez pas besoin de répéter le code de gestion d'erreur dans chaque bloc CATCH. Dans l'exemple de code ci-dessous, l'instruction SELECT du bloc TRY génère une erreur de division par zéro. L'erreur est gérée par le bloc CATCH qui utilise une procédure stockée pour retourner les informations d'erreur.

L'exemple suivant vous montre comment récupérer quelques informations sur les erreurs. Nous verrons par la suite comment mettre en place un système de gestion des erreurs plus sophistiqué en encapsulant ces instructions dans une procédure.

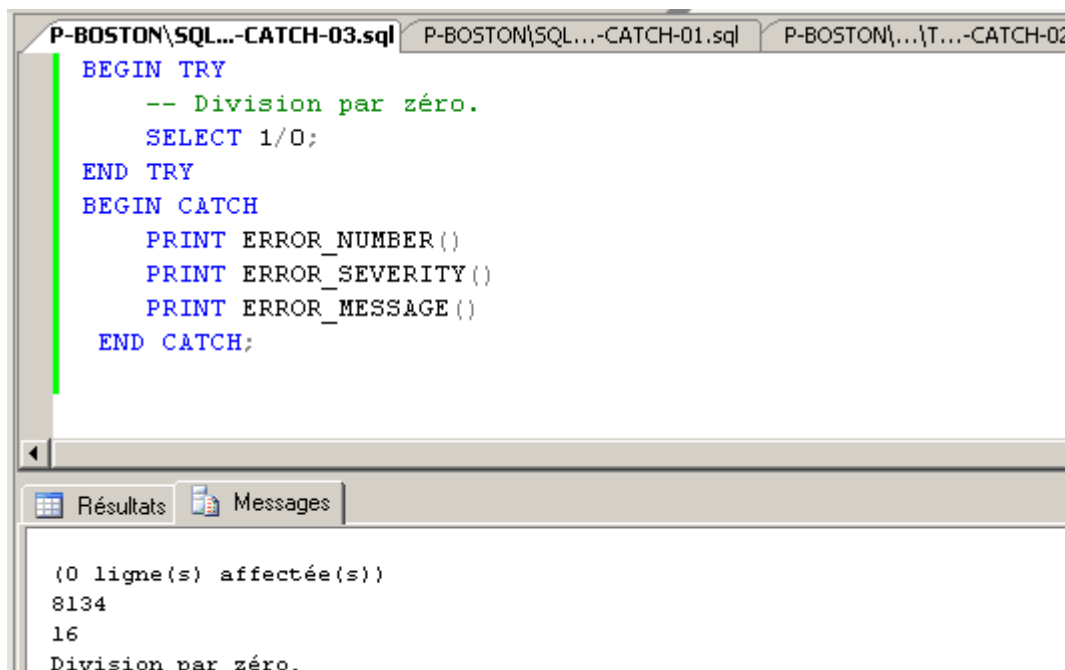


Figure 10 : Récupération des informations relatives à l'erreur

5.4 Définir des conditions propres d'exception

L'instruction RAISERROR vous permet de définir la manière dont les erreurs seront gérées. Vous pouvez utiliser RAISERROR dans le bloc TRY ou CATCH d'une construction TRY...CATCH.

Si RAISERROR a un niveau de gravité compris entre 11 et 19 et qu'il est exécuté dans un bloc TRY, le contrôle est transféré au bloc CATCH associé.

Si RAISERROR possède un niveau de gravité compris entre 11 et 19 et qu'il est exécuté dans un bloc CATCH, il retourne une erreur à l'application ou au lot appelant. Ainsi, RAISERROR permet de retourner à l'appelant des informations sur l'erreur à l'origine de l'exécution du bloc CATCH.

Les informations d'erreur fournies par les fonctions d'erreur TRY...CATCH peuvent être capturées dans le message RAISERROR, notamment le numéro d'erreur d'origine ; toutefois, le numéro d'erreur de RAISERROR doit être supérieur ou égal à 50000.

Les numéros d'erreur compris entre 1 et 50000 sont réservés au système.

Vos propres erreurs devront donc avoir un numéro supérieur à 50 000.

Les règles pour vos propres exceptions ne sont pas différentes de celles relatives aux exceptions prédéfinies sur le système. Ainsi :

- Si RAISERROR possède un niveau de gravité inférieur ou égal à 10, il retourne un message d'information à l'application ou au lot appelant sans solliciter un bloc CATCH.
- Si RAISERROR possède un niveau de gravité supérieur ou égal à 20, il met fin à la connexion à la base de données sans appeler le bloc CATCH.

Vous pouvez prédéfinir vos exceptions dans le fichier Sysmessages du système.

Les règles de construction des messages sont les mêmes qu'ils soient prédéfinis ou définis lors de l'exécution.

Programmation des SGBDR

Pour cela, vous pouvez avoir recours à l'interface graphique si vous disposez de la console de management de SQL Serveur ou à la procédure stockée système `sp_Addmessage` dans tous les cas de figure.

