Chapitre 8

Dans ce chapitre vous allez apprendre :

Comment utiliser le SQL comme langage de programmation

Comment utiliser les **curseurs** SQL

Comment créer des **procédures d’emmagasinage**

Comment créer des **déclencheurs**

Comment **utiliser** des **déclencheurs** pour **renforcer** les **contraintes** d’**intégrités**

Les **avantages** et les **désavantages** des **déclencheurs**

Comment utiliser des **requêtes récursives**

Les deux chapitres précédents se sont concentrés sur le langage principal du SGBD relationnel, à savoir SQL. Au chapitre 6, nous avons examiné les instructions ***DML*** (**langage de manipulation des données**) de SQL: SELECT, INSERT, UPDATE et DELETE. Au chapitre 7, nous avons examiné les principales instructions du **langage de définition de données (DDL)** du langage SQL. Telles que les instructions CREATE et ALTER TABLE et l'instruction CREATE VIEW.

Dans ce chapitre, nous traitons d’autres parties du standard SQL, à savoir:

**Le langage de programmation SQL SQL / PSM**

Curseurs SQL

Procédures stockées

Déclencheurs

Requêtes récursives

À l'avenir, au chapitre 9, nous aborderons les fonctionnalités ajoutées aux spécifications SQL pour prendre en charge la gestion de données orientée objet, et au chapitre 30, les fonctionnalités ajoutées à la spécification pour prendre en charge XML (eXtensible Markup Language), appelé SQL / XML: 2011. Enfin, dans l'annexe I, nous expliquons comment SQL peut être intégré dans des langages de programmation de haut niveau. Les exemples de ce chapitre utilisent la base de données de location DreamHome illustrée à la figure 4.3.

# 8.1 Le langage de programmation SQL

Les ***deux versions initiales*** du langage **SQL** ne **comportaient aucune construction de programmation**; c'est-à-dire que ce n'était pas **complètement computationnel**). Pour résoudre ce problème, les versions les plus récentes du standard SQL permettent à SQL d'être incorporé dans des langages de programmation de haut niveau afin de développer des applications de base de données plus complexes (voir l'annexe I).

Cependant, cette approche **produit** un **déséquilibre d’impédance**, car nous mélangeons différents paradigmes de programmation:

**SQL** est un **langage déclaratif qui gère plusieurs lignes de données**, alors qu'un **langage de haut** **niveau** tel que « C » **est un langage procédural** qui ne **peut gérer qu'une ligne de données** à la fois.

SQL et 3GL utilisent différents modèles pour représenter les données. Par exemple, SQL fournit les types de données intégrés Date et Interval, qui ne sont pas disponibles dans le langage de programmation traditionnel. Ainsi, il est nécessaire que le programme d'application effectue la conversion entre les deux représentations, ce qui est inefficace à la fois en termes de programmation et d'utilisation des ressources d'exécution. On estime qu'environ 30% des efforts de programmation et de l'espace de code sont consacrés à ce type de conversion (Atkinson et al., 1983). De plus, comme nous utilisons deux systèmes de types différents, il n’est pas possible de saisir automatiquement l’application dans son ensemble.

Il est avancé que la solution à ces problèmes ne consiste pas à remplacer le langage relationnel par des langages orientés objet au niveau de l'enregistrement, mais à introduire des fonctionnalités de niveau défini dans les langages de programmation (Date, 1995). Cependant, SQL est maintenant un langage de programmation complet et nous discutons de certaines des constructions de programmation dans cette section. Les extensions sont appelées SQL / PSM (**Persistant Stored Modules**); Cependant, pour rendre la discussion plus concrète, nous nous basons principalement sur le langage de programmation Oracle, PL / SQL.

**PL/SQL**. (Procedural Language/SQL) **est l’extension procédurale du langage SQL de Oracle**. Il **existe deux versions** de PL / SQL: l’**une** fait partie du **serveur Oracle** et l’**autre** est un **moteur** **distinct intégré** dans un certain **nombre d’outils Oracle**. Ils sont très similaires les uns aux autres et ont les mêmes constructions de programmation, la même syntaxe et les mêmes mécanismes logiques, bien que les outils PL / SQL pour Oracle aient une extension adaptée aux besoins de l’outil concerné (par exemple, PL / SQL a des extensions pour Oracle Forms) .

PL / SQL a des concepts similaires aux langages de programmation modernes, tels que les déclarations de variable et de constante, les structures de contrôle, la gestion des exceptions et la modularisation. PL / SQL est un langage structuré en blocs: les blocs peuvent être entièrement séparés ou imbriqués les uns dans les autres. Les unités de base constituant un programme PL / SQL sont les procédures, les fonctions et les blocs anonymes (sans nom). Comme illustré à la figure 8.1, **un bloc PL / SQL** comporte **jusqu'à trois parties**:

Une **partie déclaration facultative**, dans laquelle les variables, les constantes, les curseurs et les exceptions sont définis et éventuellement initialisés:

[DECLARE Optionnel

--- déclarations]

Une **partie exécutable obligatoire**, dans laquelle les variables sont manipulées:

BEGIN Obligatoire

--- instructions

Une **partie exception facultative**, permettant de gérer les exceptions déclenchées lors de l'exécution:

[EXCEPTION Optionnel

--- Gestionnaires d'exception]

END; Obligatoire

***GD : Exemple pris de*** [***https://www.tutorialspoint.com/plsql/plsql\_basic\_syntax.htm***](https://www.tutorialspoint.com/plsql/plsql_basic_syntax.htm)

DECLARE

message varchar2(20):= 'Hello, World!';

BEGIN

dbms\_output.put\_line(message);

END;

# 8.1.1 Déclarations

Les **variables** et les **constantes** doivent être déclarées avant qu’elles **soient appelées** par **d’autres instructions**, en incluant les instructions de déclarations. Exemple de déclarations :

**(DECLARE)**

*vStaffNo* ***VARCHAR2****(5)*

*vRent* ***NUMBER****(6,2) NOT NULL := 600;*

*MAX\_PROPERTIES* ***CONSTANT NUMBER*** *:= 100;*

Notez qu'il est possible de **déclarer une variable comme NOT NULL**, bien que dans ce cas, **une valeur initiale doit être affectée à la variable**. Il est également possible de déclarer une variable du même type qu'une colonne d'une table spécifiée ou une autre variable en utilisant l'attribut **%TYPE**. Par exemple, pour déclarer que la variable vStaffNo est du même type que la colonne staffNo de la table Staff, nous pourrions écrire:

vStaffNo Staff.staffNo%TYPE; variable vStaffNo même type que l’attribut Staff.StaffNo

vStaffNo1 vStaffNo%TYPE; variable vStaffNo1 même type que la variable vStaffNo

De même, nous pouvons déclarer une variable du même type qu'une ligne entière d'une table ou d'une vue en utilisant l'attribut %ROWTYPE. Dans ce cas, les champs de l'enregistrement prennent leurs noms et types de données dans les colonnes de la table ou de la vue. Par exemple, pour déclarer une variable vStaffRec comme une ligne de la table Staff, nous pourrions écrire:

***vStaffRec Staff%ROWTYPE;***

Noter que **%TYPE** et **%ROWTYPE** ***n’est pas du SQL standard***.

***GD : Exemple tiré du site d’ORACLE*** <https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABFAAAJ>

SQL> DECLARE

2 CURSOR **c1** IS

3 SELECT last\_name, salary, hire\_date, job\_id

4 FROM employees

5 WHERE employee\_id = 120;

6

7 **employee\_rec c1%ROWTYPE;**

8

9 BEGIN

10 OPEN c1;

11 **FETCH c1 INTO employee\_rec;**

12 DBMS\_OUTPUT.PUT\_LINE('Employee name: ' || employee\_rec.last\_name);

13 END;

14 /

Employee name: Weiss

PL/SQL procedure successfully completed.

SQL>

***GD COMMENTAIRE : La variable employee\_rec est composé de :***

***employee\_rec.last\_name***

***employee\_rec.salary***

***employee\_rec.hire\_date***

***employee\_rec.job\_id***

***En fait c’est plus une structure de donnée qu’une variable =)***

# 8.1.2 Assignations

Dans la partie exécutable d'un bloc PL / SQL, les variables peuvent être affectées de deux manières : en utilisant l'instruction d'affectation normale (:=) ou à la suite d'une instruction SQL SELECT ou FETCH. Par exemple :

GD commentaire : on verra FETCH plus tard, ça concerne les curseurs.

vStaffNo := ‘SG14’;

vRent := 500;

**SELECT COUNT(\*) INTO** x FROM PropertyForRent Where staffNo = vStaffNo;

Dans le troisième cas, la variable x est définie sur le résultat de l'instruction SELECT (égale au nombre de propriétés gérées par l'agent SG14).

