

Laboratoires de bases de données

Laboratoire n°6

Programmation SQL

par Danièle BAYERS et Louis SWINNEN
Révision 2018 : Vincent Reip

Ce document est disponible sous licence Creative Commons indiquant qu'il peut être reproduit, distribué et communiqué pour autant que le nom des auteurs reste présent, qu'aucune utilisation commerciale ne soit faite à partir de celui-ci et que le document ne soit ni modifié, ni transformé, ni adapté.



<http://creativecommons.org/licenses/by-nc-nd/2.0/be/>

La Haute Ecole Libre Mosane (HELMo) attache une grande importance au respect des droits d'auteur. C'est la raison pour laquelle nous invitons les auteurs dont une œuvre aurait été, malgré tous nos efforts, reproduite sans autorisation suffisante, à contacter immédiatement le service juridique de la Haute Ecole afin de pouvoir régulariser la situation au mieux.

Février 2018

La programmation SQL

1. Introduction

Les bases de données comme SQL Server et Oracle permettent, en plus de stocker des données, d'y ajouter des procédures. Ces procédures sont programmées dans un dialecte SQL qui est propre au SGBD ainsi, en Oracle, la programmation se fait en PL/SQL tandis que sous SQL Server, la programmation se fait en Transact-SQL (T-SQL).

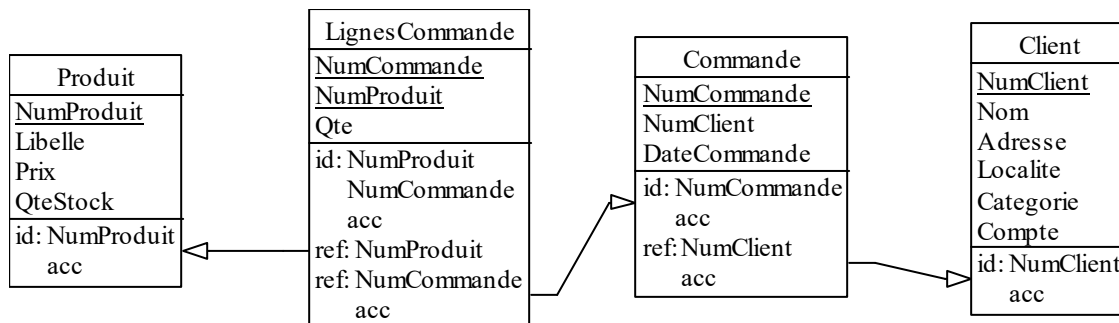
Dans la suite de ce document, nous détaillerons la programmation SQL en Transact-SQL (dialecte utilisé sous SQL Server) et, en annexe, vous trouverez l'équivalent en PL/SQL (sous Oracle).

On distingue généralement *les procédures stockées* et *les déclencheurs*. Les *procédures stockées* (ou *stored procedures*) sont des procédures attachées à une base de données. L'intérêt principal étant de concentrer les traitements centrés sur les données au niveau du SGBD. Ainsi, lorsque des mises à jour importantes doivent être effectuées sur les données, il est souvent plus aisé de réaliser les modifications par procédure stockée que via une programmation extérieure dans un langage de programmation classique.

Les *déclencheurs* (ou *triggers*) sont des procédures particulières qui se déclenchent automatiquement lorsqu'un événement précis survient, comme l'insertion d'un enregistrement dans une table. Les déclencheurs sont donc attachés à une table particulière et liés à un événement.

2. La base de données

Dans les exemples, nous supposons que nous disposons de la base de données suivante :



3. Programmation SQL

Suivant le SGBD utilisé, il est tout à fait possible de programmer en SQL en utilisant les structures habituelles comme la sélection, la répétition ou encore l'affectation. Comme expliqué ci-dessus, la programmation SQL s'utilise soit dans une procédure stockée, soit dans un déclencheur.

3.1 Déclaration de variable

Comme dans de nombreux langages, il est nécessaire de déclarer une variable avant de l'utiliser. Les déclarations de variables seront généralement regroupées en début de programme. Une déclaration définit le nom de la variable (unique et préfixé par '@') ainsi que son type (qui doit être un des types supportés par le SGBD) :

```
DECLARE @Var TYPE
```

Exemples :

```
DECLARE @Nom VARCHAR(50)
DECLARE @Aujourd'hui SMALLDATETIME
DECLARE @Prix NUMERIC(6,2)
DECLARE @i INT
```

3.2 Affectation

Durant l'exécution du programme, une valeur peut être affectée à une variable. Cette valeur peut être littérale, provenir de l'exécution d'une fonction, d'une expression arithmétique ou encore d'une requête de type SELECT. L'affectation sera précédée du mot-clé 'SET'.

```
SET @Var = expression
```

Exemples :

```
SET @Nom = 'John Doe'
SET @Aujourd'hui = GETDATE()
SET @Prix = (SELECT prix FROM Produit WHERE NumProduit = 5)
SET @SalaireAnnuel = 62500.00
SET @SalaireMensuel = @SalaireAnnuel/12
```

Il est aussi possible d'effectuer une affectation multiple à partir d'une requête de type 'SELECT'

```
SELECT @Var1 = att1, @Var2 = att2 [, ...]
FROM Table
WHERE ...
```