- `msg_id` : Numéro du message d'erreur défini par l'utilisateur et stocké dans l'affichage catalogue `sys.messages` à l'aide de `sp_addmessage`. Lorsque l'argument `msg_id` n'est pas spécifié, `RAISERROR` génère un message d'erreur portant le numéro 50 000.
- `msg_str` : Message d'erreur défini par l'utilisateur dont la mise en forme est proche de la fonction `printf` dans la bibliothèque standard C. Le message d'erreur peut compter jusqu'à 2 047 caractères.
- Lorsque l'argument `msg_str` est spécifié, `RAISERROR` génère un message d'erreur portant le numéro 50 000.

Mise en forme de la chaîne `msg_str`

`Msg_str` est une chaîne de caractères dotée de spécifications de conversion incorporées facultatives. Chaque spécification de conversion définit la manière dont une valeur de la liste d'arguments est mise en forme et placée dans un champ à l'emplacement de la spécification de conversion dans `msg_str`.

Les spécifications de conversion arborent la mise en forme suivante :

% *[[flag] [width] [. precision] [{h | l}] type*

Nous ne verrons ici que les utilisations les plus courantes, pour tout connaître se reporter au manuel de référence de SQL Server.

Argument Flag		
Code	Préfixe ou justification	Description
- (moins)	Cadré à gauche	Cadre la valeur de l'argument à gauche par rapport à la largeur du champ donnée.
+ (plus)	Préfixe	Fait précéder la valeur de l'argument d'un signe positif (+) ou négatif (-) si celle-ci est de type signé.
0 (zéro)	Remplissage avec des zéros	Fait précéder le résultat de zéros jusqu'à atteindre la largeur minimale. Si zéro et le signe moins (-) sont donnés ensemble, le zéro est ignoré.
' ' (blanc)	Remplissage avec des espaces	Fait précéder la valeur de sortie d'espaces si la valeur est signée et positive. Ceci sera ignoré si un signe positif (+) est inclus dans le drapeau +.

Argument Width	
Code	Description
Nombre entier	Définit la largeur minimale du champ dans lequel est placée la valeur de l'argument. Si la longueur de la valeur d'argument est égale ou supérieure à <code>width</code> , la valeur est imprimée sans marge intérieure. Si la valeur est inférieure à <code>width</code> , elle est complétée à concurrence de la longueur indiquée par le paramètre <code>width</code> .
*	La largeur est spécifiée par l'argument associé dans la liste, qui doit être un nombre entier.

Précision	
Code	Description
Nombre entier	Nombre maximum de caractères extraits de l'argument pour les chaînes. Par exemple, si une chaîne compte cinq caractères et que la précision est égale à 3, seuls les trois premiers caractères de la chaîne seront utilisés. Pour les nombres entiers, le paramètre précision correspond au nombre minimum de chiffres imprimés.

Argument {h l} type	
Code	Description
d ou i	Entier signé
s	Chaîne
u	Entier non signé

Illustration avec un script de contrôle du niveau de remise accordé au client.

```

P-BOSTON\...\T...-CATCH-04.sql* P-BOSTON\SQL...-CATCH-01.sql P-BOSTON\...\T...-CATCH-02.sql* Résumé
declare @pourcentremise as decimal(5,2)
set @pourcentremise = 55.2

BEGIN TRY
    if @pourcentremise > 40
    Begin
        declare @pourcentChaine as nvarchar(10)
        SET @pourcentChaine = CAST(@pourcentremise as NVARCHAR) + ' %'
        RAISERROR(N'Le client a un pourcentage de remise excessif de %s .',
            11, -- Sévérité,
            1, -- état,
            @pourcentChaine)
    END
END TRY
BEGIN CATCH
    PRINT ERROR_NUMBER()
    PRINT ERROR_SEVERITY()
    PRINT ERROR_MESSAGE()
END CATCH;

```

Messages

```

50000
11
Le client a un pourcentage de remise excessif de 55.20 % .

```

Figure 11 : illustration de la construction d'un message personnalisé

Vous noterez que, ne pouvant afficher d'autres valeurs que des entiers, nous devons convertir la valeur décimale en chaîne avant de l'introduire comme paramètre de substitution.

6 Programmer des procédures stockées

Les procédures stockées permettent de disposer de méthodes encapsulées au niveau du serveur qui prendront en charge les transactions sur les données.

Différents avantages peuvent être tirés de cette approche :

- Le code transactionnel n'est pas disséminé dans les différents composants logiciels manipulant les données mais centralisé sur le serveur.
- Cette approche permet la réutilisation de code. Par exemple, quel que soit le composant qui demande l'ajout d'un client, il devra nécessairement le faire via la procédure stockée ad hoc.

Cette approche permet d'isoler la couche transactionnelle des autres couches (métier, interface). Dans certaines sociétés, c'est aussi le moyen pour le DBA (Administrateur des Bases de Données) de s'assurer du bon respect des règles et contraintes de mise à jour des données.

La mise au point est facilitée : une fois les tests unitaires des procédures stockées réalisés et leur conformité aux règles de gestion énoncées vérifiées, ces dernières peuvent être utilisées en toute confiance.

Cette approche apporte un niveau de sécurité supplémentaire. Dès lors que les opérations de mise à jour se font via des procédures stockées, il n'est plus nécessaire d'accorder des droits de modification des données sur les tables aux utilisateurs. Cette approche restreint l'usage d'interfaces non prévues pour la mise à jour des données (Access CS, Excel, ...)

