Transact SQL
SQL SERVER

Khalid Gaber

Pourquoi Transact SQL (T-SQL) ?

- ✓ SQL est un langage non procédural
- ✓ Les traitements complexes sont parfois très difficile à écrire si on ne peut utiliser des variables et les structures de contrôle comme les boucles et les alternatives
- ✓ L'intérêt du T-SQL est de pouvoir mélanger la puissance des instructions SQL (Select, Insert, Update et Delete) avec la souplesse d'un langage procédural dans un même traitement.

Avantages de T

- ✓ Utilisation SQL : interrogation (SELECT), manipulation (INSERT, UPDATE, DELETE), gestion des transactions (COMMIT, ROLLBACK, …).
- ✓ Mise en œuvre des structures procédurales : gestion des variables, séquence, test, boucles.
- ✓ Gestion des curseurs : traitement du résultat d'une requête ligne par ligne.
- ✓ Gestion des erreurs.

Commentaires

Il existe deux façons de mettre des commentaires dans une requête T - SQL :

- ✓ **-- Pour une ligne**
- ✓ **/* Pour plusieurs lignes */**

Gestion des données

Les identifiants

- ✓ Identificateurs T-SQL :
  - sont composés de 1 à 128 caractères.
  - commencent par un des caractères suivants : une lettre , _, @, #.
  - peut contenir lettres, chiffres.
- ✓ Pas sensible à la casse

variables

- Les **variables** sont des zones mémoires nommées permettant de stocker une valeur issues d'une requête ou de calculs, pour une utilisation ultérieure.
- Les variables sont caractérisées par :
  - Leur nom,
  - Leur type, qui détermine le format de stockage et d'utilisation de la variable.
- Les variables doivent être déclarées avant leur utilisation.

Variables

Syntaxe :

DECLARE @Nom_Variable Type_Variable [= **expression**]

Exemple :

```
Declare @X  Varchar(10)
Select @X = 'toto'
Select Upper(@X)
```

Sotie : TOTO

Variables - Exemple

```
Declare @X  int = 20
Print @X
SET @X = @X*2
Print @X
```

Sotie : 20

40

Autre exemple :

```
-- Stockage du nombre de filmes dans une variable
Declare @cpt  int;
Select @cpt= COUNT(*) from Client;
Print @cpt;
print  'Nombre de clients ' +
   convert(varchar(4),@cpt);
```

Sortie : 3

Nombre de clients est : 3

Autre exemple :

```
Declare @cpt int = 0;
Select  @cpt = COUNT(*) from Client;
Select "Nombre de clients est"=@cpt;
```

Sortie :

Nombre de clients

14

Affecter deux variables avec un seul Select

```
DECLARE  @MoyennePrix   numeric(5,2);
DECLARE  @ NombreProduits  int;
SELECT  @MoyennePrix  =  AVG(prix),
   @NombreProduits  =  COUNT(*)
FROM Produit;
select @MoyennePrix , @ NombreProduits;
```

Instructions de contrôle

Bloc : BEGIN ... END

C’est une structure permettant de délimiter une série d’instructions (bloc). Elle est utilisée avec les tests (IF) est les boucles (While).

Syntaxe :

BEGIN

{instruction | bloc}

END

Structure alternative : IF

IF condition

{instruction | bloc}

[ELSE

{instruction | bloc}]

Exemple

```
Declare @CodeProduit  int  =  17;
If  exists (select  *  from Produit
                where idProduit = @ CodeProduit)
   Begin
    Delete From DetailsCommande
    where NumProduit =  @ CodeProduit;
    Print  'Suppression Ok‘;
   End;
Else print  'Pas de Produit pour ce numéro‘;
```

Structure Répétitive

WHILE Condition

{instruction | bloc}

- ✓ L'instruction BREAK permet la sortie de la boucle.
- ✓ L'instruction CONTINUE permet de repartir à la première instruction de la boucle.

Exemple

```
-- On augmente les prix des produits de 10%
   jusqu'à ce que le prix du produit de code 1 soit
   supérieur à 20€.
While (select AVG(Prix) from Produit) < 30)
   Begin
     Update Filme set prix =prix*1.1
     if (Select prix from Filme where Id_Filme = 1)
   > 20 Break;
  End
```

Case (recherche)

- ✓ Renvoie la valeur attribuée en fonction de la valeur de l'expression ou en fonction d'une condition :

```
CASE  [expression]
        WHEN {valeur |  condition} Then
              Valeur_attribuée-1
        [WHEN ...]
        [ELSE Valeur_attribuée-n ]
END
```

