Programmation en SQL : Transact SQL pour SQL Server

Diapositives de Cours

O. DUBOIS

Plan

2

Introduction

3

- SQL Server est un serveur de base de données relationnelle en SQL
- Avec le Transact SQL, il est également possible de définir des traitements procéduraux
  --> procédures et fonctions
- Des traitements procéduraux pourront également être mis en place pour définir des contraintes d'intégrité complexes
  --> déclencheurs

Variables utilisateur

4

- Déclaration de variables (obligatoire avant utilisation)

```
DECLARE @nom_variable type [,...]
```

  - `nom_variable` : nom précédé du caractère `@`
  - `type` : type système ou défini par l'utilisateur

- Affectation des variables

```
SELECT @nom_variable = expr [,...]
SET @nom_variable = expr
```

Variables utilisateur

5

Exemple 1 :

```
DECLARE @mavar char(20);
SET @mavar = 'Ceci est un test';
SELECT @mavar;
```

Exemple 2 :

```
DECLARE @choix char(4);
DECLARE @nom char(30), @prenom char(30);
SELECT @choix='K111';
SELECT @nom=Nom, @prenom=Prenom
  FROM client
  WHERE NCli=@choix
SELECT @nom, @prenom;
```

Variables systèmes

6

- Variables définies par le système et pouvant être utilisées seulement en lecture
  Elles se distinguent des variables utilisateur par le double **@@**
- Ces variables système contiennent de nombreuses informations

Exemples :

`@@LANGUAGE` : Langue actuellement utilisée

`@@SERVERNAME` : Nom du serveur SQL local

`@@VERSION` : Date, numéro de version et type de processeur de la version courante de SQL Server

`@@CURSOR_ROWS` : Nombre de lignes affectées dans le dernier curseur ouvert

`@@ERROR` : Dernier numéro d'erreur généré par le système

`@@FETCH_STATUS` : Contient le statut d'une commande de curseur FETCH

Gestion des lots et des scripts

7

- Un lot d'instructions est un ensemble d'instructions Transact SQL qui sera compilé et exécuté en une seule unité
- Il se termine par la commande `GO`

Syntaxe :

```
GO [count]
```

- `count` : entier positif : le lot qui précède `GO` sera exécuté le nombre spécifié de fois

- Un lot peut comporter n'importe quelle instruction ou série d'instructions ainsi que des transactions
- Amélioration des performances et compilation unique
- En cas d'erreur de syntaxe, aucune instruction n'est exécutée

Gestion des lots et des scripts

8

Certaines restrictions :

- impossible de combiner certaines instructions dans un même lot :
  CREATE PROCEDURE, CREATE TRIGGER, CREATE VIEW
- impossible d'agir sur des définitions de colonne et d'utiliser ces modifications dans un même lot
  valeurs par défaut, contrainte CHECK, ajout de colonnes à une table
- impossible de supprimer un objet et de le recréer dans un même lot

- Les **scripts** sont des ensembles de lots qui seront exécutés successivement à partir d'un fichier texte
  Ces fichiers ont, par convention, l'extension ***.sql**.

Contrôle de flux

9

- Ensemble de fonctionnalités comportant des instructions (RETURN, PRINT, ...) et des structures de contrôles (séquence, alternative, répétitive) qui améliorent l'utilisation des instructions Transact SQL
- Permet à l'utilisateur de contrôler leur exécution

Contrôle de flux
RETURN

10

- Permet de sortir inconditionnellement d'une procédure ou d'une fonction en renvoyant éventuellement une valeur entière

Syntaxe :

```
RETURN [exprn]
```

Exemple : Renvoi d'une valeur d'une procédure pour indiquer le bon fonctionnement :

```
CREATE PROCEDURE p1 AS
  IF (1<2)
   RETURN 0;
  ELSE
   RETURN 1;
```

Contrôle de flux
PRINT

11

- Instruction d'affichage de message (fenêtre *Messages* de SSMS)

Syntaxe :

```
PRINT {'texte'|@variable|@@variablesysteme}
```

Exemple :

```
DECLARE @nb int;
SELECT @nb=COUNT(*) FROM client;
PRINT 'Nombre total de clients:';
PRINT @nb;
```