Exemple :

```
SELECT @Prix = Prix, @Libelle = Libelle, @Qte = QteStock
FROM Produit
WHERE NumProduit = 5
```

3.3 Structure conditionnelle

Il est souvent nécessaire dans un programme d'effectuer des branchements conditionnels. Ceux-ci se font grâce à une structure classique IF_ELSE.

```
IF condition
BEGIN
.
END
ELSE
BEGIN
.
END
```

Exemple :

```
IF (@SalaireAnnuel > 60000)
BEGIN
    SET @SalaireNet = @SalaireAnnuel * 0.5
END
ELSE
BEGIN
    SET @SalaireNet = @SalaireAnnuel * 0.6
END
```

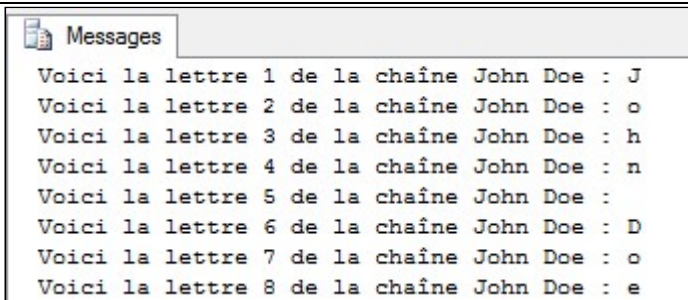
3.4 Structure de répétition

Les boucles disponibles en Transact-SQL sont de type WHILE :

```
WHILE condition
BEGIN
.
END
```

Exemple :

```
SET @i = 0
WHILE (@i <= LEN(@Nom))
BEGIN
    PRINT 'Lettre ' + CAST(@i AS VARCHAR(2)) + ' de la chaîne ' + @nom + ' : ' + SUBSTRING(@nom, @i, 1)
    SET @i = @i + 1
END
```



Messages	
Voici la lettre 1 de la chaîne John Doe :	J
Voici la lettre 2 de la chaîne John Doe :	o
Voici la lettre 3 de la chaîne John Doe :	h
Voici la lettre 4 de la chaîne John Doe :	n
Voici la lettre 5 de la chaîne John Doe :	
Voici la lettre 6 de la chaîne John Doe :	D
Voici la lettre 7 de la chaîne John Doe :	e
Voici la lettre 8 de la chaîne John Doe :	

3.5 Affichage

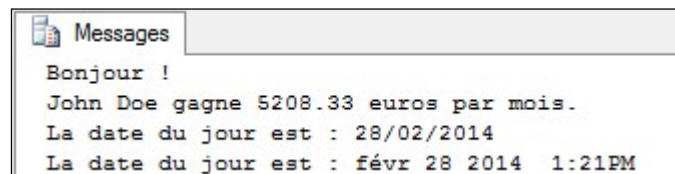
Il est parfois nécessaire d'afficher l'information (généralement pour déboguer un programme). Transact-SQL offre l'instruction 'PRINT' qui renvoie un message vers le 'client'. SQLServer Management Studio qui affichera le message dans un onglet spécifique :

```
PRINT 'information'
```

Exemple :

```
PRINT 'Bonjour !'
PRINT @Nom + ' gagne ' + CAST(@SalaireMensuel AS VARCHAR(9)) + '
euros par mois.'

PRINT 'La date du jour est : ' + CONVERT(CHAR(10), @Aujourd'hui, 103)
PRINT 'La date du jour est : ' + CAST(@Aujourd'hui AS VARCHAR(20))
```



4. Les curseurs

En programmation SQL (comme dans les déclencheurs et les procédures stockées), lorsqu'une requête retourne plusieurs lignes de résultat, il est possible de traiter chaque ligne au moyen d'un curseur.

Il s'agit donc d'un mécanisme qui va permettre de parcourir chaque enregistrement dans le résultat pour effectuer une opération précise.

L'implémentation des curseurs dépend fortement du SGBD employé. Ils sont disponibles sous Oracle et SQL Server.

4.1 Les curseurs sous SQL Server

Format :

```
DECLARE nom_curseur CURSOR
FOR
    SELECT attr1, ..., attrn
    FROM table
    [JOIN ...]
    [WHERE condition]
    [GROUP BY ...]
    [HAVING ...]
    [ORDER BY ...]
```

Le curseur est déclaré comme toutes variables avec le mot clé `DECLARE`. Le type de cette variable est `CURSOR` et ensuite, on trouve la requête. La requête peut contenir toutes les options que nous avons déjà vues.

Une fois le curseur déclaré, il est possible de l'utiliser. On distingue les étapes suivantes :

1. Ouverture du curseur au moyen de la commande SQL `OPEN` :

```
OPEN nom_curseur
```

2. Comme dans la lecture d'un fichier, on commence par lire le 1^{er} résultat :

```
FETCH nom_curseur INTO @Var1, ..., @Varn
```

Il faut remarquer que tous les attributs présents dans la requêtes sont affectés à une variable via INTO.