Notez que dans le **standard SQL**, une **affectation utilise** le mot clé **SET** au début de la ligne **avec** le symbole **“=”**, au lieu du “: =”. Par exemple:

***SET vStaffNo = ‘SG14’***

# 8.1.3 Les Déclarations de Controle

PL / SQL prend en charge le mécanisme habituel à flux de contrôle conditionnel, itératif et séquentiel

Instruction IF conditionnelle : Les instructions IF ont la forme suivante :

IF (condition) Then

<SQL statement list>

[ELSIF (condition) THEN <SQL statement list> ] gd(ELSIF concerne que Oracle)

[ELSE <SQL statement list> ]

END IF

Notez que **le standard SQL** spécifie **ELSEIF** au lieu de **ELSIF**

Par exemple :

IF (position = ‘Manager’) THEN

salary := salary \*1.05;

ELSE

salary := salary\*1.03;

END IF;

Instruction de cas conditionnelle L'instruction CASE permet de sélectionner un chemin d'exécution en fonction d'un ensemble d'alternatives et se présente sous les formes suivantes :

CASE (opérande)

[WHEN (whenOperandList) | WHEN (searchCondition)

THEN <SQL Statement list> }

[ELSE <SQL statement list>

END CASE;

Par exemple:

Case lowercase(x)

WHEN ‘a’ THEN x := 1;

WHEN ‘b’ THEN x := 2;

y := 0;

WHEN ‘default’ THEN x :=3;

END CASE;

UPDATE Staff

SET salary = CASE

WHEN position = ‘Manager’

THEN salary \* 1.05

ELSE

THEN salary \* 1.02

END;

Déclaration d'itération (LOOP) : La déclaration LOOP a la forme suivante :

[labelNAME:]

LOOP

<SQL Statement list>

EXIT [labelNAME] [WHEN(condition)] gd :(remplace par LEAVE)

END LOOP [labelNAME]

Notez que **le standard SQL** spécifie **LEAVE** au lieu de **EXIT WHEN** (condition)

**loop**

**[ exit [when condition]];**

**<instructions>**

**end loop;**

**while condition**

Par Exemple :

x := 1;

myLOOP : optionnel

LOOP

x := x + 1;

IF ( x > 3) THEN

EXIT myLOOP; --- quitte la boucle immédiatement.

END LOOP myLOOP;

--- le contrôle se resume ici

y := 2;

Dans cet exemple, la boucle est terminée lorsque **x devient supérieur à 3** et le contrôle reprend immédiatement après le mot clé **END LOOP**.

Déclaration d'itération (**WHILE** et **REPEAT**) Les instructions WHILE et REPEAT ont la forme suivante (notez que PL / SQL (**Oracle**)). N'a **pas d'équivalent** à la **boucle REPEAT** spécifiée dans le standard SQL:

**PL/SQL**  **SQL**

**WHILE** (condition) **LOOP** **WHILE** (condition) **DO**

<SQL statement list> <SQL statement list>

**END LOOP**; **END WHILE** [labelName];

**REPEAT**

<SQL statement list>

**UNTIL** (condition)

**END REPEAT** [labelNAME]

Déclaration d'itération **(FOR)** La déclaration FOR a la forme suivante:

**PL/SQL** **SQL**

**FOR** indexVariable **FOR** indexVariable

**IN** lowerBound.. upperBOUND **LOOP** **AS** querySpecification **DO**

<SQL statement list> <SQL Statement list>

**END LOOP** [labelName]; **END FOR** [labelNAME];

Voici un exemple de boucle FOR en PL / SQL:

***DECLARE***

numberOfStaff **NUMBER**;

**SELECT COUNT(\*)** **INTO** numberOfStaff **FROM** PropertyForRent

**WHERE** staffNO **=** ‘SG14’**;** myLoop1**:**

**FOR** iStaff **IN** **1 ..** numberOfStaff **LOOP**

….

**END LOOP**

myLoop1**;**

Voici un exemple de boucle FOR en SQL standard:

myLoop1:

**FOR** iSTaff **AS SELECT COUNT(\*) FROM** PropertyForRent

**WHERE** staffNo **=** ‘SG14’ **DO**

….

**END FOR** myLoop1;

Nous présentons sous peu d'autres exemples utilisant certaines de ces structures.

# 8.1.4 Les Exceptions en PL/SQL

Une **exception** est **un identifiant** dans PL / SQL **généré** **lors** de l'**exécution** d'un **bloc** qui **termine son corps principal d'actions**. Un bloc **se termine toujours** **lorsqu**'une **exception est déclenchée**, bien que le gestionnaire d'exceptions puisse effectuer certaines actions finales. Une **exception peut**, **être générée automatiquement** par **Oracle**. Par exemple, l'exception NO\_DATA\_FOUND est déclenchée chaque fois qu'aucune ligne n'est extraite de la base de données dans une instruction SELECT. Il est également possible **qu'une exception soit déclenchée explicitement** à l'aide de **l'instruction RAISE**. Pour gérer les exceptions déclenchées, des routines distinctes, appelées gestionnaires d'exceptions, sont spécifiées.

Comme mentionné précédemment,

**Une exception définie par l'utilisateur est définie dans la partie déclarative d'un bloc PL / SQL**.

**Dans la partie exécutable, une vérification de la condition d'exception est effectuée et, si elle est trouvée, l'exception est déclenchée**.

**Le gestionnaire d'exceptions lui-même est défini à la fin du bloc PL / SQL**. Un exemple de traitement des exceptions est donné à la figure 8.2. Cet exemple illustre également l'utilisation du paquet fourni par Oracle : ***DBMS\_OUTPUT***,

DECLARE

vpCount NUMBER;

vStaffNo PropertyForRent.staffNO%TYPe := ‘SG14’;

--- define an exception for the enterprise constraint that prevents a member of staff managing more than 100 properties

e\_too\_many\_properties **EXCEPTION**;

**PRAGMA EXCEPTION\_INIT**(e\_too\_many\_properties, -20000);

BEGIN

SELECT COUNT(\*) INTO vpCount

FROM PropertyForRent

WHERE staffNo = vStaffNo;

IF vpCount = 100

--- raise an exception for the general constraint

**RAISE** e\_too\_many\_properties

END IF;

UPDATE PropertyForRent SET staffNo = vSTaffNo WHERE propertyNO = ‘PG4’;

**EXCEPTION**

--- handle the exception for the general constraint

**WHEN** e\_too\_many\_properties **THEN**

**Dbms\_output.put\_line**(‘Member of staff’ || staffNo || ‘already managing 100 properties’);

END;

*GD Commentaire: Pragma est une directive passé au compilateur, elle n’est pas transformée au moment de l’exécution.*

Ce qui permet la sortie de blocs et de sous-programmes PL / SQL. La procédure **put\_line** ***renvoie les informations dans un tampon*** de la **SGA (**Shared Global Area) (une zone de mémoire partagée utilisée pour stocker des informations et des informations de contrôle pour une instance Oracle), qui peuvent être affichées en appelant la procédure **get\_line** ou en définissant **SERVEROUTPUT ON** dans SQL \* Plus

*GD : Commentaire : En gros, de base lorsque on fait :*

*DBMS\_OUTPUT.PUT\_LINE(paramètre)*

*Le terminal n’affichera pas la valeur qui est inscrite dans le paramètre par défaut il va tout simplement écrire : PL/SQL procedure sucessfully completed*

*Pour afficher les valeurs il faut mettre* ***SERVEROUPUT ON*** *dans la section déclaration.*

*La* ***différence*** *entre* ***get\_line*** *et* ***put\_line*** *est que qu’avec* ***get\_line*** *on doit créer un buffer qui va aller chercher des valeurs. Tandis que le put\_line on définit directement des valeurs et des variables à l’intérieur.*

set serveroutput on  
  
[DECLARE](http://psoug.org/definition/DECLARE.htm)  
 buffer [VARCHAR2](http://psoug.org/definition/VARCHAR2.htm)(100);  
 status [INTEGER](http://psoug.org/definition/INTEGER.htm);  
[BEGIN](http://psoug.org/definition/BEGIN.htm)  
  dbms\_output.put\_line('This is');  
  dbms\_output.put\_line('a test.');  
  dbms\_output.get\_line(buffer, status);  
  dbms\_output.put\_line('Buffer: ' || buffer);  
  dbms\_output.put\_line('Status: ' || [TO\_CHAR](http://psoug.org/definition/TO_CHAR.htm)(status));  
[END](http://psoug.org/definition/END.htm);

Traitement de condition

Le langage **SQL / PSM** (**SQL Persistent Stored Modules**) inclut la gestion des conditions pour gérer les exceptions et les conditions d'achèvement. La gestion des conditions commence par définir un gestionnaire en spécifiant son type, les conditions d’exception et d’achèvement qu’il peut résoudre, ainsi que l’action à entreprendre (instruction de procédure SQL). La gestion des conditions permet également de signaler explicitement des conditions d'exception et de complétion à l'aide de l'instruction **SIGNAL / RESIGNAL**.