Exp: avec valeur de l'expression

```
Select  titre,  LibelleCategorie = CASE id_genre
    When 1 then 'comédie'
    When 2 then 'Action'
    When 3 then 'Romantique'
    When 4 then 'Dramatique'
    Else 'Divers'
    End
 From Filme;
```

Exp: En fonction d'une condition

```
Select titre, 'Designation prix' = CASE
    When  prix < 10  then 'Pas cher'
    When  prix  BETWEEN 10 and 20 then
   'Normal'
    Else 'Très cher'
    End
 From Produit;
```

Gestion des Curseurs

- ✓ La déclaration du curseur permet de définir la requête SELECT et de l'associer à un curseur.
- ✓ L'utilisation de curseur est une technique permettant de traiter ligne par ligne le résultat d'une requête.

Syntaxe:

```
DECLARE  Nom_Curseur  CURSOR
                FOR  SELECT ...
```

Overture d'un curseur

OPEN Nom_curseur

- ✓ L'ordre OPEN permet d'allouer un espace mémoire pour le curseur.

Traitement des lignes : FETCH

- ✓ Les lignes obtenues par l'exécution de la requête SQL sont distribuées une à une par exécution d'un ordre "**FETCH**" inclus dans uns structure répétitive.

Syntaxe :

```
FETCH Nom_curseur  INTO Liste_variables
```

Fermeture et suppression d'un curseur

CLOSE Nom_curseur

- ✓ Cette fermeture doit intervenir dès que possible afin de libérer l'espace mémoire occupé par le curseur.

DEALLOCATE Nom_curseur

- ✓ Supprime le curseur et les structures associées.

Exemple: Curseur

```
DECLARE  C_Produit  CURSOR
    FOR SELECT idProduit, Nom FROM  Produit;
DECLARE  @codeP int;
DECLARE @Nom varchar(50);
OPEN C_Produit;      -- ouverture du curseur
-- Lire la 1ère ligne
```

Suite de l'exemple

```
Fetch C_Produit INTO @codeP, @Nom;
While (@@FETCH_STATUS = 0)
  BEGIN
    print  Convert(varchar(3),@codeP) +'   ' +
    @Nom;
    Fetch C_Produit INTO @codeP, @Nom;
  END;
CLOSE C_Produit;
DEALLOCATE C_Produit;
```

Gestion des exceptions

Gestion des exceptions

- ✓ T-SQL prend en compte deux types d'exceptions. Les exceptions pré-définies (ou systèmes), par exemple la division par zéro, Et les exceptions définies par le programmeur à l’aide de la procédure stockée sp_addmessage.

Introduction

- ✓ Dans tout développement, il est impératif d'avoir une gestion d'erreur.
- ✓ Pour chaque erreur, SQL Server produit un message associé. Les messages d'erreurs sont stockés dans la base Master. On peut lister le contenu de cette table avec la commande :

SELECT * FROM sys.messages;

- ✓ Si l'erreur rencontrée lors de l'exécution n'existe pas dans cette table, elle obtient le code 50000.

Structure des messages

Tous les messages d'erreurs possèdent la même structure et les mêmes champs d'informations :

- ***Numéro*** : chaque message est indentifié de façon unique par un numéro. SQL Server sélectionne le message depuis la table sys.message de la base Master en fonction du numéro de l'erreur et la langue d'installation du serveur.
- ***Message au format texte***.

Structure des messages - suite

- ***Sévérité*** (ou gravité) : c'est indicateur sur la gravité de l'erreur. Les messages avec une sévérité inférieur à 9 sont données simplement à titre informatif. Ils ne sont pas bloquants. Si la gravité est de 0 alors le message n'est pas visible. Cette gravité permet de classer les messages par rapport au risque potentiel associé. Il existe 25 niveaux de gravité. Lagravité entre 11 et 16 indique que l'erreur peut être résolue par l'utilisateur.

Structure des messages - suite

- ***Etat*** : permet de tracer l'origine de l'erreur.
- ***Nom de la procédure*** : si l'erreur est provoquée depuis une procédure stockée, alors son nom est affiché.
- ***Numéro de la ligne :*** numéro de la ligne où se trouve l'erreur.

Déclencher une erreur

- ✓ Le programmeur peut décider de lever un message d'erreur en fonction du comportement du code à l'aide de l'instruction RAISERROR.

