INFORMATIQUE

Laboratoires de bases de données

Laboratoire n°6
Programmation SQL

par Danièle BAYERS et Louis SWINNEN
Révisions 2018-2024 : Vincent Reip

Mars 2024

La programmation SQL

1. Introduction

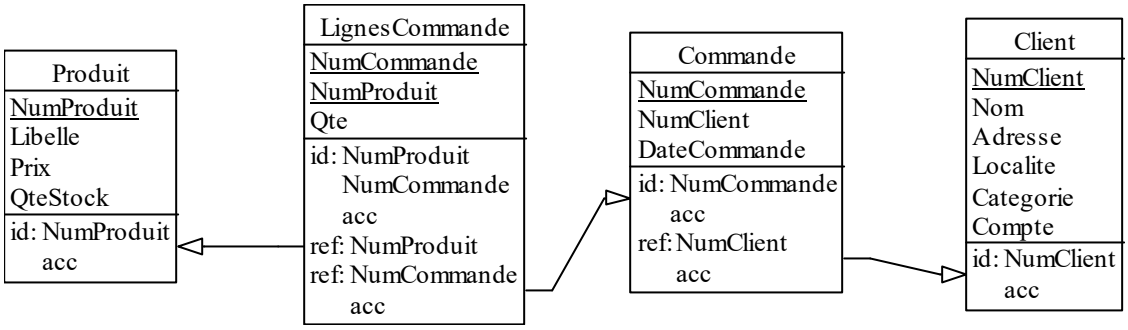
Les bases de données comme SQL Server et Oracle permettent, en plus de stocker des données, d'y ajouter des procédures. Ces procédures sont programmées dans un dialecte SQL qui est propre au SGBD ainsi, en Oracle, la programmation se fait en PL/SQL tandis que sous SQL Server, la programmation se fait 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 A, vous trouverez quelques fonctions de manipulation de dates ou de chaînes de caractères sous SQL Server.

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. Par analogie, on peut considérer que les procédures stockées sont des fonctions qui peuvent être appelées en leur communiquant éventuellement des arguments.

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 attachés à une table particulière et liés à un événement qui peut être l'insertion d'un tuple ou la modification/suppression d'un ensemble (éventuellement vide) de tuples. Lors de leur exécution suite à la survenance d'un des événements susmentionnés, les *déclencheurs* bénéficient d'un *contexte d'exécution* qui permet d'examiner le(s) tuple(s) en cours de traitement ainsi que d'accéder à toutes les tables du schéma.

2. La base de données

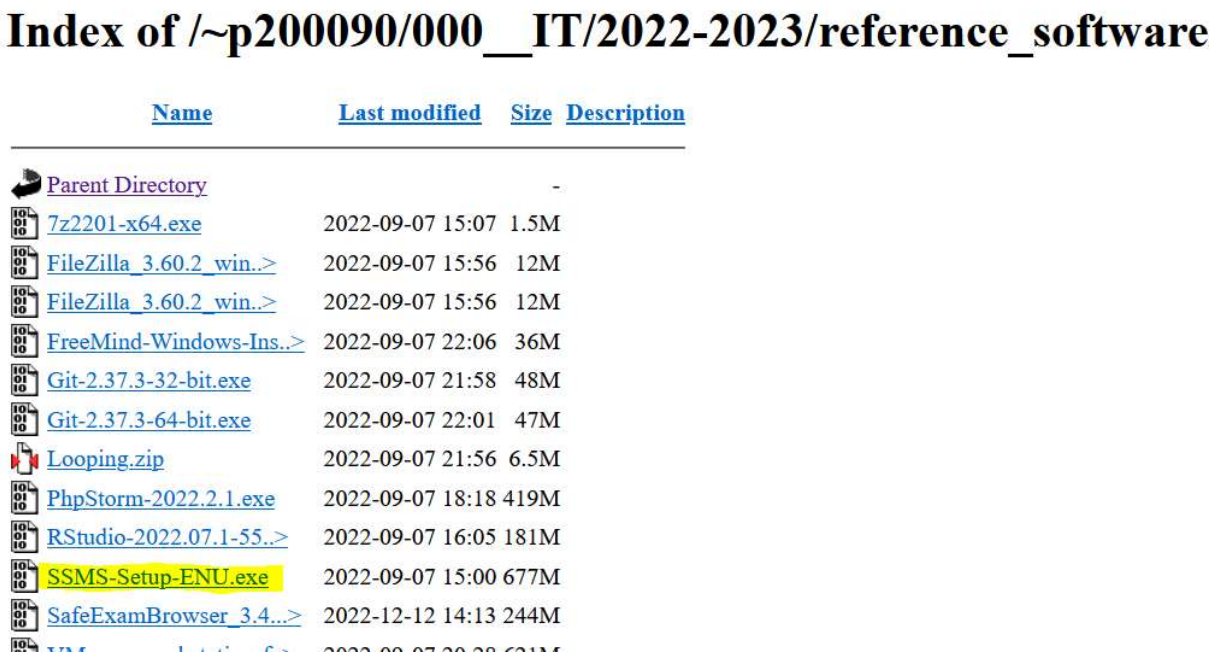
Dans les exemples, nous supposerons que nous disposons de la base de données suivante :