Un gestionnaire pour une exception ou une condition d'achèvement associée peut être déclaré à l'aide de l'instruction **DECLARE… Handler**:

DECLARE {

CONTINUE | EXIT | UNDO

} HANDLER

FOR SQLSTATE {sqlstateValue | conditionNAME | SQLEXCEPTION | SQLWARNING | NOT FOUND } handlerAction;

Un nom de condition et une valeur SQLSTATE correspondante facultative peuvent être déclarés en utilisant:

DECLARE conditionName CONDITION

[FOR SQLSTATE sqlstateValue]

Et une condition d'exception peut être signalée ou renvoyée à l'aide de :

SIGNAL sqlstateValue; or RESIGNAL sqlstateValue;

Lorsqu'une instruction composée contenant une déclaration de gestionnaire est exécutée, un gestionnaire est créé pour les conditions associées. Un gestionnaire est activé lorsqu'il est le gestionnaire le plus approprié pour la condition générée par l'instruction SQL. Si le gestionnaire a spécifié **CONTINUE**, il activera le gestionnaire **EXIT** après l'activation, puis, après avoir exécuté l'action du gestionnaire, il laissera l'instruction composée. Si le type de gestionnaire est **UNDO**, il annule toutes les modifications apportées dans l'instruction composée. Si le gestionnaire ne termine pas avec une condition d'achèvement réussie, un renvoi implicite est exécuté, ce qui détermine s'il existe un autre gestionnaire capable de résoudre la condition.

# 8.1.5 Curseurs dans PL/SQL

Une instruction **SELECT** peut, être utilisée **si la requête renvoie une et une seule ligne**. Pour traiter une requête pouvant renvoyer un nombre arbitraire de lignes (c'est-à-dire zéro, une ou plusieurs lignes), **PL / SQL utilise des curseurs** pour permettre l'accès aux lignes d'un résultat de requête, une à la fois. En effet, le curseur agit comme un pointeur sur une ligne particulière du résultat de la requête. Le curseur peut être avancé de 1 pour accéder à la ligne suivante. Un curseur doit être déclaré et ouvert avant de pouvoir être utilisé, et il doit être fermé pour le désactiver lorsqu'il n'est plus nécessaire. Une fois le curseur ouvert, les lignes du résultat de la requête peuvent être extraites une à une en utilisant une instruction **FETCH**, par opposition à une instruction SELECT. (Dans l'annexe 1, nous voyons que SQL peut également être intégré dans des langages de programmation de haut niveau et que des curseurs sont également utilisés pour gérer des requêtes pouvant renvoyer un nombre arbitraire de lignes.)

# Example 8.1 Utilisation des curseurs

La figure 8.3 illustre l'utilisation d'un curseur pour déterminer les propriétés gérées par le membre du personnel SG14. Dans ce cas, la requête peut renvoyer un nombre arbitraire de lignes. Un curseur doit donc être utilisé. Les points importants à noter dans cet exemple sont les suivants :

Dans la section DECLARER, le curseur propertyCursor est défini

Dans la section des déclarations, le curseur est d'abord ouvert. Cela a notamment pour effet d'analyser l'instruction SELECT spécifiée dans la déclaration CURSOR, d'identifier les lignes satisfaisant les critères de recherche (appelé ensemble actif) et de positionner le pointeur juste avant la première ligne de l'ensemble actif. Notez que si la requête ne renvoie aucune ligne, PL / SQL ne déclenche pas d'exception lorsque le curseur est ouvert.

Le code parcourt ensuite chaque ligne du jeu actif et récupère les valeurs de la ligne actuelle dans des variables de sortie à l'aide de l'instruction FETCH INTO. Chaque instruction FETCH fait également avancer le pointeur sur la ligne suivante du jeu actif.

Le code vérifie si le curseur ne contient pas de ligne (propertyCursor %NOTFOUND)

Et quitte la boucle si aucune ligne n'a été trouvée (EXIT WHEN). Sinon, il affiche les détails de la propriété à l'aide du package DBMS\_OUTPUT et effectue à nouveau le tour de la boucle.

Le curseur est fermé à la fin des recherches.

Enfin, le bloc d'exception affiche toutes les conditions d'erreur rencontrées.

Outre %NOTFOUND, qui renvoie **TRUE** si **l'extraction** la plus récente **ne renvoie pas de ligne**, il existe d'autres attributs de curseur utiles :

%FOUND: Évalue à **TRUE** si l'extraction la plus récente retourne une ligne (complément de% NOTFOUND)

%ISOPEN: Évalue à **TRUE** si le curseur est ouvert

%ROWCOUNT: évalue le nombre total de lignes renvoyées jusqu'à présent.

Passer des paramètres au curseur PL / SQL permet de paramétrer le curseur, de sorte que la même définition de curseur puisse être réutilisée avec des critères différents. Par exemple, nous pourrions changer le curseur défini dans l’exemple précédent en faisant :

CURSOR propertyCursor (vStaffNo VARCHAR2) IS

SELECT propertyNO, street, city, postcode

FROM PropertyForRent

WHERE staffNo = vStaffNo

ORDER BY propertyNO;

***GD: Plutôt dur à comprendre, en gros On déclare un Curseur dans la section déclaration, puis on ouvre le pointeur dans la Section BEGIN en utilisant un argument spécifié, comme par exemple dans le dernier exemple on pourrait entrer :***

***OPEN propertyCursor (‘SA9’)***

***À ce moment-là le curseur va pointer ce qui a été définit dans le code de la déclaration en utilisant ce paramètre***

***Je préfère cet exemple sur le site d’oracle =)***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | DECLARE      rec\_product products%ROWTYPE;      CURSOR cur\_product (low\_price NUMBER, high\_price NUMBER)      IS          SELECT \*          FROM products  **WHERE list\_price BETWEEN low\_price AND high\_price;**  BEGIN      -- mass products      DBMS\_OUTPUT.PUT\_LINE('Mass products: ');      OPEN cur\_product(50,100);      LOOP          FETCH cur\_product INTO rec\_product;          EXIT WHEN cur\_product%NOTFOUND;          DBMS\_OUTPUT.PUT\_LINE(rec\_product.product\_name || ': ' ||rec\_product.list\_price);      END LOOP;      CLOSE cur\_product;        -- Luxury products      DBMS\_OUTPUT.PUT\_LINE('Luxury products: ');      OPEN cur\_product(800,1000);      LOOP          FETCH cur\_product INTO rec\_product;          EXIT WHEN cur\_product%NOTFOUND;          DBMS\_OUTPUT.PUT\_LINE(rec\_product.product\_name || ': ' ||rec\_product.list\_price);      END LOOP;      CLOSE cur\_product;    END; |

Et nous pourrions ouvrir le curseur en utilisant les exemples suivants:

vStaffNo1 PropertyForREnt.staffNo%TYPE: = ‘SG14’;

OPEN propertyCursor(‘SG14’);

OPEN propertyCursor(‘SA9’);

OPEN propertyCursor(vStaffNo1);

DECLARE

vPropertyNo PropertyForRent.propertyNO%TYPE;

vStreet PropertyForRent.street%TYPE;

vCity PropertyForRent.city%TYPE;

CURSOR propertyCursor IS

SELECT propertyNO, street, city, postcode

FROM PropertyForRent

WHERE staffNO = ‘SG14’

ORDER by propertyNo;

BEGIN

--- Ouvre le curseur au début de la sélection, puis boucle pour récupérer chaque ligne de la table de résultats

OPEN propertyCursor;

LOOP

--- Récupère la ligne suivante de la table de résultats

FETCH propertyCursor

INTO vPropertyNO, vStreet, vCity, vPostcode;

EXIT WHEN propertyCursor%NOTFOUND;

--- Affiche les données

Dbms\_output.put\_line(‘Property number:’ || vPropertyNo);

Dbms\_output.put\_line(‘Street: ‘ || vStreet);

Dbms\_output.put\_line(‘City: ‘ || vCity);

IF postcode IS NOT NULL THEN

Dbms\_output.put\_line(‘Post Code: ‘ || vPostcode);

ELSE

Dbms\_output.put\_line(‘ Post Code: NULL’);

END IF

END LOOP

IF propertyCursor%ISOPEN THEN CLOSE propertyCursor END IF;

--- Error condition – print out error

EXCEPTION

WHEN OTHERS THEN

Dbms\_output.put\_line(‘error detected’);

IF propertyCursor%ISOPEN THEN CLOSE propertyCursor; END IF;

END;

Figure 8.3

**Mise à jour de lignes** à l'aide d'un curseur : il est possible de mettre à jour et de supprimer une ligne après son extraction à l'aide d'un curseur. Dans ce cas, **pour s'assurer que les lignes ne sont pas modifiées entre la déclaration des curseurs**, son ouverture et leur extraction, la clause **FOR UPDATE** est ajoutée à la déclaration du curseur. Cela a pour effet **de verrouiller les lignes du jeu actif** pour **éviter tout conflit de mise à jour lorsque le curseur est ouvert** (les conflits de verrouillage et de mise à jour sont abordés au chapitre 22).