3. Ensuite, la boucle de parcours est écrite et les instructions traitant le résultat sont contenues à l'intérieur.

```
WHILE @@fetch_status = 0  
BEGIN
```

Traitement du résultat

Note : @@fetch_status est une variable « système » qui permet d'obtenir l'état de la dernière instruction FETCH effectuée (0 = succès, valeur négative = échec).

4. Avant la fin de la boucle, il faut procéder à la lecture du résultat suivant :

```
FETCH nom_curseur INTO @Var1, ..., @Varn  
END
```

5. Une fois la lecture terminée, on ferme le curseur

```
CLOSE nom_curseur
```

6. Si le curseur n'est plus utile, il convient de libérer la ressource :

```
DEALLOCATE nom_curseur
```

Exemple :

```
DECLARE crsrClientA_F CURSOR  
FOR SELECT nom, adresse  
FROM Client  
WHERE UPPER(SUBSTRING(nom,1,1)) IN ('A','B','C','D','E','F')  
  
OPEN crsrClientA_F  
FETCH crsrClientA_F INTO @nom, @adresse  
WHILE @@FETCH_STATUS = 0  
BEGIN  
    PRINT @nom + ' habite à ' + @adresse  
    FETCH crsrClientA_F INTO @nom, @adresse  
END  
CLOSE crsrClientA_F  
DEALLOCATE crsrClientA_F
```

5. Les procédures stockées

Une procédure stockée s'exécute uniquement lorsqu'elle est appelée. Comme elle n'est pas standardisée, sa définition dépend du SGBD employé. Comme toutes les procédures programmées, la procédure stockée peut admettre des paramètres.

Les procédures stockées utilisent des structures classiques comme des boucles ou des sélections. Une force de la procédure stockée est d'autoriser des requêtes directement dans le code et de pouvoir traiter le résultat de ces requêtes à l'aide des structures classiques de programmation.

5.1 Procédure stockée en SQL Server

Format :

```
CREATE PROCEDURE nom_procedure
    [@arg1 type [OUTPUT], ..., @argn type [OUTPUT] ]
AS
    DECLARE @Var1 type
    .
    DECLARE @Varn type
    .
    .
    .
```

Dans le format précédent, on remarque que la procédure `nom_procedure` est créée au moyen d'une commande `CREATE PROCEDURE`. La suppression d'une procédure stockée existante se fait au moyen de `DROP PROCEDURE`.

On trouve ensuite les arguments éventuels. Pour chaque argument, il faut mentionner son **nom**, son **type** et s'il s'agit d'un **paramètre en entrée** (absence de `OUTPUT`) ou **en sortie** (`OUTPUT` – initialisé par la procédure). Par défaut, les paramètres sont considérés comme des paramètres d'entrée. En Transact-SQL, le type de l'argument doit être précisé complètement. Ainsi, il faut mentionner la taille maximale également comme `VARCHAR(30)`.

Particularité : il est possible de mentionner une valeur par défaut pour un paramètre en mentionnant, à la suite de la définition de type, le symbole '=' et la valeur par défaut.

Exemple :

```
CREATE PROCEDURE sp_inc_stock (
    @num_produit CHAR(5),
    @qte_act INTEGER OUTPUT)
AS
    DECLARE @Qte INTEGER

    SET @Qte = QteStock
    FROM Produit
    WHERE NumProduit = @num_produit

    SET @Qte = @Qte + 1

    UPDATE Produit
    SET QteStock = @Qte
    WHERE NumProduit = num_produit
```

```

SET @qte_act = @Qte

-- exécution :
DECLARE @RESULT DECIMAL(2,1)
EXEC SP_INC_STOCK 'P001', @RESULT OUTPUT
GO

```

Dans cet exemple, la procédure `sp_inc_stock` permet d'incrémenter le stock pour le produit `num_produit` d'une unité.

6. Les déclencheurs

Pour gérer des contraintes complexes au niveau du SGBD, il est souvent nécessaire de recourir aux déclencheurs (ou *triggers*). Un déclencheur est une procédure programmée au niveau du SGBD qui s'exécute **automatiquement** lorsqu'un événement précis survient.

6.1 Les événements

Les événements *déclencheurs* sont l'ajout, la suppression ou la mise à jour d'un enregistrement dans la table ou même d'une colonne particulière. Les déclencheurs ne sont pas normalisés dans la norme SQL-2, il convient dès lors de se référer au dialecte de son SGBD (PL/SQL pour Oracle ou Transact-SQL pour SQL Server). Les événements déclencheurs sont généralement :

- BEFORE INSERT : *exécution du code avant une insertion*
- BEFORE UPDATE : *exécution du code avant une modification*
- BEFORE DELETE : *exécution du code avant une suppression*
- AFTER INSERT : *exécution du code après une insertion*
- AFTER UPDATE : *exécution du code après une mise à jour*
- AFTER DELETE : *exécution du code après une suppression*

Sous SQL Server, les triggers BEFORE ne sont pas disponibles.

Les événements *BEFORE* sont souvent utilisés pour vérifier une contrainte particulière et éventuellement **arrêter l'insertion, la mise à jour ou la suppression** si la contrainte n'est pas respectée.

Les événements *AFTER* sont souvent utilisés pour mettre à jour des données en fonction de la demande : mettre à jour un solde, un stock, ... suite à l'ajout, la modification ou la suppression d'un enregistrement, par exemple.