3. Préparation de l'environnement

Pour ce labo, nous allons utiliser un nouvel environnement de travail : Microsoft SQL Server comme SGBD et Microsoft SQL Server Management Studio comme logiciel pour exécuter nos requêtes. Les noms d'utilisateur et mots de passe sont les mêmes que sur Oracle.

Vous pouvez télécharger une version identique à celle qui est installée sur les PCs de labo via le lien *[XX-YY] Liens vers les logiciels de référence* situé dans l'espace de cours Ressources Informatiques. Ce lien est accessible via VPN ou via le réseau HELMo.



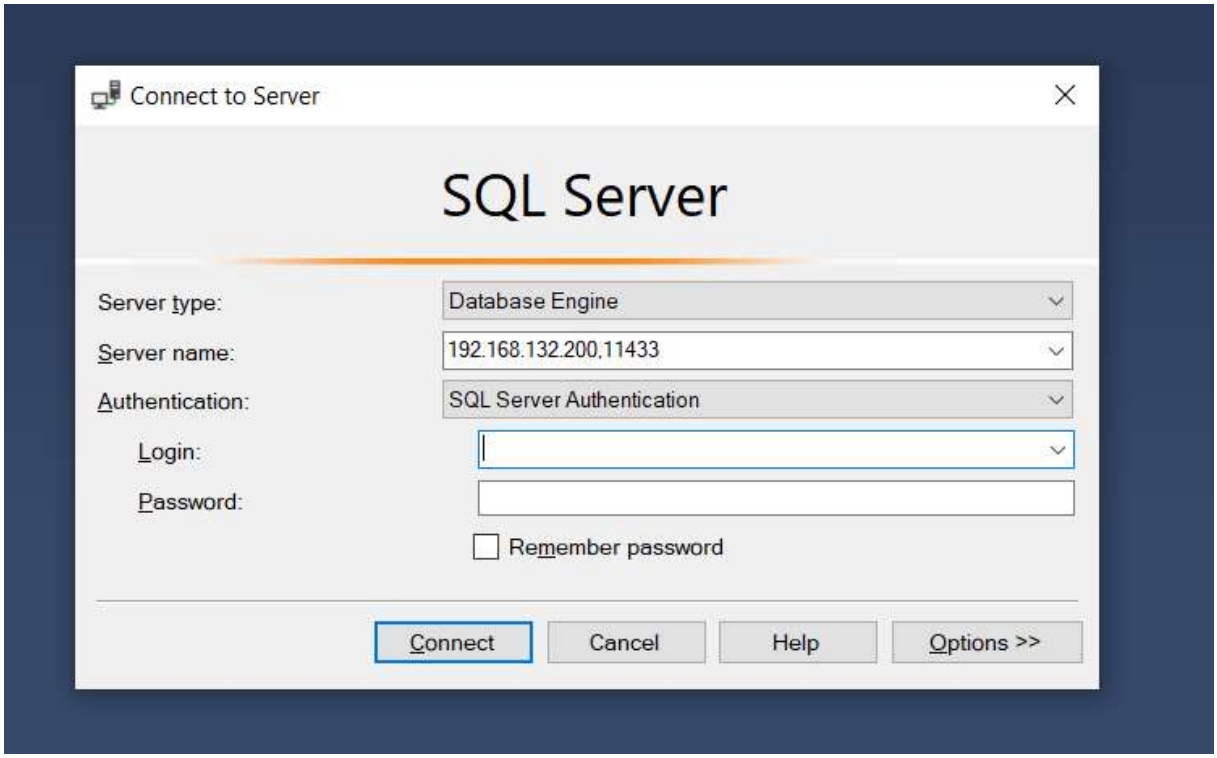
Index of /~p200090/000__IT/2022-2023/reference_software