Contrôle de flux
CASE

12

- Permet d'attribuer des valeurs conditionnelles

Syntaxe :

```
CASE [expression]
      WHEN {valeur|condition} THEN
      valeurattribuee_1
     [...]
     [ELSE valeurattribuee_n]
      END
```

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

Contrôle de flux
CASE

13

Exemple :

```
DECLARE @mention CHAR(2);
DECLARE @note numeric(4,2) = 12.43;
SET @mention =
CASE
    WHEN @note >= 16 THEN 'TB'
    WHEN @note >= 14 THEN 'B'
    WHEN @note >= 12 THEN 'AB'
    WHEN @note >= 10 THEN 'P'
    ELSE 'R'
END
PRINT 'Note = ' + CAST(@note AS CHAR(5)) + ',
   Mention = ' + @mention;
```

Contrôle de flux
BEGIN ... END

14

- Structure de contrôle permettant de délimiter une série d'instructions (bloc)
- Elle est utilisée avec les tests (IF ... ELSE) et les boucles (WHILE)

Syntaxe :

```
BEGIN
    {instruction | bloc}
    ...
END
```

Contrôle de flux
IF ... ELSE

15

- Structure de contrôle alternative qui permet de tester une condition et d'exécuter une instruction ou un bloc si le test est vrai

Syntaxe :

```
IF condition
   {instruction|bloc}
[ELSE]
   {instruction|bloc}
```

Contrôle de flux
IF ... ELSE

16

Exemple :

```
DECLARE @nocli CHAR(10) = 'F111';
IF exists(SELECT * FROM client
          WHERE NCli=@nocli)
  BEGIN
     DELETE FROM commande WHERE NCli=@nocli;
     DELETE FROM client  WHERE NCli=@nocli;
     PRINT 'Suppression OK';
  END
ELSE
  PRINT 'Pas de client pour ce numéro';
```

Contrôle de flux
WHILE

17

- Structure de contrôle répétitive qui permet d'exécuter une série d'instructions tant qu'une condition est vraie

Syntaxe :

```
WHILE condition
      {instruction|bloc}
```

- **BREAK** permet la sortie de la boucle alors que **CONTINUE** permet de repartir à la première instruction de la boucle

Contrôle de flux
WHILE

18

Exemple :

```
DECLARE @augmente numeric(4,2) = 1.01;

WHILE (SELECT AVG(PrixHT) FROM produit) < 150
BEGIN
  UPDATE produit SET PrixHT=ISNULL(PrixHT,0)*@augmente
  IF( SELECT MAX(PrixHT) FROM produit) > 231
    BREAK;
  SELECT @augmente = @augmente + 0.01;
END;
```

Contrôle de flux
OUTPUT

19

- Permet de connaître les lignes affectées par les clauses du DML INSERT, UPDATE ou DELETE
- La valeur des colonnes retournées par l'intermédiaire de la clause OUTPUT est celle après application des contraintes d'intégrité et exécution de l'instruction DML mais avant l'exécution du ou des déclencheurs associés à la table et à l'instruction DML

Syntaxe :

```
OUTPUT [listeColonne] INTO @variable
```

Contrôle de flux
OUTPUT

20

Syntaxe :

```
OUTPUT [listeColonne] INTO @variable
```

- `listeColonne` : représente la liste des colonnes retournées dans la variable
  Les données peuvent être issues directement de la table
  En utilisant les préfixes DELETED et INSERTED, il est possible de voir les données touchées par la suppression/modification et après modification/insertion
- `@variable` : la variable doit être de type table et elle doit être déclarée avant son utilisation dans la clause OUTPUT

Contrôle de flux
OUTPUT

21

```
INSERT INTO table(listeColonne)
OUTPUT [listeColonne] INTO @variable
VALUES (listeValeurs)

DELETE table
OUTPUT [listeColonne] INTO @variable
WHERE condition

UPDATE table SET colonne=valeur
OUTPUT [listeColonne] INTO @variable
WHERE condition
```

L'auto-incrémentation
La propriété IDENTITY

22

- Propriété qui peut être affectée à une colonne numérique entière, à la création ou à la modification de la table et permet de faire générer des valeurs pour cette colonne
- Les valeurs seront générées à la création de la ligne, en partant de la valeur initiale spécifiée (par défaut 1) et en augmentant ou diminuant ligne après ligne d'un incrément (par défaut 1)

Syntaxe :

```
CREATE TABLE nom (colonne typeentier IDENTITY
[(depart, increment)],  ...)
```