Les déclencheurs sont également très souvent utilisés pour maintenir une certaine « dénormalisation » du schéma de la base de données. Ainsi, les *attributs dérivables* (i.e. résultat d'une opération entre des informations présentes dans la base de données comme le total d'une commande par exemple) peuvent, pour des questions de performances, être présents dans le schéma du SGBD. Afin d'assurer la cohérence des informations et être sûr que ces attributs dérivables sont toujours à jour, les déclencheurs sont alors très souvent utilisés.

Finalement, il est possible de **restreindre l'exécution** du déclencheur à l'ajout, la modification, ou la suppression d'**un champ particulier** dans la table.

6.2 Les déclencheurs sous SQL Server

Format :

```
CREATE TRIGGER nom_trigger ON nom_table
  (FOR|AFTER|INSTEAD OF) (INSERT [,] | UPDATE [,] | DELETE [,])+
AS
  [IF [NOT] UPDATE (attrx)
    BEGIN
    END]
  [DECLARE @Var1 type, ..., @Varn type]
  .
  .
  .
```

La déclaration d'un déclencheur commence par les mots CREATE TRIGGER. Il faut ensuite spécifier le nom du déclencheur (`nom_trigger`) et la table sur laquelle il porte (`nom_table`). Il faut ensuite préciser l'événement déclencheur. Enfin, le code du déclencheur commence. Si on souhaite limiter un déclencheur à un champ particulier, il est nécessaire de spécifier l'option IF [NOT] UPDATE et l'attribut concerné. Cette option **n'est possible que pour des déclencheurs INSERT et UPDATE**. Exemple de syntaxe :

```
IF NOT UPDATE (attr)
  RETURN
```

On trouve ensuite la déclaration des variables et le code du déclencheur lui-même.

6.2.1 Événements déclencheurs

SQL Server permet les options suivantes :

- **FOR ou FOR AFTER ou AFTER** : Ces déclencheurs s'exécutent après que l'instruction SQL ait été complètement exécutée. Le déclencheur ne sera exécuté que si l'instruction SQL s'est passée correctement. En particulier, si toutes les contraintes programmées (contraintes d'intégrité, contrainte référentielle, clause CHECK) ont été vérifiées. **FOR**, **FOR AFTER** et **AFTER** sont des synonymes.
- **INSTEAD OF** : Il s'agit d'un type particulier de déclencheur. Ainsi **au lieu d'exécuter la requête SQL qui a déclenché le trigger, seul le code du trigger est exécuté**.

Les événements déclencheurs sont *l'insertion, la mise à jour et la suppression*.

6.2.2 Les types de déclencheur

SQL Serveur définit uniquement des déclencheurs de table. Ainsi, le déclencheur n'est exécuté qu'une seule fois même si la requête SQL qui a provoqué son exécution porte sur plusieurs lignes.

Pour travailler sur chaque ligne pointée par la requête SQL, il est nécessaire de définir un curseur sur les pseudo-tables *inserted* ou *deleted*.

6.2.3 Les anciennes et nouvelles valeurs

SQL Server permet d'avoir accès, lors de l'exécution du déclencheur, aux anciennes et nouvelles valeurs de la table. A cette fin, et suivant l'événement déclencheur, le programmeur peut faire appel aux pseudo-tables *inserted* et *deleted*.

INSERT	<i>inserted</i> accessible
UPDATE	<i>inserted</i> et <i>deleted</i> accessibles
DELETE	<i>deleted</i> accessible

6.2.4 Arrêter l'exécution du déclencheur

On peut lire dans [3] : « *ROLLBACK TRANSACTION behaves differently when used within a trigger than when not within a trigger. (...) When within a trigger, it is not necessary to have a matching BEGIN TRANSACTION statement because each SQL statement that is not within an explicit transaction is effectively a one-statement transaction. Nothing at the SQL batch or stored procedure can get "inside" such a one-statement transaction ; however, the Transact-SQL statement within a trigger is effectively "inside" the one-statement transaction, and therefore it does make sense to allow an unbalanced rollback to the beginning of the one-statement transaction.* ».

Il est à noter que l'utilisation de l'option `ROLLBACK TRANSACTION` ne stoppera pas l'exécution du trigger [3]. Dans maintes situations, il est donc nécessaire, après le `ROLLBACK`, de mettre fin à l'exécution du trigger grâce à l'instruction `RETURN`. On peut donc arrêter l'exécution du déclencheur **et revenir au point juste avant l'exécution de la requête SQL ayant déclenché le trigger** en utilisant l'option `ROLLBACK TRANSACTION` suivie de l'instruction `RETURN`. Comme nous pouvons définir uniquement des déclencheurs *AFTER*, le `ROLLBACK` aura comme conséquence de « défaire » les modifications effectuées. De cette manière, il est possible de « simuler » un déclencheur de type *BEFORE*.