Syntaxe:

RAISERROR({identifiant | message},gravité, état);

```
DECLARE  @x int  = 4;
if  @x  < 10
  RAISERROR('%d est petit !!',2,1,@x);
```

Gestion des erreurs

Il existe deux moyens de gérer les erreurs qui peuveut se produire lors de l'execution du code:

- ✓ La première consiste à tester la valeur de la variable systeme @@ERROR après chaque instruction pour savoir si elle est exécutée correctement (= 0) ou non.
- ✓ La deuxième possibilité consiste à regrouper les instructions qui sont susceptibles de lever des erreurs dans un bloc TRY et de centraliser la gestion des erreurs dans un bloc CATCH.

Gestion des erreurs - suite

- ✓ Les blocs TRY et CATCH sont toujours associés. Il n'est pas possible de définir un bloc TRY sans bloc CATCH et réciproquement.

Syntaxe :

```
BEGIN  TRY
    ..............
END  TRY
BEGIN  CATCH
    ..............
END  CATCH
```

Remarque

- Le Try ... Catch permet de capter les erreurs dont la sévérité est supérieure à 10 afin de passer le script dans une partie prévue à cet effet.

Fonctions système

- ✓ ERROR_NUMBER() : renvoie le numéro de l'erreur
- ✓ ERROR_SEVERITY() : renvoie la gravité de l'erreur
- ✓ ERROR_STATE() : renvoie le numéro d'état de l'erreur
- ✓ ERROR_PROCEDURE() : renvoie le nom de la procédure stockée ou du déclencheur dans lequel s'est produit l'erreur
- ✓ ERROR_LINE() : renvoie le numéro de la ligne où se trouve l'erreur
- ✓ ERROR_MESSAGE() : renvoie le texte complet du message d'erreur.

Création d'une erreur de division par 0

```
BEGIN TRY
   print  5/0;
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS NumeroErreur,
       ERROR_SEVERITY() AS GraviteErreur,
       ERROR_STATE() AS EtatErreur,
       ERROR_PROCEDURE() AS ProcedureErreur,
       ERROR_LINE() AS LigneErreur,
       ERROR_MESSAGE() AS MessageErreur;
END CATCH;
```

Triggers (déclencheurs)

CREATE TRIGGER nom_déclencheur

ON { nom_table | nom_vue }

[WITH ENCRYPTION]

{ FOR | **AFTER** | INSTEAD OF }

[INSERT] [,] [UPDATE] [,] [DELETE]

[WITH APPEND] [NOT FOR REPLICATION] **AS**

instruction_SQL...

Triggers - suite

- **La clause *ON*** désigne la table ou la vue concernées par le déclencheur.
- **La clause *WITH ENCRYPTION*** indique que SQL Server crypte le texte de l'instruction *CREATE TRIGGER*.
- **La clause *for*** indique le déclenchement du *Trigger* pour un type d'événement.
- **La clause *AFTER*** indique le déclenchement du *Trigger* suite à un type d'événement.

Triggers - suite

- **La clause *INSTEAD OF*** : le déclencheur s'exécutera avant insertion, mise à jour ou suppression dans la table qui contient le déclencheur.
- **Les instructions *INSERT*, *UPDATE* et *DELETE*** représentent chacun un type de déclencheur respectivement sur une insertion, une mise à jour et une suppression.
- **La clause *WITH APPEND*** : Cette option permet d'ajouter plusieurs déclencheurs sur un même objet et un même ordre SQL.

Triggers - suite

- **La clause *NOT FOR REPLICATION*** : Le déclencheur défini avec cette option ne sera pas pris en compte dans un processus de modification des données par réplication.

- La procédure stockée **sp_helptrigger** permet de connaître les déclencheurs définis sur une table.
- Il est possible de créer plusieurs déclencheurs pour une même table.

INSERTED, DELETED

- ✓ Lors des modifications de données, SQL Server crée des lignes dans des tables de travail ayant la même structure que la table modifiée : les tables INSERTED et DELETED.
- ✓ Lors d'une commende INSERT, la table Inserted contient une copie logique des lignes crées.
- ✓ Lors d'une commende DELETE, les lignes supprimées sont placées dans la table Deleted.

INSERTED, DELETED

- ✓ Lors d'une commende UPDATE, les lignes contenant les modifications sont placées dans la table Inserted, les lignes à modifier dans la table Deleted.
- ✓ Les tables Inserted et Deleted peuvent être accessibles pendant l'execution du trigger.

Déclencheur sur la table ligneCommande

```
CREATE  Trigger MiseAjourStock
ON  DetailsCommande AFTER  INSERT AS
BEGIN
  Declare @quantite  int, @codeprod  int;
  SELECT @quantite=quantite, @codeprod=IdProduit
  FROM INSERTED;
  UPDATE Produit
  SET stock = stock - @quantite
  Where @codeprod = IdProduit;
END;
```