Name	Last modified	Size	Description
Parent Directory		-	
7z2201-x64.exe	2022-09-07 15:07	1.5M	
FileZilla_3.60.2_win..>	2022-09-07 15:56	12M	
FileZilla_3.60.2_win..>	2022-09-07 15:56	12M	
FreeMind-Windows-Ins..>	2022-09-07 22:06	36M	
Git-2.37.3-32-bit.exe	2022-09-07 21:58	48M	
Git-2.37.3-64-bit.exe	2022-09-07 22:01	47M	
Looping.zip	2022-09-07 21:56	6.5M	
PhpStorm-2022.2.1.exe	2022-09-07 18:18	419M	
RStudio-2022.07.1-55..>	2022-09-07 16:05	181M	
SSMS-Setup-ENU.exe	2022-09-07 15:00	677M	
SafeExamBrowser_3.4...>	2022-12-12 14:13	244M	

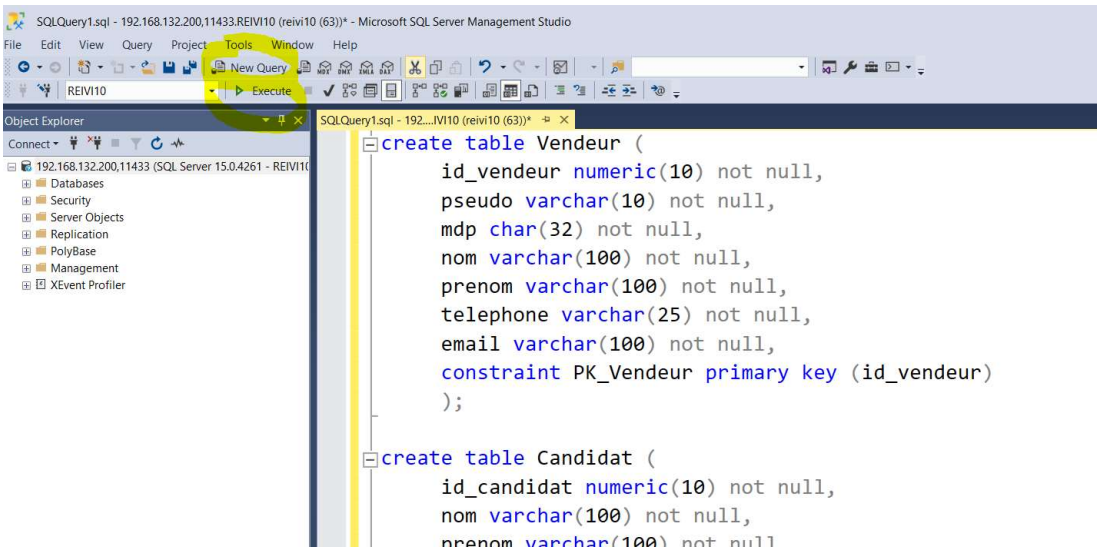
L'utilisation de Datagrip reste possible (notamment sur MacOSX).

Une fois téléchargé, double-cliquez sur l'exécutable et procédez à l'installation (un document explicatif est disponible sur Learn).