6.2.5 Les exceptions en SQL Server

Il est parfois intéressant de signaler une erreur. En effet, en levant une erreur, on peut mentionner, par un message, le type d'erreur rencontré. Pour ce faire, il faut utiliser l'instruction Transact-SQL `RAISERROR` dont le format est le suivant :

```
RAISERROR( message, sévérité, état)
```

Le `message` est une chaîne de caractères libre mentionnant le type d'erreur qui est survenu. La `sévérité` est un entier entre 0 et 18 qui représente la gravité de l'erreur.

L'`état` est un entier, compris entre 1 et 127, qui peut être utilisé pour repérer l'endroit où l'erreur s'est produite. Par exemple, si la même erreur peut survenir à plusieurs endroits du déclencheur, l'état permettra de désigner l'instruction `RAISERROR` qui a été exécutée.

6.2.6 Particularités de SQL Server

1. Il n'est pas possible de définir un déclencheur *AFTER* sur une vue. Seul un déclencheur *INSTEAD OF* est autorisé pour autant que l'option `WITH CHECK OPTION` n'ait pas été précisée lors de la création de la vue.
2. Il est interdit de modifier les tables *inserted* ou *deleted*.
3. Il est interdit d'utiliser des commandes comme `CREATE INDEX`, `ALTER INDEX`, `DROP INDEX`, `DROP TABLE` ou `ALTER TABLE` sur la table mentionnée dans le déclencheur. Les autres actions sont permises.

6.2.7 Exemples

Le trigger ci-dessous met à jour les clients de sorte qu'un client dont le compte passe en dessous de -10000 change de catégorie. Si ce client était en catégorie B2, il passe en catégorie B1 tandis que s'il était en catégorie C2, il passe en catégorie C1.

```
CREATE TRIGGER maj_autom_cat_inf_client ON CLIENT
AFTER UPDATE
AS
BEGIN
    IF NOT UPDATE(Compte)
        RETURN

    UPDATE Client
        SET Cat='B1'
        WHERE Compte <-10000
        AND Cat='B2'

    UPDATE Client
        SET Cat='C1'
        WHERE Compte <-10000
        AND Cat='C2'
END
```

Note : ce trigger sera déclenché lors de chaque update d'un tuple dans la table client mais il affectera potentiellement d'autres tuples (qui n'étaient pas concernés par l'événement déclencheur UPDATE).

```

CREATE TRIGGER maj_cat_client ON CLIENT
AFTER UPDATE
AS
BEGIN
    IF NOT UPDATE(Compte)
        RETURN

    DECLARE @NumClient INTEGER
    DECLARE @Cat CHAR(2)
    DECLARE @Cpt NUMERIC(9,2)
    DECLARE @OldCPT NUMERIC(9,2)

    DECLARE curseur CURSOR
    FOR
        SELECT NumClient, Categorie, Compte
        FROM inserted

    OPEN curseur
    FETCH curseur INTO @NumClient, @Cat, @Cpt

    WHILE @@fetch_status = 0
    BEGIN
        SELECT @OldCPT = Compte
        FROM deleted
        WHERE NumClient = @NumClient
        IF @Cat = 'B1'
        BEGIN
            IF @OldCPT > 0 AND @Cpt < 0
            BEGIN
                RAISERROR('Un client B1 ne peut passer en negatif', 7, 1)
                ROLLBACK TRANSACTION
                RETURN
            END
        END
        FETCH curseur INTO @NumClient, @Cat, @Cpt
    END
    CLOSE curseur
    DEALLOCATE curseur
END

```

Ce trigger illustre plusieurs choses : l'utilisation d'un curseur pour parcourir tous les éléments d'une table, l'utilisation des pseudo-tables *deleted* et *inserted* contenant les données qui sont en cours de modification, l'utilisation d'un message d'erreur et l'annulation du traitement en cours grâce aux instructions RAISERROR et ROLLBACK TRANSACTION.

Ce trigger assure, lors d'une mise à jour de la table client, que tous les clients modifiés (i.e. qui se trouvent dans inserted et deleted) dont la catégorie est B1 et ayant un compte positif, ne peuvent passer en négatif.

7. Les transactions

Un concept majeur des bases de données est la notion des transactions. Une transaction est un ensemble de commandes SQL qui respecte les propriétés suivantes (A-C-I-D) :

- **Atomicité** – Les commandes SQL faisant partie de la transaction sont exécutées complètement ou pas du tout
- **Cohérence** – Si l'état de la base de données était cohérent avant l'exécution de la transaction, il le sera après également.
- **Isolation** – Les transactions peuvent s'exécuter de manière concurrente. Le SGBD garantira que l'exécution d'une transaction sera sans effet sur l'exécution des autres transactions
- **Durabilité** – Les changements effectués au terme de la transaction sont permanents.

Grâce à ces principes, nous savons que la transaction est une opération atomique dont les effets seront permanents si celle-ci se déroule correctement. Pour ce faire, le SGBD va exécuter les commandes SQL de la transaction « de manière temporaire » jusqu'à ce qu'une commande **COMMIT** ou **ROLLBACK** soit rencontrée.

La commande **COMMIT** informe le SGBD qu'il peut sauvegarder les modifications effectuées par la transaction de manière durable. Les modifications sont alors permanentes et la transaction est terminée.