Une procédure stockée peut inclure n'importe quelle instruction SQL en dehors des instructions permettant la création de règles, de vues ou de programmes déclencheurs. Elle est de ce point de vue beaucoup plus souple que les fonctions intégrées.

Les procédures stockées admettent des paramètres en entrée et sortie. Elles peuvent aussi retourner une valeur scalaire (on parle alors de code retour) et/ou un jeu d'enregistrements dans le cadre d'une procédure utilisée pour sélectionner des données.

La généralisation du codage des transactions sous forme de procédures stockées peut toutefois rendre la portabilité de vos applicatifs plus difficile.

En effet, bien que la notion de procédure stockée existe sur tout SGBDR réseau (par opposition à SGBD fichier tel que Access), les conditions de mise en œuvre et la codification peuvent être quelque peu différentes.

Mais il est tout à fait envisageable de s'appuyer sur des générateurs de procédures stockées afin de réduire le temps à consacrer à leur réalisation.

Nous aurons l'occasion de découvrir un exemple de ce type d'outils.

6.1 Les différents types de procédures stockées

Il existe 3 types de procédures stockées :

- Les procédures de type système qui résident dans la base de données Master et sont utilisées pour des tâches d'administration : préfixe SP_.
- Les procédures de type système, mais étendues, qui résident dans la base de données Master et sont utilisées pour des tâches diverses mais hors services SQL. Elles sont implémentées comme des DLL. Leur nom est précédé d'un préfixe XP_. Par exemple, nous pouvons citer la procédure xp_cmdshell qui permet l'exécution de commandes du système d'exploitation.
- Les procédures stockées locales. Elles sont définies par l'utilisateur et sont stockées dans les bases de données utilisateur.

6.2 Codification des procédures stockées

La conception des procédures stockées doit obéir aux mêmes règles que celles relatives à la conception des fonctions et procédures définies dans d'autres langages : la procédure stockée doit effectuer une et une seule tâche. Ainsi, si nous devons programmer les opérations de maintenance des données de la table client, nous devrons coder 3 procédures relatives à l'ajout, la modification et la suppression d'un client.

Comme pour les autres objets SQL, nous associerons les mots clefs CREATE, ALTER ou DROP, au type de l'objet PROCEDURE et à son nom, afin de créer, modifier ou supprimer une procédure.

```
CREATE PROCEDURE PS_Client_Supprimer
@CustomerID nchar(5)
AS
/* Code */

ALTER PROCEDURE PS_Client_Supprimer
@CustomerID nchar(5)
AS
/* Code */

DROP PROCEDURE PS_Client_Supprimer
```

Figure 12 : Opérations de création et maintenance des Procédures stockées

6.3 Définition des paramètres

Les procédures stockées acceptent des paramètres dont il sera nécessaire de définir les caractéristiques :

- Le nom du paramètre, préfixé d'un @ pour l'utiliser comme variable dans le programme.
- Le type de données choisi parmi les types SQL ou les types définis par l'utilisateur
- Une valeur par défaut optionnelle
- La direction, par défaut en entrée. Pour définir un paramètre en sortie, il sera nécessaire de lui associer le mot clé OUTPUT

Par convention, lorsque le paramètre s'apparente à la valeur d'une colonne, il portera un nom identique à celle-ci. Dans l'exemple précédent, la procédure reçoit en entrée la valeur de l'identifiant de la table Customers CustomerID.

Le paramètre sera donc nommé @CustomerID.

Il existe un paramètre en sortie défini par défaut pour toute procédure stockée @RETURNVALUE qui reçoit la valeur de l'opération RETURN et la transmet au programme appelant. La valeur retournée est une valeur de type entier qui prend, par convention, 0 lorsque l'objectif est atteint et une valeur négative dans les autres cas. Nous éviterons les valeurs de la plage -1 à -100 réservées pour le système.

Les paramètres doivent être définis en entête de la procédure avant la clause AS qui délimite le début du code implémenté dans la procédure.

```
create procedure dbo.psCCP_insert
( @IdCCP          int output
, @NomCCP         varchar(50)
, @Descriptif     text = NULL )
as
```

Figure 13 : Paramètres d'une procédure stockée

Vous noterez ici le caractère optionnel du paramètre @Descriptif qui, par défaut, prendra la valeur NULL.

Le paramètre @IdCCP est défini en sortie car nous souhaitons récupérer dans ce dernier une valeur attribuée par le système à la colonne identifiant de la table CCP. La colonne a comme attribut la propriété IDENTITY (compteur).

6.4 Exemples de procédures stockées

L'exemple suivant représente une opération d'insertion sur une table dont la valeur de la clé primaire est affectée par le système.

```
create procedure dbo.psCCP_insert
( @IdCCP          int output
, @NomCCP         varchar(50)
, @Descriptif     text = NULL )
as

insert dbo.CCP
( NomCCP
, Descriptif
) values (
    @NomCCP
, @Descriptif
)

set nocount on
select @IdCCP = SCOPE_IDENTITY()
return 0
```

Figure 14 : Procédure d'insertion d'une ligne

A noter :

Utilisation de NOCOUNT :

Le positionnement de NOCOUNT à ON permet d'éviter que l'instruction SELECT ne comptabilise les lignes affectées par la sélection. En fait, ce qui intéresse le programme appelant est de connaître le nombre de lignes affectées par l'opération INSERT (ici 1).