Vous pourrez ensuite vous connecter avec votre identifiant BD habituel et les données de connexion (adresse IP : 192.168.132.200 et port 11433 – ces deux données sont séparées par une virgule comme ci-dessous) :



Enfin, vous pourrez exécuter les scripts de création des tables et d'insertion des données dans l'espace Learn. Il vous faudra ouvrir un nouvel onglet avec le bouton "*New Query*", copier/coller le script et l'exécuter avec le bouton "*Execute*" :



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. Pour rappel, nous abordons la programmation SQL dans le contexte du SGBD Microsoft SQL Server.

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 @Aujourdhui 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'affection sera précédée du mot-clé 'SET'.

```
SET @Var = expression
```

Exemples :

```
SET @Nom = 'John Doe'
SET @Aujourdhui = 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
```

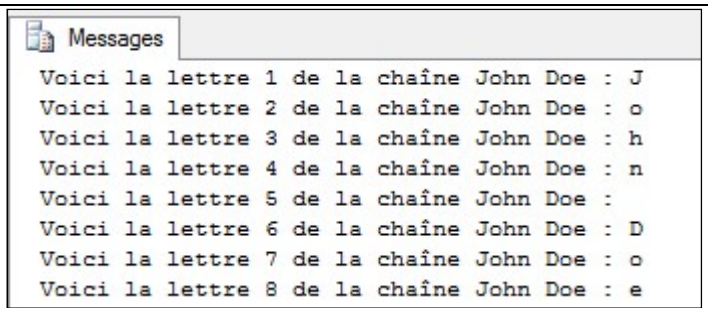
3.4 Structure de répétition

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
```



3.5 Affichage

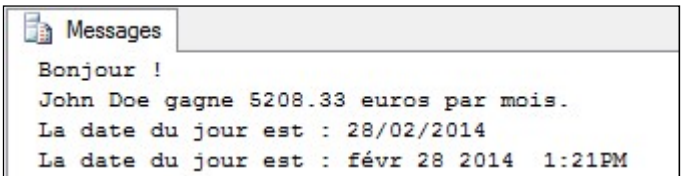
Il est parfois nécessaire d'afficher le l'information (généralement pour débugger un programme). Transact-SQL offre l'instruction 'PRINT' qui renvoie un message vers le 'client'. SQLServer Management Studio qui affichera le message dans un onglet spécifique :

```
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), @Aujourdhui,  103)
PRINT 'La date du jour est : ' + CAST(@Aujourdhui 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 tuple provenant du résultat d'une requête pour effectuer une opération précise avec ce tuple.
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 …]
    [UNION|INTERSECT|EXCEPT]
    […]
```

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 dans les labos précédents.

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 1ier enregistrement (ic, on parle d'un tuple) et on affecte les valeurs de chacun des attributs à des variables (de type adéquat) déclarées au sein du trigger :

```
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 (l'échec signifie que l'on a lu tous les tuples présents dans le curseur).

4. Avant la fin de la boucle, il faut procéder à la lecture du résultat suivant afin qu'il puisse à son tour être traité :

```
    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
```

Dans cet exemple, on effectue les étapes suivantes :

1. On déclare un curseur nommé `crsrClientA_F` qui est associé à une requête qui renvoie le nom et l'adresse des clients dont la première lettre du nom est comprise entre A et F.
2. On ouvre ensuite le curseur ce qui revient à executer la requête et stocker le résultat dans une structure interne que l'on peut apparenter à un tableau à deux dimensions

(dans notre exemple, ce tableau aura 2 colonnes et autant de lignes qu'il y a de clients dont la première lettre du nom est comprise entre A et F).