L'auto-incrémentation
La propriété IDENTITY

23

- Syntaxe :

```
CREATE TABLE nom (colonne typeentier IDENTITY
[(depart, increment)],  ...)
```

- Il ne peut y avoir qu'une colonne IDENTITY par table
- La propriété IDENTITY doit être définie en même temps que la colonne à laquelle elle est rattachée
- Les fonctions systèmes IDENT_INCR(nom_table), IDENT_SEED(nom_table) et IDENT_CURRENT(nom_table)
  retournent respectivement, la valeur de l'incrémentation, la valeur initiale et la dernière valeur de type identité utilisée par la table nom_table

Exemple :

```
CREATE TABLE categorie (
     CatPro int IDENTITY(100,1) PRIMARY KEY,
     LibelleCatPro varchar(200));
```

L'auto-incrémentation
La propriété IDENTITY

24

Exemple : Utilisation de la clause OUTPUT afin de connaître la valeur affectée par une colonne de type IDENTITY lors de l'ajout d'informations

```
DECLARE @AJOUT table(
      NFac int,
      DateFac datetime,
      NCom varchar(5),
      MontantHT decimal(5,2),
      EtatFac char(2));
INSERT INTO facture (DateFac, NCom, MontantHT, EtatFac)
  OUTPUT inserted.* INTO @AJOUT
  VALUES (getdate(), '30179', 234.67,'EC');

SELECT NFac FROM @AJOUT;
```

L'auto-incrémentation
Les séquences

25

- Une séquence est un compteur défini indépendamment de toute table qui peut être considérée comme un distributeur de numéros paramétrable : valeur de départ, pas d'incrément, valeur maximum et s'il faut recommencer à la valeur de départ lorsque la valeur maximum est atteinte
- Une séquence permet de partager la même suite de numéros entre différentes tables
- Elle présente aussi l'avantage de permettre à l'application de connaître la valeur avant d'exécuter l'instruction INSERT
- Une séquence permet également d'obtenir une série de numéros qui se suivent en appelant la procédure stockée **`sp_sequence_get_range`**
- Toutes les informations relatives aux séquences sont disponibles en interrogeant la table système **sys.sequences**

L'auto-incrémentation
Les séquences

26

- CREATE SEQUENCE permet de créer une séquence
- ALTER SEQUENCE permet de modifier une séquence
- DROP SEQUENCE permet de supprimer une séquence

Syntaxe :

```
CREATE SEQUENCE nomSéquence
 [ AS typeEntier ]
 [ START WITH valeurDeDépart ]
 [ INCREMENT BY pasIncrément ]
 [ MINVALUE valeurMini | NO MINVALUE ]
 [ MAXVALUE valeurMaxi |NO MAXVALUE ]
 [ CYCLE | NO CYCLE ]
 [ CACHE nombreValeursEnCache | NO CACHE ] ;
```

L'auto-incrémentation
Les séquences

27

- Exemple : La séquence `seq_categorie` est définie avec une valeur de départ de 100 et un incrément de 1 :

```
CREATE SEQUENCE seq_categorie
  AS int
  START WITH 100
  INCREMENT BY 1;
```

L'auto-incrémentation
Les séquences

28

- Pour utiliser une séquence, il faut accéder à cet objet afin de connaître quelle est la prochaine valeur disponible
- C'est la fonction NEXT VALUE FOR qui permet de connaître cette valeur
- Cette possibilité permet d'utiliser des séquences à la place de la propriété IDENTITY de façon transparente pour l'utilisateur de la table

Syntaxe :

```
NEXT VALUE FOR nomSequence
```

- Exemple : La séquence `seq_categorie` est utilisée lors de la création d'une nouvelle catégorie :

```
INSERT INTO categorie(CatPro, LibelleCatPro)
VALUES (next value FOR seq_categorie,'SSD');
```

Les procédures stockées

29

- Une procédure stockée est une séquence d'instructions précompilées stockée dans la base de données dont l'exécution peut être demandée par un utilisateur, un programme d'application, un déclencheur ou une autre procédure
- Une procédure stockée peut inclure des paramètres en entrée et en sortie
- Les différents cas d'utilisation de procédures stockées sont les suivants :
  - Enchaînement d'instructions
  - Accroissement des performances
  - Sécurité d'exécution
  - Manipulation des données système
  - Traitements en cascade