Utilisation de la fonction SCOPE_IDENTITY() :

Cette fonction permet de récupérer la dernière valeur affectée à une colonne identité lors de la dernière instruction INSERT exécutée.

Valeur retournée

Par convention, lorsqu'une procédure s'exécute correctement, la valeur retournée est 0.

Le deuxième exemple est une procédure d'insertion dans une table dont les valeurs des clés primaires ne sont pas allouées par le système. On doit alors vérifier qu'il n'existe pas de ligne avec une valeur de clé identique à celle que l'on souhaite insérer.

```
create procedure dbo.psCustomers_insert
(
    @CustomerID          char(5)
    , @CompanyName        varchar(40) = NULL
    , @ContactName        varchar(30) = NULL
    , @ContactTitle        varchar(30) = NULL
    , @Address            varchar(60) = NULL
    , @City              varchar(15) = NULL
    , @Region            varchar(15) = NULL
    , @PostalCode         varchar(10) = NULL
    , @Country            varchar(15) = NULL
    , @Phone              varchar(24) = NULL
    , @Fax                varchar(24) = NULL
)
as

if exists (select 1 from dbo.Customers
where CustomerID = @CustomerID )
    return 1

insert dbo.Customers
(
    CustomerID
    , CompanyName
    , ContactName
    , ContactTitle
    , Address
    , City
    , Region
    , PostalCode
    , Country
    , Phone
    , Fax
) values (
    @CustomerID
    , @CompanyName
    , @ContactName
    , @ContactTitle
    , @Address
    , @City
    , @Region
    , @PostalCode
    , @Country
    , @Phone
    , @Fax
)

return 0
```

Figure 15 : Insertion d'une ligne avec contrôle valeur clé primaire

A noter :

Utilisation de l'instruction EXISTS :

Elle permet de s'assurer qu'aucune ligne ne figure dans la table avec une valeur de clé identique.

Le troisième exemple est une procédure réalisée pour exécuter des tâches d'administration.

```
CREATE PROCEDURE PS_Transferer_FichierSauvegarde
@NomBase sysname
AS
/* Copie du fichier via le réseau Appel au shell de l'OS
via la procédure étendue xp_cmdshell */

DECLARE @cmd sysname, @cible sysname, @source sysname
SET @source = 'E:\BACKUP\' + @nombase + '.bak'
SET @cible = ' \\P-HENAFF\SQLServer\' + @nombase+'.bak'
SET @cmd = 'copy ' + @source + @cible

EXEC master..xp_cmdshell @cmd
```

Figure 16 : Procédure de copie de fichier via le réseau

A noter :

La construction de la commande du système d'exploitation par concaténation de constantes de type chaîne et de variables.

Le recours à la procédure stockée étendue xp_cmdshell qui permet d'exécuter des commandes de l'OS.

6.5 Appel des procédures stockées

Les procédures stockées sont exécutées à l'aide de l'instruction EXECUTE ou EXEC. Il est nécessaire de respecter quelques principes de base lors du passage ou la réception de valeurs en arguments de procédure.

Les paramètres peuvent être passés selon deux façons :

- Par position : ce procédé est conforme aux standards SQL.
- Par référence nommée : ce procédé est réservé aux technologies Microsoft.

6.6 Passage des paramètres par position

Soit la procédure stockée suivante qui prend en charge la suppression des clients. Cette suppression ne sera réalisée que si aucune commande ne lui est associée.

Dans le cas contraire, la valeur de retour -101 sera retournée et la demande de suppression abandonnée.

```
-- =====
-- Author:      Bost
-- Create date: 11/07/2017
-- Description: Suppression des clients sans commande
-- =====
CREATE PROCEDURE Ps_Supprimer_Client
    @CustomerID char(5)
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    IF (select COUNT(OrderID) from Orders where CustomerID=@CustomerID) is not null
    begin
        return -101
    end
    SET NOCOUNT OFF;
    delete Customers where CustomerID=@CustomerID
    return 0
END
GO
```

Et la demande d'exécution de cette procédure

```
declare @retValue int;
Declare @IDCustomer char(5);

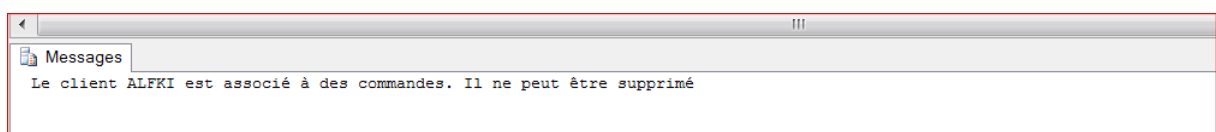
set @IDCustomer='ALFKI'

execute @retValue = Ps_Supprimer_Client @IDCustomer
if @retValue = 0
begin
    Print 'Suppression réussie'
end
if @retValue = -101
begin
    Print 'Le client ' + @idcustomer + ' est associé à des commandes. Il ne peut être supprimé'
end
```

Figure 17 : Récupération de la valeur retournée par la procédure

La valeur de retour est récupérée dans une variable @RetVal qui doit être insérée entre l'instruction EXECUTE et le nom de la procédure stockée.