Par exemple, nous pouvons **vouloir réaffecter les propriétés** que SG14 gère et les donner à SG37. Les curseurs seraient maintenant déclarés comme:

***GD: Déjà pourquoi utiliser FOR UPDATE, à quoi ça sert? On l’utilise lorsqu’on veut faire une requête de quelques rangées et qu’on veut les mettre à jour par la suite, la clause FOR UPDATE va les bloquer les rangées pour que personne ne les corrompt pendant la modification.***

CURSOR propertyCursor IS

SELECT propertyNO, street, city, postcode

FROM PropertyForRent

WHERE staffNo = ‘SG14’

ORDER BY propertyNO

**FOR UPDATE NOWAIT**;

Par défaut, **si le serveur Oracle ne peut pas acquérir les verrous sur les lignes** du jeu actif dans **un curseur SELECT FOR UPDATE**, il attend indéfiniment. Pour éviter cela, le mot clé facultatif **NOWAIT** peut être spécifié et **un test peut être effectué pour voir si le verrouillage a réussi**. Lorsque vous passez en boucle sur les lignes du jeu actif, la clause **WHERE CURRENT OF** est ajoutée à **l'instruction SQL UPDATE ou DELETE** pour indiquer que la mise à jour doit être appliquée à la ligne actuelle du jeu actif.

***GD : On va rechercher des valeurs précises dans les tables avant de les mettre à jour ou les supprimer.***

Par exemple:

UPDATE PropertyForRent

Set StaffNo = ‘SG37’

WHERE CURRENT OF propertyCursor;

…

COMMIT;

<https://www.techonthenet.com/oracle/cursors/current_of.php>

# 8.2 Sous-programmes, procédures stockées, fonctions et packages

Les **sous-programmes** sont des blocs PL / SQL nommés qui peuvent prendre des **paramètres et être appelés**. PL / SQL a **deux types** de sous-programme appelés **procédures** et **fonctions** (stockées). Les procédures et fonctions peuvent prendre un ensemble de paramètres qui leur sont attribués par le programme appelant et effectuer un ensemble d’actions. **Les deux peuvent modifier et renvoyer les données** qui leur sont transmises en tant que paramètre. La **différence** entre une procédure et une fonction **est qu'une fonction retournera toujours une valeur unique** à l'appelant, contrairement à une procédure. **Habituellement, les procédures sont utilisées sauf si une seule valeur de retour est requise.**

Les procédures et les fonctions sont très similaires à celles de la plupart des langages de programmation de haut niveau et présentent les mêmes **avantages**: **elles apportent modularité et extensibilité, favorisent la réutilisation et la maintenabilité, et aident à l'abstraction**. Un paramètre a un nom et un type de données spécifiés, mais peut également être désigné comme:

**IN –** Ce paramètre est utilisé uniquement comme valeur d'entrée.

**OUT –** Ce paramètre est utilisé uniquement comme valeur de sortie.

**IN OUT** – paramètre est utilisé à la fois comme une valeur d'entrée et une valeur de sortie

Par exemple, nous pourrions changer le bloc anonyme PL / SQL donné à la FIGURE 8.3 en une procédure en ajoutant les lignes suivantes au début:

**CREATE OR REPLACE PROCEDURE** PropertiesForStaff

(**IN** vStaffNo **VARCHAR2**)

**AS . . .**

**create [or replace] procedure SP\_*NOM\_PROCEDURE***

**[(ARGUMENT\_1...[,ARGUMENT\_N])] is**

**[*Section de déclaration de variable*]**

**BEGIN**

***Section exécutable***

**[*Section\_gestion\_exception*]**

**end [*SP\_NOM\_PROCEDURE*];**

La procédure peut ensuite être exécutée dans SQL \* Plus en tant que:

SQL > SET SERVEROUTPUT ON;

SQL > EXECUTE PropertiesForStaff(‘SG14’);

Nous discutons des fonctions et des procédures plus en détail au chapitre 9.

Package (PL/SQL)

**Un package** est un **ensemble de procédures, fonctions, variables et instructions SQL regroupées et stockées dans une seule unité de programme**. Un paquet a **deux parties** : une **spécification** et **un corps**. **La spécification d’un paquet déclare toutes les constructions publiques du paquet** et **le corps définit toutes les constructions (publiques et privées) du paquet et implémente donc la spécification**. De cette manière, les packages fournissent une forme d'encapsulation. Oracle effectue les étapes suivantes lors de la création d'une procédure ou d'un package :

**Il compile la procédure ou le package.**

**Il stocke le code compilé en mémoire.**

**Il stocke la procédure ou le package dans la base de données.**

Pour l'exemple précédent, nous pourrions créer une spécification de paquet comme suit:

**CREATE OR REPLACE PACKAGE** StaffPropertiesPackage **AS**

**Procedure** PropertiesForStaff(vStaffNo **VARCHAR2**);

**END** StaffPropertiesPackage;

Et nous pourrions créer le corps du paquet (c'est-à-dire, son implémentation comme:

**CREATE OR REPLACE PACKAGE BODY** StaffPropertiesPackage

**AS**

….

**END** StaffPropertiesPackage;

Pour référencer les éléments déclarés dans une spécification de package, nous utilisons la notation par points. Par exemple, nous pourrions appeler la procédure PropertiesForStaff comme suit:

StaffPropertiesPackage.PropertiesForStaff(‘SG14’);

# 8.3 Déclencheurs

Un déclencheur définit une action que la base de données doit entreprendre lorsqu'un événement survient dans l'application. Un déclencheur peut être utilisé pour appliquer certaines contraintes d'intégrité référentielle, pour appliquer des contraintes complexes ou pour auditer les modifications apportées aux données. Le format général d'un déclencheur en SQL est:

**CREATE TRIGGER** TriggerName

**BEFORE | AFTER| INSTEAD OF**

**INSERT | DELETE | UPDATE** [ OF TriggerColumnList ]

**ON** TableName

[REFERENCING { OLD | NEW ) AS { OldName | NewName }

[FOR EACH {ROW | STATEMENT } ]

[WHEN Condition]

< trigger action >

Ce n'est pas la définition complète, mais c'est suffisant pour démontrer le concept de base. Le code contenu dans un déclencheur, appelé corps de déclencheur ou action de déclencheur, est constitué d'un bloc SQL. Les déclencheurs sont basés sur le modèle Event-Condition-Action (ECA):

L'événement (ou les événements) qui déclenchent la règle, qui peut être une instruction INSERT, UPDATE ou DELETE sur une table spécifiée (ou éventuellement une vue). Dans Oracle, cela peut aussi être:

* Les declarations CREATE, ALTER, ou DROP sur tout schéma objet.
* Un démarrage de la base de données ou un arrêt d'instance, ou une connexion ou déconnexion d'un utilisateur;
* Un message d'erreur spécifique ou tout message d'erreur

Il est également possible de spécifier si le déclencheur doit se déclencher avant ou après l'événement.

La condition qui détermine si l'action doit être exécutée. La condition est facultative, mais si elle est spécifiée, l'action ne sera exécutée que si la condition est vraie.

L'action à prendre. Ce bloc contient les instructions SQL et le code à exécuter lorsqu'une instruction de déclenchement est émise et que la condition de déclencheur est évaluée à true.