Les procédures stockées

30

- Instruction de création d'une procédure stockée (SQL Server) :

```
CREATE PROCEDURE nom[;numero][(param1[,...])]
[WITH [ENCRYPTION] [,] [RECOMPILE]]
AS [BEGIN ]
   corps de la procédure;
[END];
```

  - `nom` : nom de la procédure
  - `numéro` : numéro d'ordre pour des procédures ayant le même nom
  - `param1,...` : paramètre sous la forme :

```
@nom  type  [ VARYING ] [ = valeur ] [ OUTPUT ]
```

    - `OUTPUT` : permet de spécifier un paramètre retourné par la procédure
    - `VARYING` : spécifie le jeu de résultats pris en charge comme paramètre de sortie. S'applique uniquement aux paramètres de type cursor
  - `WITH RECOMPILE` : la procédure sera recompilée à chaque exécution
  - `WITH ENCRYPTION` : permet de crypter le code dans les tables système

Les procédures stockées

31

Exemple : Procédure de suppression du client dont l'identifiant est passé en paramètre

```
CREATE PROCEDURE sup_cli(@nocli CHAR(10)) AS
BEGIN
    IF NOT EXISTS (SELECT * FROM client
                   WHERE NCli=@nocli)
    BEGIN
          PRINT 'Client inexistant';
          RETURN;
    END;
    IF EXISTS (SELECT * FROM commande WHERE NCli=@nocli)
    BEGIN
          PRINT 'Ce client possède des commandes';
          RETURN;
    END;
    DELETE FROM client WHERE NCli=@nocli;
END;
```

Les procédures stockées

32

- Sous SQL Server, l'appel à une procédure stockée est réalisée par l'instruction EXEC :

Ex :

```
EXEC supp_cli K111;
```

Résultat à l'affichage :

Ce client possède des commandes

Les fonctions utilisateur

33

- Une fonction utilisateur (ou UDF pour User Defined Function) est une routine destinée à fournir en sortie, une valeur scalaire ou une table
- Une fonction utilisateur peut être exploitée dans différents contextes :
  - instruction SQL de type DML (INSERT, UPDATE, DELETE et SELECT)
  - paramétrage d'une vue
  - définition d'une colonne dans une table ou d'une contrainte CHECK sur une colonne
  - ...

Les fonctions utilisateur

34

- Une fonction utilisateur ne peut pas contenir :
  - d'instructions de mise à jour de table (INSERT, UPDATE, DELETE)
  - de transaction
  - d'appel à des procédures stockées

  Son but étant d'être utilisée dans une requête, une procédure, un déclencheur ou une fonction
- Une fonction est créée par l'instruction CREATE FUNCTION
- Selon la nature du résultat retourné, on distingue les fonctions scalaires et les fonctions table
- Les fonctions table sont de deux natures : "en ligne" et "multi-instructions"

Les fonctions utilisateur

35

- Instruction de création d'une fonction scalaire :

```
CREATE FUNCTION  nom_fonction ( [ liste_des_paramètres])
RETURNS type_données
[ WITH ENCRYPTION| WITH SCHEMABINDING]
AS BEGIN
    corps de la fonction
    RETURN valeur ;
END;
```

Exemple :

```
CREATE FUNCTION nbre_cmde(@nocli CHAR(10)) RETURNS int AS
    BEGIN
          DECLARE @nbre int;
          SELECT @nbre=COUNT(*)FROM commande
                       WHERE NCli=@nocli;
          RETURN @nbre;
    END;
```

Les fonctions utilisateur

36

- Instruction de création d'une fonction table en ligne (utilise un simple SELECT) :

```
CREATE FUNCTION nom_fonction ( [ liste_des_paramètres]
)
RETURNS TABLE
[ WITH ENCRYPTION| WITH SCHEMABINDING]
AS RETURN (requête_SELECT );
```

Exemple :

```
CREATE FUNCTION ProduitsPetitPrix(@seuil int)
RETURNS TABLE
AS RETURN (     SELECT *
                FROM produit
                WHERE Prix < @seuil);
```

Les fonctions utilisateur

37

- Instruction de création d'une fonction table multi-instructions :

```
CREATE FUNCTION nom_fonction ( [ liste_des_paramètres] )
RETURNS @variable_retour TABLE (nom_colonne type, ...)
[ WITH ENCRYPTION| WITH SCHEMABINDING]
AS BEGIN
    corps de la fonction
RETURN;
END;
```