3. On extrait le premier tuple du résultat et on affecte les valeurs des colonnes `nom` et `adresse` à des variables précédemment déclarées `@nom` et `@adresse`. Si la requête associée au curseur n'a renvoyé aucun résultat (il n'y a pas de client dont la première lettre du nom est comprise entre A et F), la variable système `@@FETCH_STATUS` prendra un valeur negative, sinon elle prendra la valeur 0.
4. On évalue le gardien de boucle @@FETCH_STATUS qui permet (ou non) d'exécuter le corps de la boucle. Si le gardien de boucle est évalué à FALSE, on passe à l'étape 8.
5. On effectue le traitement spécifique du tuple qui vient d'être extrait via les variables @nom et @adresse. Dans notre exemple, il s'agit simplement de provoquer un affichage via l'instruction PRINT.
6. On extrait le tuple suivant du résultat et on affecte les valeurs des colonnes `nom` et `adresse` à des variables précédemment déclarées `@nom` et `@adresse`. Si tous les tuples du résultat ont été préalablement extraits et traités, la variable système `@@FETCH_STATUS` prendra un valeur negative, sinon elle prendra la valeur 0.
7. On passe à l'étape 4.
8. On ferme le curseur
9. On désalloue les ressources

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 paramètres éventuels. Pour chaque paramètre, 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 du paramètre 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 Evénements 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	*inserted* accessible
UPDATE	*inserted* et *deleted* accessibles
DELETE	*deleted* 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 quelle instruction RAISERROR a été exécutée et ainsi plus facilement en comprendre l'origine.

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 pseudo-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). Cette situation n'est pas idéale d'un point de vue performance. Dans une telle situation, on préférera cibler les clients impactés par l'instruction UPDATE en interrogeant la pseudo-table INSERTED.

```
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 tuples clients impactés (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) :

- ***A*tomicité** – Les commandes SQL faisant partie de la transaction sont exécutées complètement ou pas du tout
- ***C*ohérence** – Si l'état de la base de données était cohérent avant l'exécution de la transaction, il le sera après également.
- ***I*solation** – 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
- ***D*urabilité** – 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 coté 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*, 2ème édition, Armand Colin, 1994, Paris

[2] P. DELMAL, *SQL2 – SQL3 : application à Oracle*, 3ème édition, De Boeck Université, 2001, Bruxelles

[3] Microsoft, MSDN Microsoft Developper 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 : Fonctions de SQL Server

Cette partie sera utilisée au second quadrimestre lors du laboratoire sur la programmation SQL.

A.1 Fonctions sur les dates

Ainsi, Microsoft propose dans SQL Server, les fonctions suivantes :

DATEADD	Retourne une nouvelle date dont un intervalle a été ajouté à la date donnée
DATEDIFF	Retourne, sous la forme d'un entier, la différence entre 2 dates données
DATENAME	Retourne, sous la forme d'une chaîne, une partie de la date donnée
DATEPART	Retourne, sous la forme d'un entier, une partie de la date donnée
DAY	Retourne le jour d'une date donnée
GETDATE	Retourne la date du jour (et l'heure)
GETUTCDATE	Retourne la date du jour (et l'heure) en mode UTC GMT (Greenwich)
MONTH	Retourne le mois d'une date donnée
YEAR	Retourne l'année d'une date donnée
CONVERT	Conversion explicite

Source : Tableau extrait de [3]

a) DATEADD

Retourne une nouvelle date dont un intervalle a été ajouté à la date donnée.

Format :

```
DATEADD (datepart, number, date)
```

Le champ *datepart* mentionne l'élément de la date *date* sur lequel l'entier *number* sera ajouté. Il est possible de mentionner, comme valeur de *datepart*, les codes suivants :

- Année : `year` ou les codes `yy` ou `yyyy`
- Mois : `month` ou les codes `mm` ou `m`
- Jour dans l'année : `dayoftheyear` ou les codes `dy` ou `y`
- Jour : `day` ou les codes `dd` ou `d`
- Semaine : `week` ou les codes `wk` ou `ww`
- Heure : `hour` ou le code `hh`
- Minute : `minute` ou les codes `mi` ou `n`
- Seconde : `second` ou les codes `ss` ou `s`
- Milliseconde : `millisecond` ou le code `ms`

Exemple :

```
SELECT NumClient, DATEADD(day, 15, DateCommande)
FROM Commandes
```

Sélectionne les clients ayant commandé et affiche la date de leur commande augmentée de 15 jours.

b) DATEDIFF

Retourne, sous la forme d'un entier, la différence entre 2 dates données

Format :