Un message est imprimé à la console qui précise le résultat obtenu.



Programmation des SGBDR

Dans ce deuxième exemple, nous souhaitons modifier le statut du compte pour le bloquer ou le débloquent. Vous remarquerez l'usage de la fonction CAST pour produire une chaîne à partir d'un booléen et utiliser le résultat dans une opération de concaténations de chaînes.

```
-- =====
-- Author:      Bost
-- Create date: Juillet 2017
-- Description: Modification du statut bloqué on/off
-- =====
CREATE PROCEDURE [dbo].[PS_Utilisateur_ChangerStatut]

@IDUtilisateur as nvarchar(50),
@CompteBloque bit,
@Message as varchar(255) output

AS
BEGIN

    SET NOCOUNT ON;
    -- Récupération des valeurs du compte
    IF (SELECT 1 FROM Utilisateurs WHERE IDUtilisateur=@IDUtilisateur) is null
    BEGIN
        set @Message = 'Impossible de modifier le statut du compte de l''utilisateur '
        + @IDUtilisateur + ' car ce compte n''existe pas ';
        return -101
    END
    SET NOCOUNT OFF;
    UPDATE Utilisateurs SET CompteBloque=@CompteBloque WHERE IDUtilisateur = @IDUtilisateur;
    set @Message = 'Le compte de l''utilisateur ' + @IDUtilisateur + ' a été passé à '
    + CAST(@CompteBloque as varchar)
    RETURN 0
END
```

Figure 18 : Exemple avec des paramètres en sortie

Dans cet exemple, nous avons 4 paramètres : 2 en entrée/sortie, 1 en sortie et celui correspondant à la valeur retournée. Les différents paramètres doivent être séparés par des virgules et le mot clé OUTPUT associé à la variable qui recevra la valeur du paramètre en sortie. L'image suivante illustre ces propos.

```
SQLQuery5.sql - I...-Perso\Bost (52))* Changer Statut Ut...T-Perso\Bost (58))
DECLARE @RC int
DECLARE @IDUtilisateur nvarchar(50)
DECLARE @CompteBloque bit
DECLARE @Message varchar(255)

-- À faire : définir des valeurs de paramètres ici.
set @IDUtilisateur = 'Bost';
set @CompteBloque = 0;

EXECUTE @RC = [ComptoirAnglais_V1].[dbo].[PS_Utilisateur_ChangerStatut]
    @IDUtilisateur
    ,@CompteBloque
    ,@Message OUTPUT

PRINT @message
Print @RC
```

Figure 19 : Utilisation de paramètres en sortie

6.7 Passage des paramètres par référence nommée

Cette approche, qui n'est pas conforme à la norme SQL mais présente dans l'ensemble de l'architecture des produits Microsoft, permet de ne pas se soucier de l'ordre de déclaration des paramètres.

Elle offre aussi l'avantage de ne pas avoir à se soucier des paramètres optionnels qui, sans valeur mentionnée, prendront la valeur par défaut définie dans la procédure.

```
create procedure dbo.psCustomers_insert
(
    @CustomerID          char(5)
    , @CompanyName        varchar(40) = NULL
    , @ContactName        varchar(30) = NULL
    , @ContactTitle       varchar(30) = NULL
    , @Address            varchar(60) = NULL
    , @City               varchar(15) = NULL
    , @Region            varchar(15) = NULL
    , @PostalCode         varchar(10) = NULL
    , @Country            varchar(15) = NULL
    , @Phone              varchar(24) = NULL
    , @Fax               varchar(24) = NULL )
as
```

Figure 20 : Procédure avec de nombreux paramètres optionnels

```
Declare @RetVal int
EXECUTE @RetVal = psCustomers_Insert
    @CustomerId = 'Bost',
    @CompanyName = 'Bost''''on Corporation',
    @City = 'Brive'
```

Figure 21 : Passage par référence nommée des paramètres

A noter : Cette approche doit être réservée aux technos Microsoft. Le recours à cette technique est pratique lorsque de nombreux paramètres optionnels sont présents.

7 Programmation des fonctions

Vous avez déjà eu l'occasion de vous familiariser avec les fonctions définies au niveau du système telles que celles opérant sur des dates, des chaînes ou des types.

Vous aurez éventuellement recours aux fonctions mathématiques disponibles telles ABS (valeur absolue), LOG (logarithme népérien), TAN (Tangente), ...

Mais vous exprimerez peut-être aussi le besoin de disposer de fonctions propres inexistantes sur le système. Je vous propose donc de vous initier à la réalisation de ces dernières au travers d'exemples que nous mettrons en œuvre dans cette séance de formation.

La syntaxe diffère peu de celle mise en œuvre au sein des procédures stockées, qui seront vues ultérieurement, mais les conditions d'utilisation sont bien différentes.

Une fonction est utilisée en interne au sein de programmes ou scripts alors que la procédure stockée sera le plus souvent associée à un composant logiciel tiers développé dans un langage évolué.