Les fonctions utilisateur

38

- Appel d'une fonction utilisateur :

Ex :

```
SELECT NPro FROM ProduitsPetitPrix(50);
```

Résultat à l'affichage :

CLE21
CLE22

- Suppression d'une fonction :

```
DROP FUNCTION nom_fonction ;
```

Gestion des curseurs

39

- L'utilisation de curseurs est une technique permettant de traiter ligne par ligne le résultat d'une requête (contrairement au SQL qui traite un ensemble de lignes)
- Les curseurs sont utilisés lorsqu'il faut traiter les lignes individuellement dans un ensemble ou lorsque le SQL ne peut pas agir uniquement sur les lignes concernées
- Les curseurs sont souvent utilisés dans le code associé aux déclencheurs

Gestion des curseurs

40

- Pour pouvoir fonctionner correctement, le curseur a besoin de respecter les étapes suivantes :
  - être correctement défini, c'est-à-dire définir correctement et complètement la requête de type SELECT associée
  - être ouvert, c'est-à-dire exécuter la requête associée et stocker le résultat en mémoire
  - parcourir le jeu de résultat
  - fermer le curseur, ce qui permet de libérer l'espace mémoire occupé par le jeu de résultat
  - désallouer le curseur
- Il faut également définir les variables nécessaires pour stocker les données issues d'une ligne de résultat

Gestion des curseurs
DECLARE CURSOR

41

- Permet la déclaration et la description du curseur

Syntaxe :

```
DECLARE nomcurseur[INSENSITIVE][SCROLL]CURSOR
    FOR SELECT ....
    FOR {READ ONLY|UPDATE[OF liste_colonne]}]
```

- `INSENSITIVE` : seules les opérations sur la ligne suivante sont permises
- `SCROLL` : les déplacements dans les lignes du curseur peuvent être effectués dans tous les sens
- `UPDATE` : précise que des mises à jour vont être réalisées sur la table d'origine du curseur

Un curseur INSENSITIVE avec une clause ORDER BY ne peut pas être mis à jour. Un curseur contenant ORDER BY, UNION, DISTINCT ou HAVING est INSENSITIVE et READ ONLY

Gestion des curseurs
OPEN

42

- Permet de rendre le curseur utilisable et de créer éventuellement les tables temporaires associées
- La variable **`@@CURSOR_ROWS`** est affectée après le **OPEN**
- Compte tenu de l'espace disque et mémoire utilisé et du verrouillage éventuel des données lors de l'ouverture du curseur, cette opération doit être exécutée la plus proche possible du traitement des données issues du curseur

Syntaxe :

```
OPEN [GLOBAL] nomcurseur
```

Gestion des curseurs
FETCH

43

- Instruction qui permet d'extraire une ligne du curseur et d'affecter des variables avec leur contenu
- Après FETCH, la variable **@@FETCH_STATUS** est à 0 si tout s'est bien passé

Syntaxe :

```
FETCH[{NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n]
     [FROM] [GLOBAL] nomcurseur[INTO Listevariable]
```

- NEXT : lit la ligne suivante (seule option possible pour les INSENSITIVE CURSOR)
- PRIOR : lit la ligne précédente
- FIRST : lit la première ligne
- LAST : lit la dernière ligne
- ABSOLUTE n : lit la nième ligne de l'ensemble
- RELATIVE n : lit la nième ligne à partir de la ligne courante

Gestion des curseurs
CLOSE et DEALLOCATE

44

- **CLOSE** : fermeture du curseur et libération de la mémoire
- Cette opération doit intervenir dès que possible afin de libérer les ressources le plus tôt possible.