Il existe **deux types de déclencheurs**: **les déclencheurs de niveau ligne (FOR EACH ROW)** qui s'exécutent pour chaque ligne de la table affectée par l'événement déclencheur et **le déclencheur de niveau instruction (FOR EACH STATEMENT)** qui n'exécutent qu'un événement si plusieurs lignes. Sont affectés par l'événement déclencheur. SQL prend également en charge **les déclencheurs INSTEAD OF**, qui offrent un moyen transparent de modifier des vues qui ne peuvent pas être modifiées directement via des instructions SQL DML (INSERT UPDATE et DELETE). Ces déclencheurs sont appelés déclencheurs **INSTEAD OF car**, contrairement à d'autres types de déclencheurs, **le déclencheur est activé au lieu d'exécuter l'instruction SQL d'origine**. Les déclencheurs peuvent également s'activer l'un après l'autre. Cela peut se produire lorsque l'action de déclenchement modifie la base de données, ce qui a pour effet de déclencher un autre évent auquel est associé un déclencheur.

EXEMPLE 8.2 APRES Déclenchement au niveau de la ligne

Créez un déclencheur AFTER au niveau de la ligne pour conserver une trace d'audit de toutes les lignes insérées dans la table Personnel.

*GD : CREATE TRIGGER, AFTER INSERT, ça sert à déclencher une séquence de code après qu’une opération d’insertion a été fait sur une table. Dans cet exemple, on veut garder des informations sur la personne qui vient d’entrer des informations dans la table Staff. Mais bon ils font ça encore trop compliqué.*

*L’exemple sur ce site est meilleur :* <https://www.techonthenet.com/oracle/triggers/after_insert.php>

*Sur ce site on veut créer un déclencheur pour chaque commande qui a été créé, on veut garder les informations sur qui a créé la commande, le montant associé, la quantité etc.*

**CREATE TRIGGER** StaffAfterInsert

**AFTER INSERT ON** Staff

**REFERENCING NEW AS** new

**FOR EACH ROW**

**BEGIN**

**INSERT INTO** StaffAudit

**VALUES** (:new.staffNo, :new.fName, :new.lName, :new.position, :new.sex, :new.DOB, :new.salary, :new.branchNo);

**END;**

Notez que **le standard SQL** utilise **NEW ROW** au lieu de **NEW** et **OLD ROW** au lieu de **OLD**.

EXEMPLE 8.3 Utilisation d'un déclencheur BEFORE

DreamHome a une règle qui empêche un membre du personnel de gérer plus de 100 propriétés en même temps. Nous pourrions créer le déclencheur illustré à la figure 8.4 pour appliquer cette contrainte. Ce déclencheur est appelé avant qu'une ligne ne soit insérée dans le PropertyForRent

*GD : Assez facile celui-là, c’est un déclencheur qui s’exécute avant qu’on fasse des insertions sur des tables.*

**CREATE TRIGGER** StaffNotHandlingTooMUCH

**BEFORE INSERT ON** PropertyForRent

**REFERENCING NEW AS** newrow

**FOR EACH ROW**

**DECLARE**

vpCount **NUMBER;**

**BEGIN**

**SELECT COUNT(\*) INTO** vpCOUNT

**FROM** PropertyForRent

**WHERE** staffNO = :newrow.staffNo;

**IF** vpCount = 100

**Raise\_application\_error**(-20000, (‘Member’ || :newrow.staffNo || ‘already managing 100 properties’);

**END IF;**

**END;**

Figure 8.4 Déclencheur pour appliquer la contrainte selon laquelle un membre du personnel ne peut pas gérer plus de 100 propriétés à la fois.

La table ou la ligne existante est mise à jour. Si le membre du personnel gère actuellement 100 propriétés, le système affiche un message et annule la transaction. Les points suivants doivent être notés

Le mot clé BEFORE indique que le déclencheur doit être exécuté avant qu'une insertion ne soit appliquée à la table PropertyForRent.

Le mot clé FOR EACH ROW indique qu'il s'agit d'un déclencheur au niveau de la ligne, qui s'exécute pour chaque ligne de la table PropertyForRent mise à jour dans l'instruction.

EXEMPLE 8.4 Utilisation de déclencheurs pour appliquer l'intégrité référentielle

**Par défaut, Oracle applique les actions référentielles ON DELETE NO ACTION et ON UPDATE NO ACTION** **sur les clés étrangères nommées** (voir la section 7.2.4). Il permet également de **spécifier la clause supplémentaire ON DELETE CASCADE** afin de permettre la suppression en cascade des suppressions de la table parent vers la table enfant. **Cependant**, **il ne prend pas en charge l'action** **ON UPDATE CASCADE**, ni les actions **SET DEFAULT et SET NULL**. Si l'une de ces actions est requise, elle devra être implémentée en tant que déclencheur ou procédure stockée, ou dans le code de l'application. Par exemple, dans l'exemple 7.1, la clé étrangère keyNo dans la table PropertyForRent doit avoir l'action **ON UPDATE CASCADE**. Cette action peut être mise en œuvre à l'aide des déclencheurs illustrés à la figure 8.5.

Trigger I (PropertyForRent\_Check\_Before)

Le déclencheur de la figure 8.5 (a) est activé chaque fois que la colonne staffNo de la table PropertyForREnt est mise à jour. La vérification du déclencheur avant la mise à jour a lieu si la nouvelle valeur spécifiée existe dans la table Personnel. Si une exception Invalid\_Staff est déclenchée, le déclencheur génère un message d'erreur et empêche la modification de se produire.

--- Avant que la colonne staffNo ne soit mise à jour dans la table PropertyForRent, activez ce déclencheur

--- pour vérifier que la nouvelle valeur de clé étrangère est présente dans la table Personnel.

**CREATE TRIGGER** PropertyForRENT\_CHECK\_Before

**BEFORE UPDATE OF** staffNo **ON** PropertyForRent

**FOR EACH ROW WHEN** (new.staffNo **IS NOT NULL**)

**DECLARE**

Dummy **CHAR(5);**

Invalid\_staff **EXCEPTION;**

valid\_staff **EXCEPTION;**

mutating\_table **EXCEPTION;**

**PRAGMA EXCEPTION\_INIT** (mutating\_table, -4091);  
--- Utilisez les curseurs pour vérifier que la valeur de la clé parent existe.

--- Utilisez FOR UPDATE OF pour verrouiller la ligne de la clé parent afin qu’elle ne puisse pas être supprimée

--- par une autre transaction jusqu'à la fin de la transaction

**CURSOR** update\_cursor (sn **CHAR(5)**) **IS**

**SELECT** staffNO **FROM** Staff

**WHERE** staffNO **=** sn

**FOR UPDATE OF** staffNo;

**BEGIN**

**OPEN** update\_cursor (:new.staffNo);

**FETCH** update\_cursor **INTO** dummy;  
--- vérifier la clé parent. Relever les exceptions, le cas échéant.

**IF** update\_cursor%NOTFOUND **THEN**

**RAISE** invalid\_staff;

**ELSE**

**RAISE** valid\_staff;

**END IF;**

**CLOSE** update\_cursor;

**EXCEPTION**

**WHEN** invalid\_staff **THEN**

**CLOSE** update\_cursor;

**raise\_application\_error**(-20000, ‘Invalid Staff Number’ || :new.staffNo);

**WHEN** valid\_staff **THEN**

**CLOSE** update\_cursor;  
--- La table de mutation est une table en cours de modification par un INSERT, UPDATE,

--- ou déclaration DELETE, une déclaration qui pourrait devoir être mise à jour par les effets d’un déclaratif

--- DELETE CASCADE contrainte d'intégrité référentielle.

--- Cette erreur lève une exception, mais dans ce cas l'exception est OK, donc piègez-la, --- mais cela ne fais rien

**WHEN** mutating\_table **THEN**

**NULL;**

**END;**

*GD : En gros ce code permet de s’assurer que l’utilisateur va mettre à jour la clé étrangère associée à la relation. Cette séquence va vérifier si il y a une valeur dans la variable new.staffNo, si il n’y en as pas ou qu’elle est incorrecte un message d’erreur sera envoyé.*

Figure 8.5 (a) Déclencheurs Oracle à appliquer à ON UPDATE CASCADE sur la clé étrangère staffNo dans le tableau PropertyForRent lorsque la clé primaire staffNo est mise à jour dans la table Staff:

Trigger for the PropertyForRent table

Déclencheur pour la table PropertyForRent

Les trois déclencheurs illustrés à la figure 8.5 (b) sont activés chaque fois que la colonne staffNo du tableau Staff est mise à jour. Avant la définition des déclencheurs, un numéro de séquence updateSequence est créé, ainsi qu'une variable publique updateSeq (accessible par les trois déclencheurs via le package seqPackage). De plus, la table PropertyForRent est modifiée pour ajouter une colonne appelée updateId, utilisée pour indiquer si une ligne a été mise à jour, afin d'éviter qu'elle ne soit mise à jour plus d'une fois lors de l'opération en cascade.