Il existe deux types de fonction :

- Le premier type permet de renvoyer une valeur scalaire
- Le deuxième type permet de renvoyer une table.

Deux exemples sont proposés dans les pages suivantes :

Dans le premier exemple, il s'agit d'une fonction scalaire de calcul d'écart entre deux dates. Elle est dite scalaire car elle ne peut retourner qu'une valeur unique.

Dans le deuxième exemple, la fonction permet de renvoyer une table temporaire au point d'appel en fonction de critères transmis en arguments à celle-ci.

Plus souple que le recours à une vue dans la mesure où des paramètres peuvent être passés et pris en compte dans la construction du jeu de résultats.

7.1 Syntaxe de la création d'une fonction

```
CREATE FUNCTION NomDeLaFonction
(
paramètres
)
RETURNS définition du type de la valeur retournée
AS

BEGIN
Code de la fonction
...
RETURN valeur de retour

END
```

A noter : Il existe de nombreuses restrictions quant à l'usage de fonctions intégrées dans une fonction définie par l'utilisateur. Ainsi, les fonctions intégrées susceptibles de renvoyer des données différentes d'un appel à l'autre ne sont pas autorisées :

@@CONNECTIONS	@@PACK_SENT	GETDATE
@@CPU_BUSY	@@PACKET_ERRORS	GetUTCDate
@@IDLE	@@TIMETICKS	NEWID
@@IO_BUSY	@@TOTAL_ERRORS	RAND
@@MAX_CONNECTIONS	@@TOTAL_READ	TEXTPTR
@@PACK_RECEIVED	@@TOTAL_WRITE	

Figure 22 : Fonctions intégrées interdites

7.2 Fonction Scalaire

Calcul de l'écart entre deux dates

```
CREATE FUNCTION CalculEcartDates
(
  @DateDebut datetime,
  @DateFin datetime,
  @Unite varchar(25)
)
-- la variable de retour est de type entier
RETURNS int
AS
BEGIN
  DECLARE @ChaineSQL varchar(250)
  DECLARE @Ecart int

  SET @Ecart =
    CASE
      When (@Unite = 'Minute') THEN ABS(DATEDIFF(Minute, @DateDebut, @DateFin))
      When (@Unite = 'Heure') THEN ABS(DATEDIFF(Hour, @DateDebut, @DateFin))
      When (@Unite = 'Jour') THEN ABS(DATEDIFF(Day, @DateDebut, @DateFin))
    END
  RETURN @Ecart
END
```

Figure 23 : Fonction de calcul d'écart entre deux dates

```
use Northwind
select OrderID, CustomerID, dbo.CalculEcartDates(OrderDate, ShippedDate, 'Jour') As DelaiLivraison from Orders
where dbo.CalculEcartDates(OrderDate, ShippedDate, 'Jour') > 10
```

	OrderID	CustomerID	DelaiLivraison
1	10248	VINET	12
2	10254	CHOPS	12
3	10261	QUEDE	11
4	10264	FOLKO	30
5	10265	BLONP	18
6	10271	SPLIR	29