Syntaxe :

```
CLOSE nomcurseur
```

- **DEALLOCATE** : suppression du curseur et des structures associées

Syntaxe :

```
DEALLOCATE nomcurseur
```

Gestion des curseurs

45

- Exemple : affiche la liste des clients sous la forme "numero_cli : prenom nom"

```
DECLARE @NumeroClient char(4);
DECLARE @PrenomClient varchar(40);
DECLARE @NomClient varchar(40);
DECLARE @Message varchar(100);
DECLARE Clients_Cursor CURSOR FOR
SELECT NCli, ISNULL(Prenom,' ? '), Nom FROM client;
OPEN Clients_Cursor;
FETCH NEXT FROM Clients_Cursor
      INTO @NumeroClient, @PrenomClient, @NomClient;
...
```

Gestion des curseurs

46

- Exemple : affiche la liste des clients sous la forme "numero_cli : prenom nom"

```
...
   WHILE @@FETCH_STATUS = 0
   BEGIN
      SET @Message = CAST(@NumeroClient AS varchar(30))
                          + ' : ' + @PrenomClient
                          + ' ' + @NomClient;
      PRINT @Message;
      FETCH NEXT FROM Clients_Cursor
          INTO @NumeroClient, @PrenomClient, @NomClient;
   END;
CLOSE Clients_Cursor;
DEALLOCATE Clients_Cursor;
```

Gestion des exceptions
Les messages d'erreur

47

- Pour chaque erreur, SQL Server produit un message d'erreur
- Par défaut, ce message est affiché à l'écran et sa lecture complète permet bien souvent de résoudre le problème
- La plupart de ces messages sont définis dans SQL Server, mais il est également possible de définir ses propres messages par l'intermédiaire de la procédure **sp_addmessage**

Gestion des exceptions
Structure des messages d'erreur

48

- Quelle que soit leur origine, tous les messages d'erreur possèdent la même structure et les mêmes champs d'informations qui sont :
  - numéro : chaque message est identifié de façon unique par un numéro
  - message au format texte : le message est spécifique à chaque message d'erreurs
  - sévérité : c'est un indicateur sur la gravité de l'erreur
  - état associé au contexte de l'erreur
  - nom de la procédure ayant éventuellement provoquée l'erreur
  - numéro de la ligne à l'origine de l'erreur

Gestion des exceptions
Gravité des messages d'erreur

49

- Chaque message d'erreur possède une gravité également nommée sévérité
- Cette gravité permet de classer les messages par rapport au risque potentiel associé
- Il existe 25 niveaux différents de gravité croissante représentés par un nombre entier compris entre 0 et 24

Gestion des exceptions
Gestion des erreurs

50

- Deux moyens de gérer les erreurs :
  - tester la valeur de la variable système `@@error` après chaque instruction
  - regrouper une ou plusieurs instructions dans un bloc **`TRY`** et de centraliser la gestion des erreurs dans un bloc **`CATCH`**
- Dans le bloc `CATCH`, il est possible d'utiliser les fonctions `ERROR_MESSAGE()`, `ERROR_NUMBER()`, `ERROR_SEVERITY()` et `ERROR_STATE()` pour obtenir des informations sur le message d'erreur, le numéro de l'erreur, la gravité de l'erreur et l'état de l'erreur
- Il n'est possible de gérer dans le bloc `CATCH` que des erreurs dont la sévérité est supérieure à 10 et à condition que l'erreur ne mette pas fin au script

Gestion des exceptions
Gestion des erreurs

51

- Syntaxe :

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

Gestion des exceptions

52

- Exemple :

```
BEGIN TRY
    -- Génère une violation de contrainte
    DELETE FROM produit WHERE NPro = 'CLE22';
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_LINE() AS ErrorLine
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

Les déclencheurs

53

- Mécanismes constitués d'une section de code accompagnée des conditions qui entraînent son exécution
- Un déclencheur (trigger) peut être vu comme un sous-programme associé à un événement particulier (mise à jour de données, instruction SQL, connexion d'un utilisateur, ...)
- Contrairement à une procédure, l'exécution d'un déclencheur n'est pas explicite : c'est l'événement qui exécute automatiquement le code présent dans le déclencheur
  = le déclencheur "se déclenche" (fired trigger)
- Un déclencheur est un objet de la base (créé avec CREATE et supprimé avec DROP)

Les déclencheurs

54

Les événements régis par déclencheurs sont généralement :

- INSERT, UPDATE ou DELETE sur une table
  = déclencheur DML
  Un déclencheur est associé à une seule table qui peut en héberger plusieurs
- CREATE, ALTER, DROP ou d'autres commandes sur un objet (table, index, ...)
  = déclencheur DDL
  L'objectif de ces déclencheurs est de suivre l'évolution de la base pour réaliser différentes tâches d'administration
- La connexion d'un utilisateur (phase de logon qui précède l'établissement d'une session)
  = déclencheur de connexion

Les déclencheurs

55

On distingue 3 types de déclencheurs :

- Les déclencheurs de validation
  Ils renforcent les contraintes par des règles métiers complexes nécessitant généralement une lecture de données externe à la table visée
- Les déclencheurs de correction
  Ils gèrent la qualité des données
  Par exemple : ils corrigent des erreurs de saisie ou réalise un formatage des données
- Les déclencheurs administratifs
  Ils permettent par exemple, de pister les actions de mise à jour effectuées par les utilisateurs ou de gérer des aspects sécuritaires d'accès aux données

Les déclencheurs

56

- L'instruction de création d'un déclencheur comporte deux parties : la description de l'événement et les actions à réaliser lorsque l'événement se produit
- Syntaxe d'un déclencheur DML :