*GD: Encore quelques trucs compliqués qu’il faut rechercher… déjà le CREATE Package c’Est une ~~structure de données~~ Le prof dit que ce n’Est pas une structure de données mais un conteneur de procédures et déclencheurs : seqpackage contient updateseq et on peut accéder à updateseq en faisant seqpackage.Updateseq*

*Lorsque un package contient plusieurs variables sous-programmes ou fonctions on doit définir un PACKAGE BODY. (pas à l’aise avec ce bout)*

--- Créez un numéro de séquence et une variable publique UPDATESEQ.

**CREATE SEQUENCE** updatesequence **INCREMENT BY 1 MAXVALUE 500 CYCLE**;

**CREATE PACKAGE** seqpackage **AS**

Updateseq **NUMBER;**

**END** seqpackage;

**CREATE or REPLACE PACKAGE BODY** seqpackage **AS END** seqpackage;  
--- Ajoutez un nouvel attribut à la table PropertyForRent pour marquer les lignes modifiées.

**ALTER TABLE** PropertyForRent **ADD** updated **NUMBER**;  
--- Avant de mettre à jour la table Staff à l’aide de ce déclencheur, créez un nouveau

--- Numéro de séquence et affectez-le à la variable publique UPDATESEQ.

**CREATE TRIGGER** Cascade\_STaffNO\_Update1

**BEFORE UPDATE** staffNo ON Staff

**DECLARE**

Dummy **NUMBER**;

**BEGIN**

**SELECT** updatesequence.NEXTVAL

**INTO** dummy **FROM** dual;

Seqpackage.updateseq := dummy;

**END;**  
--- créer une suite de ligne qui déclenche la mise à jour de la table PropertyForRent.  
--- Mettez la mise à jour en cascade uniquement si la ligne enfant n'a pas déjà été mise à jour par le déclencheur.

**CREATE TRIGGER** Cascade\_StaffNo\_Update2

**AFTER UPDATE OF** staffNO **ON** Staff

**FOR EACH ROW**

**BEGIN**

**UPDATE** PropertyForREnt **SET** staffNo = :new.staffNo,

Updateid = seqpackage.updateseq

**WHERE** staffNo = :old.staffNo **AND** updateid **IS NULL**;

**END;**  
--- Créer une déclaration finale après déclenchement pour réinitialiser les indicateurs updatedid

**CREATE TRIGGER** Cascade\_StaffNo\_Update3

**AFTER UPDATE OF** staffNo **On** Staff

**BEGIN**

**UPDATE** PropertyForRent **Set** updateid = **NULL**

**WHERE** updateid = seqpackage.updateseq;

**END;**

Figure 8.5 (b) Déclencheurs du tableau des effectifs.

Déclencheur 2 (Cascade\_StaffNo\_Update 1)

Ce déclencheur (au niveau de l'instruction) est déclenché avant la mise à jour de la colonne staffNo dans la table Staff afin de définir un nouveau numéro de séquence pour la mise à jour.

Déclencheur 3 (Cascade\_StaffNo\_Update2)

Ce déclencheur (au niveau de la ligne) se déclenche pour mettre à jour toutes les lignes de la table PropertyForRent contenant l'ancienne valeur staffNo (: old.staffNo) vers la nouvelle valeur (: new.staffNo) et pour marquer la ligne comme ayant été mise à jour.

Déclencheur 4 ( Cascade\_StaffNo\_Update3)

Le déclencheur final (au niveau de l'instruction) se déclenche après la mise à jour pour réinitialiser les lignes marquées à unflagged

Supprimer les déclencheurs

Les déclencheurs peuvent être supprimés à l'aide de l'instruction DROP TRIGGER <TriggerNAME>.

TRIGGER Privilège

Pour créer un déclencheur sur une table, l'utilisateur doit être le propriétaire de la table (dans ce cas l'utilisateur héritera du privilège TRIGGER) ou l'utilisateur devra avoir reçu le privilège TRIGGER sur la table (voir Section 7.6).

Avantages et inconvénients des déclencheurs

Les déclencheurs de base de données présentent un certain nombre d'avantages et d'inconvénients. Les avantages des déclencheurs incluent:

Élimination du code redondant. Au lieu de placer une copie de la fonctionnalité du déclencheur dans chaque application cliente qui en a besoin, le déclencheur n'est stocké qu'une seule fois dans la base de données.

Simplification des modifications: La modification d'un déclencheur nécessite de le modifier à un endroit unique: toutes les applications utilisent automatiquement le déclencheur mis à jour. Ainsi, ils ne sont codés qu'une fois, testés une fois, puis appliqués de manière centralisée pour toutes les applications accédant à la base de données. Les déclencheurs sont généralement contrôlés, ou au moins audités, par un administrateur de base de données qualifié. Le résultat est que les déclencheurs peuvent être mis en œuvre efficacement.

Sécurité accrue: le stockage des déclencheurs dans la base de données leur confère tous les avantages de la sécurité fournie automatiquement par le SGBD.

Intégrité améliorée: les déclencheurs peuvent être extrêmement utiles pour la mise en œuvre de certains types de contraintes d'intégrité, comme nous l'avons démontré précédemment. Le stockage de tels déclencheurs dans la base de données signifie que le SGBD peut appliquer de manière cohérente des contraintes d'intégrité dans toutes les applications.

Amélioration de la puissance de traitement. Les déclencheurs ajoutent une puissance de traitement au SGBD et à la base de données dans son ensemble.

Bon ajustement avec l'architecture client-serveur. L'activation et le traitement centralisés des déclencheurs conviennent parfaitement à l'architecture client-serveur (voir le chapitre 3). Une seule demande émanant d'un client peut entraîner l'exécution automatique d'une séquence complète de vérifications et d'opérations ultérieures par le serveur de base de données. De cette manière, les performances sont potentiellement améliorées car les données et les opérations ne sont pas transférées sur le réseau entre le client et le serveur.

Les déclencheurs ont aussi des inconvénients, notamment:

Frais généraux de performance. La gestion et l'exécution des déclencheurs ont un surcoût de performance qui doit être mis en balance avec l'avantage cité précédemment.

Effets en cascade. L'action d'un déclencheur peut provoquer le déclenchement d'un autre déclencheur, et ainsi de suite, en cascade. Cela peut non seulement entraîner une modification importante de la base de données, mais il peut également être difficile de prévoir cet effet lors de la conception du déclencheur.

Ne peut pas être programmé sous horaire; ils se produisent lorsque le l’événement sur lequel ils sont basés se produit.

Moins portable. Bien qu'ils soient maintenant couverts par le standard SQL, la plupart des SGBD implémentent leur propre dialecte pour les déclencheurs, ce qui affecte la portabilité.

# 8.4 Recursivité

L’Atomicité des données signifie que les groupes répétitifs ne sont pas autorisés dans le modèle relationnel. En conséquence, il est extrêmement difficile de traiter les requêtes récursives, c'est-à-dire les requêtes sur les relations qu'une relation a avec elle-même (directement ou indirectement). Pour illustrer cette nouvelle opération, nous utilisons l’exemple simplifié Relation de personnel présenté à la figure 9.16 du chapitre suivant, qui stocke les numéros d’effectif et le numéro d’effectif du responsable correspondant. Pour trouver tous les responsables de tout le personnel, nous pouvons utiliser la requête récursive suivante en SQL: 2008:

**WITH RECURSIVE**

AllManagers (staffNO, managerSTaffNO) **AS**

(**SELECT** staffNo, managerStaffNo

**FROM** Staff

**UNION**

**SELECT** in.staffNo, out.managerStaffNo

**FROM** AllManagers in, Staff out

**WHERE** in.managerStaffNo = out.staffNO);

**SELECT \* FROM** AllManagers

**ORDER BY** staffNo, managerStaffNo;

Cette requête crée une table de résultats AllManagers avec deux colonnes StaffNo et managerStaffNo contenant tous les gestionnaires de tous les employés. L'opération UNION est effectuée en prenant l'union de toutes les lignes produites par le bloc interne jusqu'à ce qu'aucune nouvelle ligne ne soit générée. Notez que si nous avions spécifié UNION ALL, toutes les valeurs en double resteraient dans la table de résultats.

Dans certaines situations, une application peut nécessiter que ces données soient insérées dans la table de résultats dans un certain ordre. La déclaration de récursivité permet de spécifier deux commandes:

Depth-first, où chaque élément « parent » ou «contenant» apparaît dans le résultat avant l'élément qu'il contient, ainsi que ses «frères ou sœurs» (éléments avec le même parent ou conteneur);

Breadth-first, où les articles suivent leurs "frères et sœurs" sans suivre leurs enfants.