Programmation des SGBDR

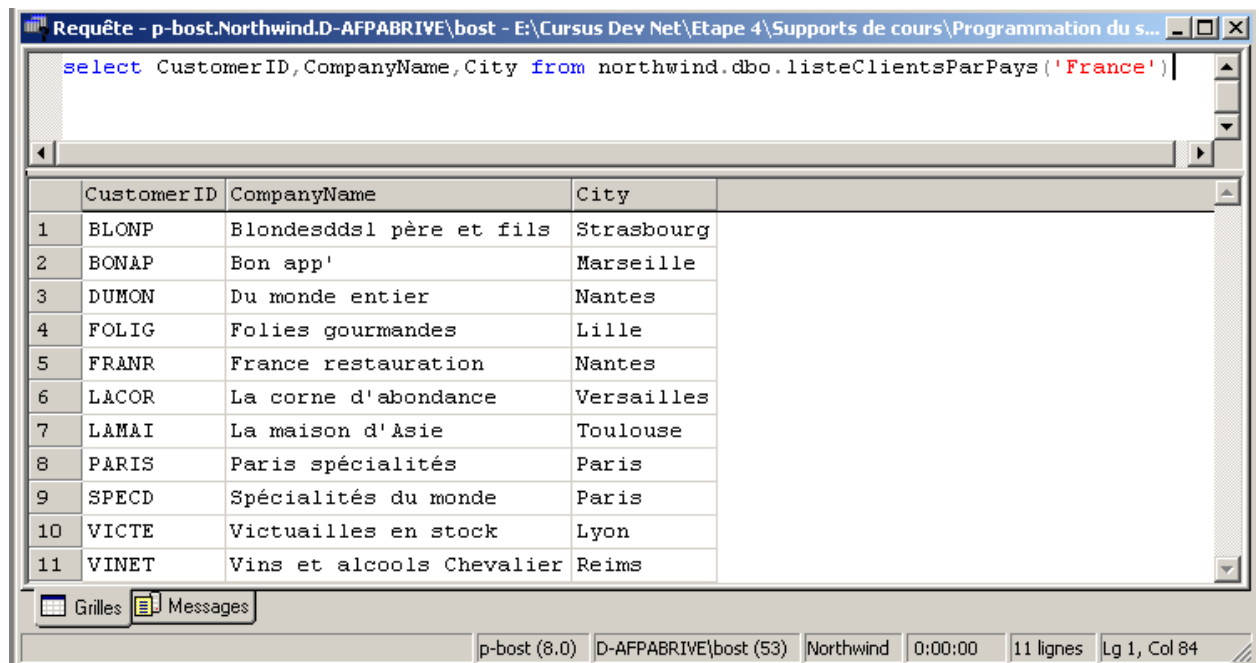
7.3 Fonction table

Client par pays

```
create function ListeClientsParPays
(
    @Pays varchar(15)
)
RETURNS TABLE
AS

RETURN(SELECT * FROM Customers WHERE Country = @Pays)
```

Figure 24 : Fonction renvoyant une table



The screenshot shows a SQL query window titled "Requête - p-bost.Northwind.D-AFPABRIVE\bost - E:\Cursus Dev Net\Etape 4\Supports de cours\Programmation du s...". The query entered is `select CustomerID,CompanyName, City from northwind.dbo.listeClientsParPays('France')`. The results are displayed in a table with 4 columns: CustomerID, CompanyName, and City. The table contains 11 rows of data for French customers.

	CustomerID	CompanyName	City
1	BLONP	Blondesdds1 père et fils	Strasbourg
2	BONAP	Bon app'	Marseille
3	DUMON	Du monde entier	Nantes
4	FOLIG	Folies gourmandes	Lille
5	FRANR	France restauration	Nantes
6	LACOR	La corne d'abondance	Versailles
7	LAMAI	La maison d'Asie	Toulouse
8	PARIS	Paris spécialités	Paris
9	SPECD	Spécialités du monde	Paris
10	VICTE	Victuailles en stock	Lyon
11	VINET	Vins et alcools Chevalier	Reims

The bottom of the window shows a status bar with the following information: p-bost (8.0) | D-AFPABRIVE\bost (53) | Northwind | 0:00:00 | 11 lignes | Lg 1, Col 84.

Figure 25 : Résultats renvoyés par la fonction

A noter : Lors de l'appel d'une fonction, il est nécessaire de préciser la base et le propriétaire de celle-ci, ici `northwind.dbo.ListeClientsParPays`

8 Programmer la récursivité

Nous avons parfois besoin de mettre au point des fonctions ou algorithmes qui s'invoquent eux-mêmes.

Nous pouvons alors dire que ces méthodes ou algorithmes sont directement récursifs.

Pour comprendre, reprenons le cas du calcul du factoriel d'un nombre positif.

Nous pouvons le traiter selon un mode simple itératif ou selon une approche récursive :

Selon l'approche itérative :

$$N! = 1 * 2 * 3 * \dots * N$$

L'algorithme de la fonction calcul du factoriel d'un nombre positif est :

Fonction entier Factoriel (entier n)

Début

Entier i, Fact ;

Fact <- 1

 Pour i de 1 à n faire

 Fact <- Fact * i

 Fin Pour

Factoriel <- Fact

Fin

Selon l'approche récursive :

Nous exprimons le caractère récurrent du calcul :

$$N! = N * (N-1) !$$

Jusqu'à la condition d'arrêt $N = 0$ avec le factoriel de 0 valant 1

$$0! = 1$$

Fonction entier Factoriel (entier n)

Début

Si $n = 0$ alors

 Factoriel <- 1

Sinon

 Factoriel <- $n * \text{Factoriel}(n - 1)$

Fin si

Fin

Ainsi, pour le calcul de factoriel de 3

Factoriel (3)

 3 × Factoriel (2)

 2 × Factoriel (1)

 1 × Factoriel (0)

Programmation des SGBDR

Avant les évolutions proposées par SQL 3, seule la forme itérative était proposée.

La norme SQL 3 permet de mettre en place des approches récursives basées sur des auto-jointures.

Nous passerons par un nouveau concept de programmation SQL, les CTE ou Common Table Expressions.

Les CTE se présentent sous la forme suivante :

```
With T as ( E ) R
```

Ou :

With T définit la forme de la table temporaire des résultats par son nom et la liste de ses attributs.

E définit l'expression de construction du résultat sous une forme récursive

R définit l'exécution de la requête E et les données devant figurer dans le jeu de résultats.

Voyons de suite un exemple de l'usage de cette forme.

Nous souhaitons calculer la factorielle d'un nombre entier positif n selon l'approche itérative.

$$n! = n * (n-1) !$$

Nous souhaitons obtenir en résultat, la factorielle d'un nombre entier positif.