La commande **ROLLBACK** informe le SGBD qu'un problème est survenu durant le traitement de la transaction et **qu'il faut défaire les commandes SQL pour revenir au point précédent l'exécution de la transaction.**

Les transactions sont très largement utilisées dans la programmation afin d'obtenir des opérations atomiques. Par exemple : *le virement d'argent d'un compte vers un autre* est une transaction car :

- Soit l'échange se passe bien et le premier compte est débité d'une somme tandis que le second compte est crédité de cette même somme ;
- Soit l'échange ne se passe pas bien et aucun compte n'est crédité, ni débité.

Ainsi les transactions sont très utiles lorsqu'il faut réaliser **plusieurs opérations** en une seule fois. Il est bien évident qu'une transaction contenant une seule requête n'a aucun intérêt.

Attention ! Une transaction se doit d'être toujours la plus petite possible et être automatique. En effet, afin de gérer les transactions, le SGBD pose automatiquement des verrous sur des tables. Si la transaction est longue, les performances du SGBD peuvent se dégrader. Si la transaction n'est pas automatique (i.e. une fois lancée, elle attend des données de l'utilisateur par exemple), son temps d'exécution peut ici aussi être très important et donc dégrader fortement les performances du SGBD.

Les transactions sont surtout utilisées du côté applicatif (par exemple en Java, lors de la conception de couche d'accès BD) pour assurer la cohérence des mises à jour..

7.1 Transactions sous SQL Server

En SQL Server, chaque instruction SQL est également une transaction. Si l'utilisateur souhaite définir ses transactions, il doit respecter le schéma suivant :

```
BEGIN TRANSACTION nom ;  
    Instructions de la transaction  
(COMMIT|ROLLBACK) TRANSACTION nom ;
```

Le nom est précisé afin de pouvoir faire face à des transactions imbriquées. Toutes les informations concernant les transactions sont disponibles dans la documentation en ligne installée sur chaque machine.

Bibliographie

- [1] C. MAREE et G. LEDANT, *SQL-2 : Initiation, Programmation*, 2^{ème} édition, Armand Colin, 1994, Paris
- [2] P. DELMAL, *SQL2 – SQL3 : application à Oracle*, 3^{ème} édition, De Boeck Université, 2001, Bruxelles
- [3] Microsoft, MSDN Microsoft Developer Network, <http://msdn.microsoft.com>, consulté en janvier 2009 et en février 2013, Microsoft Corp.
- [4] Diana Lorentz, et al., Oracle Database SQL Reference, 10g Release 2 (10.2), published by Oracle and available at <http://www.oracle.com/pls/db102/homepage>, 2005
- [5] Frédéric Brouard, Petit guide de Transact SQL, <http://sqlpro.developpez.com/cours/sqlserver/transactsql>, publié sur developpez.com, 2004
- [6] JL. HAINAUT, *Bases de données: concepts, utilisation et développement*, Dunod, 2009, Paris.

Remerciements

Un merci particulier à mes collègues Vincent REIP et Vincent WILMET pour leur relecture attentive et leurs propositions de correction et d'amélioration.

Annexe A : Programmation SQL sous Oracle

A.1 Procédure stockée en PL/SQL (Oracle)

Format :

```
CREATE [OR REPLACE] PROCEDURE nom [(arg1 [IN|OUT|IN OUT] type, ...,
                                     argn [IN|OUT|IN OUT])]
AS
    [Var1 type
      .
      Varn type ]
BEGIN
    .
    .
    .
END ;
```

Dans le format précédent, on remarque que la procédure `nom` est créée au moyen d'une commande `CREATE PROCEDURE`. Lors de la mise au point, il est toujours conseillé d'utiliser la forme `CREATE OR REPLACE PROCEDURE` afin de mettre à jour une procédure existante.

On trouve ensuite les arguments éventuels. Pour chaque argument, il faut mentionner son **nom**, s'il s'agit d'un **paramètre en entrée** (`IN` – reçu lors de l'appel), **en sortie** (`OUT` – initialisé par la procédure) ou **en mise à jour** (`IN OUT` – mis à jour par la procédure) et puis **son type**. Par défaut, les paramètres sont considérés comme des paramètres d'entrée `IN`. En ce qui concerne le type, il s'agit des types classiques Oracle sans indication de taille. Ainsi, on peut mentionner comme type `VARCHAR`.

Particularité : il est possible de mentionner une valeur par défaut pour un paramètre en mentionnant le mot clé `DEFAULT` et la valeur par défaut pour ce paramètre à la suite de la définition de type.

Exemple :

```
CREATE OR REPLACE PROCEDURE sp_inc_stock (num_produit IN CHAR,
                                           qte_act OUT INTEGER)
AS
    Qte INTEGER ;
BEGIN
    SELECT QteStock INTO Qte
    FROM Produit
    WHERE NumProduit = num_produit ;
    Qte := Qte + 1 ;

    UPDATE Produit
    SET QteStock = Qte
    WHERE NumProduit = num_produit ;
    qte_act := Qte ;
END ;
```

Dans cet exemple, la procédure `sp_inc_stock` permet d'incrémenter le stock pour le produit `num_produit` d'une unité.