```
DATEDIFF (datepart, startdate, enddate)
```

Retourne un entier représentant la différence entre les dates *startdate* et *enddate*. Cet entier peut représenter, suivant le code utilisé, un nombre d'années, un nombre de mois, un nombre de jours dans l'année, un nombre de jours, un nombre de semaines, un nombre d'heures, de minutes, de secondes, de millisecondes. Le format de ce champ est identique à celui vu précédemment.

Exemple :
```
SELECT NumClient, DATEDIFF(year, DateCommande, GETDATE())
FROM Commande
```
Sélectionne les clients ayant commandé et affiche la différence (en année) entre la date de cette commande et la date du jour.

c) DATENAME
Retourne, sous la forme d'une chaîne, une partie de la date donnée

Format :
```
DATENAME (datepart, date)
```

Retourne sous la forme d'une chaîne la partie *datepart* de la date *date* donnée. Les paramètres de *datepart* sont inchangés.

Exemple :
```
SELECT NumClient, DATENAME(mm, DateCommande)
FROM Commande
```
Sélectionne les clients ayant commandé et affiche le nom du mois durant lequel la commande a été passée.

d) DATEPART
Retourne sous la forme d'un entier, un élément de la date donnée

Format :
```
DATEPART (datepart, date)
```

Retourne un entier représentant l'élément *datepart* de la date *date* donnée. Les paramètres de *datepart* sont inchangés.

Exemple :
```
SELECT NumClient, DATEPART(mm, DateCommande)
FROM Commande
```
Sélectionne les clients ayant commandé et affiche le mois durant lequel la commande a été passée.

e) DAY – MONTH – YEAR
Retourne le jour, le mois ou l'année d'une date donnée.

Format :
```
DAY (date)
MONTH (date)
YEAR (date)
```

Retourne l'élément jour, mois ou année de la date *date* donnée. Ces fonctions sont équivalentes à

- `DAY (date) = DATEPART(dd, date)`
- `MONTH (date) = DATEPART(mm, date)`
- `YEAR (date) = DATEPART(yy, date)`

f) GETDATE – GETUTCDATE
Retourne la date du jour (et l'heure), éventuellement en mode UTC ou GMT (Greenwich).

Format :
```
GETDATE ()
GETUTCDATE ()
```

Retourne, sous la forme d'une date, la date et l'heure du jour (éventuellement en mode GMT).

Exemple :
```
SELECT NumClient, DATEDIFF(year, DateCommande, GETDATE())
FROM Commande
```
Sélectionne les clients ayant commandé et affiche la différence (en année) entre la date de cette commande et la date du jour.

g) CONVERT
Réalise une conversion explicite.

Format :
```
CONVERT (data_type [(length)], expression [, style])
```

Convertit l'expression *expression* dans le type de donnée *data_type* (avec éventuellement une limite sur la longueur en respectant le style *style* mentionné.

Le style peut être :

- 1 pour la date sous la forme `mm/dd/yy` ou 101 pour `mm/dd/yyyy`
- 3 pour la date sous la forme `dd/mm/yy` ou 103 pour `dd/mm/yyyy`
- 5 pour la date sous la forme `dd-mm-yy` ou 105 pour `dd-mm-yyyy`
- 7 pour la date sous la forme `Mon dd, yy` ou 107 pour `Mon dd, yyyy`
- 8 ou 108 pour l'heure sous la forme `hh:mm:ss`
- 13 ou 113 pour un format `dd mon yyyy hh:mi:ss:mmm`
- 14 ou 114 pour l'heure sous la forme `hh:mi:ss:mmm` (24h)

Exemple :
```
INSERT INTO Commande VALUES (…, CONVERT(smalldatetime, '13/03/2000',103))
```
Insère un enregistrement dans la table commande en créant une date à partir de la date fournie en paramètre.

A.2 Fonctions sur les chaines

Les fonctions suivantes sont définies dans Transact-SQL, le langage de Microsoft SQL Server (voir [3]) :

CHARINDEX	Retourne la position d'une chaîne donnée