Nous écrivons le script suivant pour calculer le factoriel de 5 :

```
DECLARE @Nombre AS bigint;
DECLARE @Factoriel AS bigint;

SET @Nombre = 5;

WITH ResFactoriel(Factoriel, cycles)
AS
(
    SELECT Cast(@Nombre as bigint), CAST(@Nombre-1 as Bigint)
    UNION ALL
    SELECT Factoriel * cycles, cycles-1
    FROM ResFactoriel
    WHERE cycles > 1
)
-----
SELECT @Factoriel =Factoriel
FROM ResFactoriel

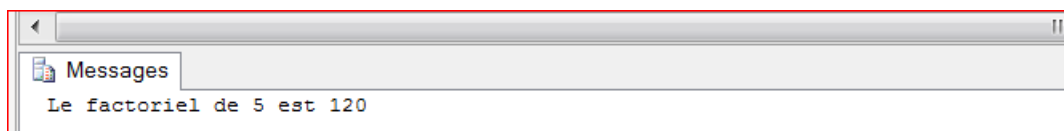
Print 'Le factoriel de ' + Cast(@nombre as Varchar(50)) + ' est ' + cast(@Factoriel as varchar(50));
```

L'expression E0 initialise la table T avec Factoriel = nombre et Cycle = nombre-1

L'expression Ei calcule le factoriel et stocke le résultat.

E0 est exécuté une fois tandis qu'Ei sera itérée autant de fois que de cycles, le nombre de cycles étant déterminé par la valeur du nombre donc on calcule le factoriel.

Vérifions le résultat obtenu en ajoutant en fin de script l'expression suivante :



Programmation des SGBDR

Tests avec borne mini, intermédiaire et grand nombre. Il faudrait ajouter un test sur la qualité de l'entier (doit être positif).

```
La factorielle de 1 est : 1
La factorielle de 5 est : 120
La factorielle de 10 est : 3628800
La factorielle de 18 est : 6402373705728000
```

```
Msg 8115, Niveau 16, État 2, Ligne 9
Une erreur de dépassement arithmétique s'est produite lors de la conversion de expression en
type de données bigint.
La factorielle de 22 est : 2432902008176640000
```

Une autre forme pour l'expression du calcul du factoriel :

```
DECLARE @Nombre AS bigint;
DECLARE @Factoriel AS bigint;

SET @Nombre = 5;

WITH ResFactoriel(Factoriel, cycles)
AS
(
  SELECT Cast(1 as bigint), CAST(1 as Bigint)
  UNION ALL
  SELECT Factoriel * cycles, cycles+1
  FROM ResFactoriel
  WHERE cycles <= @Nombre
)
-----
SELECT @Factoriel =Factoriel
FROM ResFactoriel

Print 'Le factoriel de ' + Cast(@nombre as Varchar(50)) + ' est ' + cast(@Factoriel as varchar(50));
```


Après la découverte de notre objet CTE, nous allons l'employer dans une approche plus intéressante, le calcul de la factorielle ne nécessitant pas de mettre en œuvre ce type de traitement.

Il s'agit ici de traiter du problème de composants composés que l'on retrouve dans bien des modèles de gestion comme par exemple :

- Construire l'organigramme d'une société en mettant en relation un salarié et ses subordonnés.
- Construire une liste de décomposition d'un produit composé d'un ensemble de composants eux-mêmes produits.
- ...

Nous allons prendre un exemple simple avec une pseudo-liste d'articles composés/composants.

Le schéma des données se résume à une table disposant d'une relation sur elle-même.



Produit		
Nom de la colonne	Type condensé	Autorise la valeur Null
CodeProduit	nchar(10)	Non
Libelle	nvarchar(100)	Non
Compose	nchar(10)	Oui

La colonne compose permet de préciser l'article composant de l'article considéré (le père).
La profondeur maximum de décomposition n'est pas définie.

Le jeu d'essai est le suivant :

Table - dbo.Produit		Schéma - P-BO...rks.Diagram_0*	Ré:
	CodeProduit	Libelle	Compose
	A1	Produit A1	NULL
	A11	Produit A11	A1
	A111	produit A111	A11
	A112	Produit A112	A11
	A1121	Produit A1121	A111
	B1	Produit B1	NULL
	B11	Produit B11	B1
	B111	Produit B111	A11
▶*	NULL	NULL	NULL

Nous allons construire une requête résultante constituée de :

- Niveau de décomposition, code produit, libellé, code composé

Nous passerons en paramètre :

- Le code du produit dont nous souhaitons connaître la nomenclature.

La requête d'initialisation retient la ligne du produit dont on doit construire la décomposition.

Le traitement récursif prend ici tout son sens dans **E0 union all E1** :

- La requête d'initialisation E0 est exécutée une fois. Elle retient les premiers éléments qui sont rangés dans la table résultat Nomenclature
- La requête Ei calcule les éléments dépendants des éléments déjà stockés et les ajoute.
- La requête Ei est exécutée tant que des éléments issus de la jointure entre Nomenclature et Produit existent.

Programmation des SGBDR

```
-- Table temporaire nomenclature

with Nomenclature (Niveau,CodeProduit,Libelle,Compose)
AS
(
-- EO : Initialisation Articles racines non composant - Ancrage
Select 0, CodeProduit,Libelle,Compose
from Produit
Where CodeProduit = @CodeProduit

Union All -- EO union all Ei

-- Ei| Requete récursive

Select N.Niveau + 1, P.CodeProduit, P.Libelle,P.Compose
From Nomenclature N Inner join Produit P
on N.CodeProduit = P.Compose

)

-- Exécution et restitution du résultat :

select * from Nomenclature
OPTION (MAXRECURSION 100)
```

A noter : La limitation du nombre d'appels récursifs avec Option MaxRecursion