Par exemple, au besoin de l'instruction WITH RECURSIVE, nous pourrions ajouter la clause suivante:

**SEARCH BREADTH FIRST BY** staffNo, manager StaffNo

**SET** orderColumn

La clause SET identifie un nouveau nom de colonne (orderColumn), utilisé par SQL pour classer le résultat dans la traversée largeur-avant requise.

Si les données peuvent être récursives, et pas seulement la structure de données, une boucle infinie peut se produire à moins que le cycle ne puisse être détecté. L'instruction récursive comporte une clause CYCLE qui indique à SQL d'enregistrer une valeur spécifiée pour indiquer qu'une nouvelle ligne a déjà été ajoutée à la table de résultats. Chaque fois qu'une nouvelle ligne est trouvée, SQL vérifie que la ligne n'a pas été ajoutée précédemment en déterminant si la ligne a été marquée avec la valeur spécifiée. Si tel est le cas, SQL suppose qu'un cycle a été rencontré et arrête la recherche d'autres lignes de résultat. Un exemple de clause CYCLE est:

**CYCLE** staffNo, managerStaffNo

**SET** cycleMark **TO** ‘Y’ **DEFAULT** ‘N’

**USING** cyclePath

**CycleMark et cyclePath** sont des noms de colonne définis par l'utilisateur que SQL doit utiliser en interne.

cyclePath est un ARRAY avec une cardinalité suffisamment grande pour prendre en charge le nombre de lignes du résultat et dont le type d'élément est un type de ligne avec une colonne pour chaque colonne de la liste des colonnes du cycle (staffNo et un manager StaffNo dans notre exemple).

Les lignes satisfaisant la requête sont mises en cache dans cyclePath. Lorsqu'une ligne satisfaisant la requête est trouvée pour la première (ce qui peut être déterminé par son absence de cyclePath), la valeur de la colonne cycleMark est définie sur "N". Lorsque la même ligne est à nouveau trouvée (ce qui peut être déterminé par sa présence dans cyclePath), la colonne cycleMark de la ligne existante de la table de résultats est modifiée en une valeur cycleMark de «Y» pour indiquer que la ligne commence un cycle.

# **Résumé du chapitre**

Les versions initiales du langage SQL ne comportaient aucune construction de programmation; c'est-à-dire qu'il n'était pas complètement computationnel. Cependant, avec les versions les plus récentes de la norme, SQL est maintenant un langage de programmation complet avec des extensions connues sous le nom de SQL / PSM (modules persistants stockés).

SQL / PSM prend en charge la déclaration de variables et comporte une instruction d'affectation, un flux d'instructions de contrôle (IF-THEN-ELSE-END IF; Boucle LOOP-EXIT-WHIT-END; Boucle FOR-END; Boucle WHILE-END] et des exceptions.

Une instruction SELECT peut être utilisée si la requête renvoie une et une seule ligne. Pour traiter une requête pouvant renvoyer un nombre arbitraire de lignes (c'est-à-dire zéro une ou plusieurs lignes), SQL utilise des curseurs pour permettre l'accès aux lignes d'un résultat de requête une à la fois. En effet, les curseurs agissent comme un pointeur sur une ligne particulière du résultat de la requête. Le curseur peut être avancé par un pour accéder à la ligne suivante. Un curseur doit être déclaré et ouvert avant de pouvoir être utilisé, et il doit être fermé pour le désactiver lorsqu'il n'est plus nécessaire. Une fois le curseur ouvert, les lignes du résultat de la requête peuvent être extraites une à une à l'aide d'une instruction FETCH, par opposition à une instruction SELECT.

Les sous-programmes sont des blocs PL / SQL nommés qui peuvent prendre des paramètres et être appelés. PL / SQL a deux types de sous-programmes appelés procédures et fonctions (stockées). Les procédures et les fonctions peuvent prendre un ensemble de paramètres qui leur sont attribués par le programme appelant et effectuer un ensemble d’actions. Les deux peuvent modifier et renvoyer les données qui leur sont transmises en tant que paramètre. La différence entre une procédure et une fonction est qu'une fonction retournera toujours une valeur unique à l'appelant, alors qu'une procédure ne le fera pas. Habituellement, les procédures sont utilisées sauf si une seule valeur de retour est requise.

Un déclencheur définit une action que la base de données doit entreprendre lorsqu'un événement survient dans l'application. Un déclencheur peut être utilisé pour appliquer certaines contraintes d'intégrité référentielle, pour appliquer des contraintes d'intégrité complexes ou pour auditer les modifications apportées aux données. Les déclencheurs sont basés sur le modèle Event-Condition-Action (ECA): l'évent (ou les évents) qui déclenchent la règle, la condition qui détermine si l'action doit être exécutée et l'action à exécuter.

Les avantages du déclencheur sont les suivants : élimine le code redondant, simplifie les modifications, augmente la sécurité, améliore l'intégrité, améliore la puissance de traitement et s'intègre parfaitement à l'architecture client-serveur. Les inconvénients des déclencheurs sont les suivants : surcharge de performances, effets en cascade, impossibilité d’être programmé et moins transférable.

# **Review Question**

## **8.1 Expliquer le terme “impedance mismatch” Décrivez brièvement comment SQL surmonte maintenant l’impedance mismatch.**

*GD ma réponse : (pas nécessairement parfaite)*

*L’impedance mismatch est le terme utilisé pour se référer aux problèmes causés par les différences entre le modèle de base de données et le langage de programmation.*

*Certains problèmes sont dû aux typages, le langage de programmation n’a pas nécessairement les mêmes types de données que ceux de la base de données.*

*Dans le modèle relationnel, chaque action renvoi toujours une relation qui sont plusieurs tuples, au niveau de la programmation il n’est pas possible de faire ce genre de chose car les données sont tous différentes.*

*Le PL/SQL d’Oracle a réduit ce problème en créant les curseurs, et en utilisant le même modèle de données and les mêmes types de données.*

## **8.2 Décrire la structure générale d'un bloc PL / SQL.**

*La structure d’un Bloc PL/SQL est composé de 3 blocs :*

*La première est DECLARE, elle est optionnelle et permet de déclarer des variables, routine, procédure trigger etc.*

*La deuxième est le BEGIN, elle est obligatoire, c’est dans ce bloc qu’il y a exécution.*

*Le Troisième est le bloc EXCEPTION il est optionnel.*

*Tout cette formule doit toujours finir avec un END;*

## **8.3 Décrire les instructions de contrôle en PL / SQL. Donnez des exemples pour illustrer vos réponses.**

*Les instructions de contrôle en PL / SQL sont :*

*Les déclaration conditionnel IF :*

*IF (position = ‘manager’) THEN*

*salary := \*1.05;*

*ELSE*

*salary := \*1.03;*

*END IF;*

*Les déclaration conditionnel CASE*

*CASE lowercase(x)*

*WHEN ‘a’ THEN x := 1;*

*WHEN ‘b’ THEN x :=2;*

*y := 0;*

*WHEN ‘default’ THEN x := 3;*

*END CASE;*

*Les déclaration itérative LOOP*

*x := 1;*

*myLoop :*

*LOOP*

*x := x+1;*

*IF (x > 3) THEN*

*EXIT myLoop;*

*END LOOP myLoop;*

*y := 2;*

*Les déclarations d’itération WHILE and REPEAT*

*DECLARE*

*numberOfStaff NUMBER;*

*SELECT COUNT(\*) INTO numberOfStaff FROM PropertyForRent*

*WHERE staffNo = ‘SG14’; myLoop :*

*FOR iStaff IN 1 … numberOfStaff LOOP*

*….*

*END LOOP*

*myLoop1;*

## 8.4 Decrivez les differences des declarations PL/SQL versus le Standard SQL.

***Les declarations de types: %TYPE et %ROWTYPE ne font pas partie du SQL Standard seulement du PL/SQL***

***Assignation de variables:***

***Dans le SQL standard on assigne une variable avec SET et “=”***

***SET vStaffNo = ‘SG14’ (SQL standard)***

***Contrairement au PL/SQL ou l’ont utilise :=***

***vStaffNo := ‘SG14’;***

***Flux conditionnel:***

***Le standard SQL spécifie ELSEIF au lieu de ELSIF***

***Boucle Loop:***

***le standard SQL spécifie LEAVE au lieu de EXIT WHEN (condition)***

***PL/SQL:***

**[labelNAME:]**

**LOOP**

**<SQL Statement list>**

**EXIT [labelNAME] [WHEN(condition)] gd :(remplace par LEAVE en SQL standard)**

**END LOOP [labelNAME]**

***Boucle While:***

***La difference reside dans le fait que le standard SQL possède des declarations REPEAT UNTIL et END REPEAT ce que le PL/SQL ne possède pas.***