LEFT – RIGHT	Retourne une sous-chaîne en partant du début ou de la fin d'une chaîne donnée
LEN	Retourne la longueur d'une chaîne
LOWER – UPPER	Transforme une chaîne en minuscule ou majuscule
LTRIM – RTRIM	Retourne la chaîne amputée des caractères espaces se trouvant au début à la fin de celle-ci.
REPLACE	Remplace les éléments d'une chaîne
STR	Convertit une donnée numérique en chaîne
SUBSTRING	Extrait une sous-chaîne

a) CHARINDEX
Retourne la position d'une chaîne donnée

Format :
```
CHARINDEX (str1, str2[, pos])
```

Retourne la position de la chaîne *str1* dans le chaîne *str2*, à partir de la position *pos*. Si *pos* n'est pas mentionné, la recheche s'effectue du début (1ère position). Si la chaîne n'est pas trouvée, la valeur 0 est retournée.

Exemple :
```
SELECT * FROM produit
WHERE CHARINDEX('dupont', nomProduit) > 0
```

b) LEFT – RIGHT
Retourne une sous-chaîne en partant du début ou de la fin de la chaîne donnée.

Format :
```
LEFT (str, len)
RIGHT (str, len)
```

Dans le cas de `LEFT`, retourne les len premiers caractères de la chaîne str. Dans le cas de `RIGHT`, retourne les len derniers caractères de la chaîne str.

Exemple :
```
SELECT * FROM client
WHERE LEFT(nomClient, 1) <> UPPER(LEFT(nomClient, 1))
```
Retourne le nom des clients dont la 1ère lettre n'est pas une majuscule.

c) LEN
Retourne la longueur d'une chaîne donnée

Format :
```
LEN (str)
```

Retourne la longueur de la chaîne *str*.

Exemple :
```
SELECT numClient, nomClient, LEN(nomClient)
FROM client
```
Retourne le numéro, le nom et la longueur du nom des clients présents dans la table.

d) LOWER – UPPER
Transforme la chaîne en minuscule ou majuscule.

Format :
```
LOWER(str)
UPPER(str)
```

Retourne la chaîne str en minuscule (dans le cas de LOWER) ou majuscule (dans le cas de UPPER).

Exemple :
```
UPDATE client
SET nomClient = UPPER(nomClient)
```
Transforme le nom des clients existants en majuscule.

e) LTRIM – RTRIM
Retourne la chaîne amputée des caractères espaces se trouvant au début à la fin de celle-ci.

Format :
```
LTRIM(str)
RTRIM(str)
```

Supprime les caractères blanc se trouvant au début (dans le cas de `LTRIM`) ou à la fin (dans le cas de `RTRIM`) de la chaîne *str*.

Exemple :
```
SELECT * from Client
WHERE LTRIM(nomClient) = 'dupont'
```
Retourne les clients dont le nom est dupont, peu importe si celui-ci contient des espaces au début.

f) REPLACE
Remplace les éléments d'une chaîne

Format :
```
REPLACE (str1, str2, str3)
```

Retourne la chaîne *str1* dans laquelle toutes les occurrences de *str2* ont été remplacée par *str3*.

Exemple :
```
UPDATE Produit
SET libelle = REPLACE(libelle, 'digital', 'numérique')
```
Remplace, dans le libellé de la table Produit, *digital* par *numérique*.

g) STR
Convertit une donnée numérique en chaîne

Format :
```
STR(float, len, prec)
```

Convertit la donnée numérique *float* de longueur *len* en chaîne avec une précision décimale de *prec* (nombre de chiffre après la virgule).
Exemple (tiré de [3]) :

```
SELECT STR(123.45, 6, 1)
```

Cet exemple retournera la chaîne 123.5 puisque la précision décimale est placée à 1.

h) SUBSTRING

Extrait une sous-chaîne

Format :

```
SUBSTRING(str, pos, len)
```

Extrait de la chaîne *str* , la sous-chaîne débutant à la position *pos* et d'une longueur de *len*.

Exemple :

```
SELECT * FROM Client
WHERE SUBSTRING(nomClient, 2, 5) = 'upont'
```

Cet exemple retourne tous les clients dont le nom contient la sous-chaîne 'upont' commençant à la position 2.