A.2 Instructions PL/SQL (Oracle)

- Affectation :
`Var := expression ;`
- Sélection :
`IF (condition) THEN`
.
`ELSE`
.
`END IF;`
- Boucle :
`LOOP`
.
`EXIT;`
`END LOOP ;`
- Affichage à l'écran :
`DBMS_OUTPUT.PUT ('ma variable = ' || var) ;`

A.3 Les curseurs en PL/SQL (Oracle)

Format :

```
DECLARE
    CURSOR nom_curseur IS
        SELECT attr1, ..., attrn
        FROM table
        [JOIN ...]
        [ WHERE condition]
        [GROUP BY ...]
        [HAVING ...]
        [ORDER BY ...] ;
```

Le curseur est déclaré dans la section `DECLARE` avec toutes les autres variables. La requête SQL peut contenir toutes les options que nous avons déjà vues.

Une fois le curseur déclaré, il est possible de l'utiliser. On distingue les étapes suivantes :

1. Ouverture du curseur au moyen de la commande SQL `OPEN` :

```
OPEN nom_curseur ;
```

2. Boucle de parcours et condition de sortie :

```
LOOP
    FETCH nom_curseur INTO var1, ..., varn ;
    EXIT WHEN nom_curseur%NOTFOUND ;
    Traitement
END LOOP ;
```

L'instruction `FETCH` lit le premier résultat et place l'ensemble des attributs de la requête dans les variables mentionnées après `INTO`.

3. Fermeture du curseur :


```
CLOSE nom_curseur ;
```

A.4 Déclencheurs en PL/SQL (Oracle)

Format :

```
CREATE [OR REPLACE] TRIGGER nom_trigger
  (BEFORE|AFTER) (INSERT|UPDATE|DELETE) ON nom_table
  [FOR EACH ROW]
  [WHEN condition]
  [DECLARE
    Var1 type ;
    .
    Varn type ; ]
BEGIN
  .
  .
  .
END ;
```

Le format proposé montre comment un déclencheur peut être défini. Ainsi, on débute par `CREATE OR REPLACE TRIGGER` suivi du nom de ce déclencheur (l'option `REPLACE` permet de modifier un déclencheur existant). Le nom doit être unique. Ensuite on précise l'événement déclencheur et on indique la table sur laquelle il porte.

Le type de déclencheur (ligne ou table) est précisé par la présence de `FOR EACH ROW` (trigger ligne). Si cette option n'est pas précisée, il s'agit d'un trigger table. Enfin, la clause `WHEN` permet de limiter le déclencheur à une condition ou un champ particulier de la table.

Le code du déclencheur suit cette définition et est séparé en 2 parties : la 1^{ère} partie contient l'ensemble des déclarations introduites par le mot clé `DECLARE`. La 2^{ème} partie débute avec le mot réservé `BEGIN` et le code SQL du déclencheur est alors placé. Le mot réservé `END` termine la définition du déclencheur.

A.4.1 Les types de déclencheurs

Il existe **deux** types de déclencheurs différents : **les déclencheurs de table** (STATEMENT) et **les déclencheurs de ligne** (ROW). Quelle est la différence ?

- Les *déclencheurs de table* sont exécutés **une seule fois** lorsque des modifications surviennent sur une table (même si ces modifications concernent plusieurs lignes de la table). Ils sont utiles si des *opérations de groupe doivent être réalisées* (comme le calcul d'une moyenne, d'une somme totale, d'un compteur, ...). Pour des raisons de performance, il est préférable d'employer ces triggers plutôt que les triggers lignes.
- Les *déclencheurs de lignes* sont exécutés « **séparément** » **pour chaque ligne modifiée** dans la table. Ils sont très utiles s'il faut mesurer une *évolution pour certaines valeurs, effectuer des opérations pour chaque ligne en question*.

A.4.2 Les anciennes et nouvelles valeurs

Oracle définit 2 tables temporaires particulières lors de l'exécution des déclencheurs. Ces deux tables permettent d'avoir accès aux anciennes et/ou aux nouvelles valeurs. **Ces deux tables n'existent que dans le cas de déclencheurs de type ligne.**

Ces deux pseudo-tables sont très importantes car elles permettent d'effectuer des opérations particulières comme vérifier qu'une contrainte est toujours vérifiée ou mettre à jour une donnée particulière. *Par exemple, la mise à jour d'une ligne de commande met à jour le total de la commande.*

Ainsi, Oracle désigne ces tables par *:old* et *:new*. Suivant l'événement déclencheur, ces tables sont accessibles ou non :

INSERT	OLD n'est pas accessible car inexistant
UPDATE	OLD et NEW sont accessibles
DELETE	NEW n'est pas accessible car inexistant

A.4.3 Arrêter l'exécution du déclencheur

Dans certaines situations, par exemple lorsqu'une contrainte n'est pas vérifiée, il est intéressant d'arrêter la modification en cours. Pour ce faire, il est nécessaire **de lever une exception** qui aura pour conséquence l'arrêt de cette modification et le retour à l'état précédent.

Les exceptions sous Oracle

Parmi les limites d'un déclencheur, on peut lire dans [4] : « *DDL statements are not allowed in the body of a trigger. Also, no transaction control statement are allowed on a trigger. ROLLBACK, COMMIT and SAVEPOINT cannot be used. (...)* »

Comme il n'est pas possible de définir des transactions ou de lancer un *ROLLBACK* pour terminer l'exécution normale, d'autres moyens doivent être mis en œuvre.