**PL/SQL**  **SQL Standard**

**WHILE (condition) LOOP** **WHILE (condition) DO**

**<SQL statement list>** **<SQL statement list>**

**END LOOP;** **END WHILE [labelName];**

**REPEAT**

**<SQL statement list>**

**UNTIL (condition)**

**END REPEAT [labelNAME]**

***For LOOP:***

***Dans le PL/SQL on doit declarer une variable V\_NUMBER***

***Et ensuite utiliser un iterateur suivi du IN V\_NUMBER LOOP***

***Tandis que dans le SQL Standard Il faut designer un Loop suivi de l’iterateur AS select count(\*)***

**PL/SQL:**

**DECLARE**

**numberOfStaff NUMBER;**

**SELECT COUNT(\*) INTO numberOfStaff FROM PropertyForRent**

**WHERE staffNO = ‘SG14’; myLoop1:**

**FOR iStaff IN 1 .. numberOfStaff LOOP**

**….**

**END LOOP**

**myLoop1;**

**SQL Standard:**

**myLoop1:**

**FOR iSTaff AS SELECT COUNT(\*) FROM PropertyForRent**

**WHERE staffNo = ‘SG14’ DO**

**….**

**END FOR myLoop1;**

**Dans les déclencheurs:**

**le standard SQL utilise NEW ROW et OLD ROW tandis que PL/SQL utilise NEW et OLD.**

***PL/SQL:***

**VALUES (:new.staffNo, :new.fName, :new.lName, :new.position, :new.sex, :new.DOB, :new.salary, :new.branchNo);**

## **8.5 C’est quoi un curseur SQL à quoi ça sert? Donner un exemple.**

**Un curseur est une forme de pointeur qui permet de naviguer parmis les différente lignes d’une table.**

**Pour créer un curseur il faut le declarer en premier en sélectionnant les attributs et les lignes que nous voulons pointer. Puis faire une boucle pour itérer à l’intérieur de ce curseur.**

**declare**

**curseur** **PROPRIETE\_CURSEUR** **is**

**select** **NUM\_PROPRIETE, RUE\_PRO, VILLE\_PRO, NB\_PIECES\_PRO**

**from** **PROPRIETE\_A\_LOUER**

**where** **NUM\_PERSONNEL** **=** **‘EP21’**

**order by** **NUM\_PROPRIETE** **asc**;

**begin**

**for** **ENR\_PROPRIETE** **in** **PROPRIETE\_CURSEUR**

**loop**

**dbms\_output.put\_line(‘No de propriété : ‘|| ENR\_PROPRIETE.NUM\_PROPRIETE);**

**dbms\_output.put\_line(‘Rue : ‘|| ENR\_PROPRIETE.RUE\_PRO);**

**dbms\_output.put\_line(‘Ville : ‘|| ENR\_PROPRIETE.VILLE\_PRO);**

**if** **ENR\_PROPRIETE.NB\_PIECES\_PRO** **is not null then**

**dbms\_output.put\_line(‘nb pièces : ‘|| ENR\_PROPRIETE.NB\_PIECES\_PRO);**

**else**

**dbms\_output.put\_line(‘nb pièces : ‘|| ‘non spécifié’);**

**end if;**

**end loop;**

## **8.6 Qu’est-ce qu’un déclencheur SQL et de qu’elles façons peuvent-ils être utilisés?**

**Un déclencheur/Trigger/gachette est un sequence de code déclenchée par un évènement, on parle d’évènement before update/insert/delete ou after update/insert/delete**

**Les déclencheurs peuvent être utilisé pour rajouter des Nouvelles règles de gestion qui n’ont pas été définit.**

**Ils peuvent maintenir des règles de sécurité.**

**Générer des valeurs des champs calculés**

**Prévenir des accès incorrect.**

~~8.7: Discuss the differences between BEFORE, AFTER, and INSTEAD OF triggers. Give examples to illustrate your answers.~~  (Pas vu en cours)

## **8.8 Discuter des differences entre la déclaration FOR EACH ROW et FOR EACH STATEMENT, donner des exemple pour illustrer vos reponses.**

***La clause FOR EACH ROW va executer la sequence de code autant de fois qu’ils y a des enregistrements/tuples/lignes, tandis que FOR EACH STATEMENT est complètement independent du nombre de tuples: il s’executera qu’une seule fois pour toutes les tables.***

***PS: Attention aux erreurs de mutations: ils sont en lien avec les FOR EACH ROW (p-e revoir ça)***

## **8.9 Discuter des avantages et désavantages des déclencheurs/triggers.**

***Avantages:***

Élimination du code redondant. Au lieu de répéter des fonctionnalités dans chaque application ont met le code qu’une seule fois dans la base de données.

Simplification des modifications: Puisque on a besoin de modifier le déclencheur à un seul endroit unique il est beaucoup plus simple de mettre tous les applications à jour de façon centralisée

Sécurité accrue: Les déclencheurs dans la base de données permet de sécuriser mieux les données car en étant dans le SGBD les déclencheurs sont intimement liés à la BD.

Intégrité améliorée: Les déclencheurs permettent d’améliorer l’intégrité des données en y ajoutant d’autres contraintes d’intégrité que les applications devront respecter.

Amélioration de la puissance de traitement. Les déclencheurs ajoutent une puissance de traitement au SGBD et à la base de données dans son ensemble.

Bon ajustement avec l'architecture client-serveur.

Désavantages :

**Frais généraux de performance : Les exécution des déclencheurs amène des couts en performance**

**Effets en cascade. L'action d'un déclencheur peut provoquer le déclenchement d'un autre déclencheur, et ainsi de suite, en cascade. Cela peut non seulement entraîner une modification importante de la base de données, mais il peut également être difficile de prévoir cet effet lors de la conception du déclencheur.**

**Ne peut pas être activé de façon momentanée; ils se produisent lorsque l’événement sur lequel il doit réagir se produit.**

**Moins portable. Bien qu'ils soient maintenant couverts par le standard SQL, la plupart des SGBD implémentent leur propre dialecte pour les déclencheurs, ce qui affecte la portabilité.**

Exercices

For the following questions, use the Hotel shcema from the exercices at the end of Chapter 4

8.10 Create a stored procedure for each of the queries specified in Exercises 6.7 -6.11

*List full details of all hotels.*

**DECLARE**

**vHotelNo** **Hotel.hotelNo%TYPE**;

**vHotelName** **Hotel.hotelName%TYPE**;

**vCity**  **Hotel.city%TYPE**;

**CURSOR** **hotelCursor** **IS**

**SELECT \* FROM** **Hotel;**

**BEGIN**

**OPEN** **hotelCursor;**

**LOOP**

**FETCH** **hotelCursor**

**INTO** **vHotelNo, vHotelName, vCity;**

**EXIT WHEN** **hotelCursor%NOTFOUND**;

**dbms\_output.put\_line**(‘**Hotel Number: ’ || vHotelNo**);

**dbms\_output.put\_line**(‘**Hotel Name: ’ || vHotelName**);

**dbms\_output.put\_line**(‘**City: ’ || vCity**);

**END LOOP**

**IF** **hotelCursor%ISOPEN THEN CLOSE** **hotelCursor** **END IF;**

**EXCEPTION**

**WHEN OTHERS THEN**

**dbms\_output.put\_line(‘Error detected’**);

**IF** **hotelCursor%ISOPEN THEN CLOSE** **hotelCursor;** **END IF;**

**END;**

8.11 Create a database trigger for the folllowing situations:

(a) The price of all double rooms must be greater than 100$.

(b) The price of double rooms must be greater than the price of the highest single room.

(c) A booking cannot be for a hotel room that is relady booked ofr any of the specified dates.

(d) A guest cannot make two bookings with overlapping dates.

(e) Maintain an audit table with the names and adreses of all guest who make bookings for hotels in London (do not store duplicates guest details).

~~8.12 Create an INSTEAD OF database trigger that will allow data to be inserted into the following view:~~

~~CREATE VIEW LondonHotelRoom AS~~

~~SELECT h.hotelNO, hotelName, city, roomNO, type, price~~

~~FROM hotel h, Room r~~

~~WHERE h.hotelNo = r.hotelNo AND city = ‘London’~~

~~8.13 Analyse the RDBMS that you are currently using and determine the support the sytem provides for SQL programming constructs, database triggers, and recursive queries. Document the difference between each system and the SQL standard.~~