```
CREATE TRIGGER nom_trigger ON { table | vue }
 [ WITH ENCRYPTION ]
 {FOR | AFTER | INSTEAD OF } {  INSERT , UPDATE ,DELETE}
  [ NOT FOR REPLICATION ]
  AS
  [ IF UPDATE ( colonne )
  | IF ( COLUMNS_UPDATED ( )opérateur_comparaison_bits)]
 Instructions_SQL;
```

Les déclencheurs

57

Syntaxe d'un déclencheur DML :

- `WITH ENCRYPTION` : la définition du déclencheur est enregistrée de façon cryptée. Il n'est donc pas possible de connaître le code du déclencheur a posteriori
- `FOR` : permet de préciser à quel ordre SQL DML le déclencheur est associé
- `AFTER` : C'est le mode par défaut des déclencheurs. Le code est exécuté après vérification des contraintes d'intégrité et après modification des données
- `INSTEAD OF` : le corps du déclencheur est exécuté à la place de l'ordre SQL envoyé sur la table ou la vue. Ce type de déclencheur est particulièrement bien adapté pour les vues
- `INSERT, UPDATE, DELETE` : un déclencheur peut agir par rapport à une ou plusieurs actions (séparées par des virgules)

Les déclencheurs

58

Syntaxe d'un déclencheur DML :

- `NOT FOR REPLICATION` : indique que le déclencheur ne doit pas être exécuté lorsque la modification des données est issue d'un processus de réplication
- `IF UPDATE (colonne)` : ne peut être utilisé que pour les déclencheurs UPDATE ou INSERT et ne s'exécutera que si la ou les colonnes sont concernées
- `IF(COLUMNS_UPDATED()opérateur_comparaison_bits)` cette fonction permet de connaître les indices de la ou des colonnes qui ont été mises à jour
  Pour chaque colonne affectée par la mise à jour, un bit est levé
  Pour savoir quelles ont été les colonnes mises à jour, une comparaison binaire suffit
- `Instructions_SQL` : il est possible d'utiliser toutes les instructions de manipulation de données (DML)

Les déclencheurs

59

- Il est possible de définir sur une même table plusieurs déclencheurs pour chaque opération INSERT, UPDATE et DELETE
- Si l'instruction SQL échoue, alors le déclencheur n'est pas exécuté
- Pseudo-tables DELETED et INSERTED
  - Pour un événement déclencheur AFTER DELETE,
    chaque ligne supprimée est accessible au niveau d'un déclencheur par la pseudo-table DELETED
  - Pour un événement déclencheur AFTER INSERT,
    chaque ligne insérée est accessible au niveau d'un déclencheur par la pseudo-table INSERTED
  - Pour un événement déclencheur AFTER UPDATE,
    les pseudo-tables INSERTED et DELETED peuvent être manipulées

Les déclencheurs

60

- Exemple : Déclencheur d'insertion de la date de la dernière facture saisie

```
CREATE TRIGGER ins_fac_date ON facture
AFTER INSERT AS
BEGIN
  DECLARE @numero AS int
  DECLARE fInserted CURSOR FOR SELECT NFac FROM inserted;
  OPEN fInserted;
  FETCH fInserted INTO @numero;
  WHILE (@@FETCH_STATUS=0) BEGIN
    UPDATE facture SET DateFac=GETDATE()WHERE NFac=@numero;
    FETCH fInserted INTO @numero;
  END;
  CLOSE fInserted;
  DEALLOCATE fInserted;
END;
```