Ainsi on peut lire dans [4] : « *Oracle Database allow user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle Database.*

User-specified error message are returned using the RAISE_APPLICATION_ERROR procedure. (...) This procedure stops procedure execution, rolls back any effects of the procedure, and returns a user-specified error number and message (unless the error is trapped by an exception handler). The error number must be in the range of -20000 to -20999.

In database PL/SQL program units, an unhandled user-error condition or internal error condition that is not trapped by an appropriate exception handler causes the implicit rollback of the program unit. »

Par conséquent, dans un déclencheur *BEFORE*, vous pouvez empêcher la modification (*INSERT*, *UPDATE*, *DELETE*) en lançant une erreur via la procédure *RAISE_APPLICATION_ERROR*.

Syntaxe à utiliser lorsqu'un traitement d'exception est présent :

```
DECLARE
    mon_exception EXCEPTION;
.
BEGIN
    .
    RAISE mon_exception;
```

```

EXCEPTION
  WHEN mon_exception THEN
    RAISE_APPLICATION_ERROR(-20005, 'Exception');
END;
```

A.4.4 Particularités d'Oracle

1. Un déclencheur ligne ne peut pas accéder à une table mutante. Dans [4], on peut lire : « *A mutating table is a table that is being modified by an UPDATE, DELETE or INSERT statement, or a table that might be updated by the effects of a DELETE CASCADE constraint. (...) The session that issued the triggering statement cannot query or modify a mutating table. This restriction prevents a trigger from being seeing inconsistent set of data. (...)* »

Il faut donc comprendre qu'il **n'est pas permis** de consulter ou modifier une table *mutante*. Cette restriction préservera le déclencheur ligne de lire des données inconsistantes. Donc votre trigger ligne ne devrait jamais accéder à la table sur laquelle il porte autrement que par OLD et NEW.

2. Dans un déclencheur ligne, il est possible de modifier les éléments de NEW. Ainsi, on peut lire dans [4] que : « *Old and new values are available in both BEFORE and AFTER row triggers. A new column value can be assigned in a BEFORE row trigger, but not in an AFTER row trigger (because the triggering statement takes effect before an AFTER row trigger is fired). If a BEFORE row trigger changes the value of new.column, then an AFTER row trigger fired by the same statement sees the change assigned by the BEFORE row trigger.* »

Il est par conséquent autorisé de modifier les valeurs des colonnes au travers de NEW (et donc écrire dans le code PL/SQL :new.val :=2 par exemple), uniquement dans des déclencheurs lignes BEFORE.

3. Si vous spécifiez dans un déclencheur ligne que son déclenchement dépend *d'une colonne précise* dans une table (via la clause WHEN), il serait logique que l'élément OLD et NEW **ne porte que** sur cette colonne. Puisque c'est cet élément précis qui intervient dans votre déclencheur, il serait très étrange d'utiliser OLD ou NEW pour atteindre une autre colonne de la table même si le SGBD autorise cela.

A.4.5 Exemple

```

CREATE OR REPLACE TRIGGER upd_qt_produit
AFTER UPDATE OF QSTOCK ON produit
FOR EACH ROW
  DECLARE
    quantite_insuffisante exception;
    quantite_com number;
    quantite_stock number;
  BEGIN
    SELECT MAX(qcom) INTO quantite_com
    FROM lignecom
    WHERE npro = :new.npro;

    quantite_stock := :new.qstock;
    IF quantite_stock < quantite_com THEN
```

```

        RAISE quantite_insuffisante;
    END IF;
EXCEPTION
    WHEN quantite_insuffisante THEN
        raise_application_error (-20001, 'Quantite de stock
            insuffisante');
END;
```

A.5 Transactions en PL/SQL (Oracle)

Oracle gère les transactions de manière implicite. Ainsi, toute instruction de *manipulation des données* (comme SELECT, UPDATE ou DELETE) démarre une transaction. La transaction se termine automatiquement lorsque :

- COMMIT ou ROLLBACK est rencontré
- Lorsqu'une instruction appartenant au LDD (*langage de définition de données* comme CREATE, DROP, ALTER) ou au LCD (*langage de contrôle de données* gérant l'accès comme GRANT, REVOKE, ...) est exécutée (→ COMMIT)
- Le système tombe en panne (→ ROLLBACK)

Dans l'interface web, les commandes SQL s'exécutent en mode **autocommit**. Cela signifie qu'une instruction *COMMIT* implicite est automatiquement effectuée après chaque instruction SQL de modification des données.

En fait, l'option *Validation automatique* assure ce comportement par défaut. C'est pourquoi il n'a jamais été nécessaire d'exécuter un COMMIT.

A.6 Exercice en PL/SQL (Oracle)

1. (Sous Oracle uniquement) La suppression d'une catégorie principale (i. e. ayant des sous-catégories) est interdite. La suppression d'une catégorie enfant entraîne le changement de catégorie pour tous les livres concernés vers la catégorie parent.

Reprendre les exercices précédents (§ 8) et les réaliser sous Oracle.