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 déclencheur pour réenforcer 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 complet au niveau de la programmation. 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 les 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’elle soit reférencés par d’autres instructions, en incluant les instructions de déclarations. Exemple de déclarations :

*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:

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 merchanisme 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> ]

[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 (seachCondition)

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)] (remplace par LEAVE)

END LOOP [labelNAME]

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

Par Exemple :

x := 1;

myLOOP :

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). 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 .. numberOf Staff 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 (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

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 FTECH, 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 FETACH 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: Evalue à TRUE si l'extraction la plus récente retourne une ligne (complément de% NOTFOUND)

% ISOPEN: Evalue 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’)***

***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 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;

(a revoir j’ai du mal avec celui là)

# 8.2 Subprograms, Stored Procedure, Functions and Packages

Subprogram are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprogram called (stored) procedures and functions. Procedures and fonctions can take a set of parameters given to them by the calling program and perform a set of actions. Both can modify and return data passed to them as a parameter. The difference between a procedure and a function is that a function will always return a single value to the caller, whereas a procedure does not. Usually, procedures are used unless only one return value is needed.

Procedures and functions are very similar to those found in most high-level programming languages, and have the same advantages: they provide modularity and extensibility, they promote reusability and maintenability, and they aid abstraction. A paraemter has a specified name and data type but can also be designated as:

IN – parameter is used as an input value only.

OUT – parameter is used as an output value only.

IN OUT – parameter is used as both an input and an output value

For example, we could change the anonymous PL/SQL block given in FIGURE 8.3 into a procedure by adding the following lines at the start:

CREATE OR REPLACE PROCEDURE PropertiesForStaff

(IN vStaffNo VARCHAR2)

AS . . .

The procedure could then be executed in SQL\*Plus as:

SQL > SET SERVEROUTPUT ON;

SQL > EXECUTE PropertiesForStaff(‘SG14’);

We discuss functions and procedures in more detail in Chapter 9.

Packages (PL/SQL)

A package is a collection of procedures, fucntions, variables, and SQL statements that are grouped together and stored as a single program unit. A package has two parts: a specification and a body. A package’s specification declares all public constructs of the package, and the body defines all constructs (public and private) of the package, and so implemetns the specification. In this way, packages provide a form of encapsulation. Oracle performs the following steps when a procedure or package is created:

It compiles the procedure or package.

It stores the compiled code in memory.

It stores the procedure or package in the database.

For the previous example, we could create a package specification as follows:

CREATE OR REPLACE PACKAGE StaffPropertiesPackage AS

Procedure PropertiesForStaff(vStaffNo VARCHAR2);

END StaffPropertiesPackage;

And we could create the package body (that is, the implemetnation of the package as:

CREATE OR REPLACE PACKAGE BODY StaffPropertiesPackage

AS

….

END StaffPropertiesPackage;

To reference the items declared within a package specification, we use the dot notation. For example, we could call the PropertiesForStaff procedure as follows:

StaffPropertiesPackage.PropertiesForStaff(‘SG14’);

# 8.3 Déclencheurs

A Trigger defines an action that the datase should take when some event occurs in the application. Atrigger ma ybe used to enforce some referentiel integrity constraints, to enforce complex constraints, or to audit changes to data. The general format of a trigger in SQL is:

CREATE TRIGGER TriggerName

BEFORE | AFTER| INSTEAD OF

INSERT | DELETE | UPATE [ OF TriggerColumnList ]

ON TableNAme

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

[FOR EACH {ROW | STATEMENT ) ]

[WHEN Condition]

< trigger action >

This is not the complete defintion, but it sufficient to demonstrate the basic concept. The code within a trigger, called the trigger body or trigger action, is made up of an SQL block. Triggers are based on the Event-Condition-Action (ECA) model:

The event (or events) that trigger the rule, which can be an INSERT, UPDATE, or DELETE statement on a specified table (orpossibly view). In Oracle, it can also be:

A CREATE, ALTER, or DROp statement on any schema object;

A database startup or instance shutdown, or a user logon or logoff;

A psecific error message or any error message

It is also possible to specify whether the trigger should fire before the event or after the event

The condition that dtermines wehter the action should be executed. The condition is optional but, if specifed, the action will be executed only if the condition is true.

The action to be taken. This block contains the SQL statemetns and code to be executed when a triggering staement is issued and the trigger condition evaluate to true.

There are two type of trigger: row-level triggers (FOR EACH ROW) that execute for each row of the table that is affected by the triggering event, and statement-level trigger (FOR EACH STATEMENT) that execute only once event if multiples rows are affected by the triggering event. SQL also syupports INTEAD OF triggers, which provide a transparent way of modifying views that cannot be modifed directly through SQL DML statements (INSERT’ UPDATE, and DELETE). These triggers are called INTEAD OF triggers because, unlike other types of trigger, the trigger is fire intead of executing the original SQL statement. Triggers can als oactivate themselves on after the other. This can happen when the trigger action makes a change to the database that has the effect of causing another vent that has a trigger associated with it to fire

EXAMPLe 8.2 AFTER Row-level trigger

Create an AFTER row-level trigger to keep an audit trail of all rows inserted into the Staff table

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;

Note that the SQL standard uses NEW ROW instead of NEW and OLD ROW instead of OLD.

EXAMPLE 8.3 Using a BEFORE trigger

DreamHome has a rule that pevents a member of staff from managing more than 100 properties at the same time. We could create the trigger shown in Figure 8.4 to enforce this constraint. This trigger is invoked before a row is inseted into the PropertyForRent

CREATE TRIGGER STaffNotHandlingTooMUCH

BEFORE INSRET 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 Trigger to enforce the constraint that a member of staff cannot manage more than 100 properties at any one time.

Table or existing row is updated. If the member of staff currently manages 100 properties, the system displays a message and aborts the transaction. The following points should be noted

The BEFORE keyword indicates that the trigger should be executed before an insert is applied to the PropertyForRent table.

The FOR EACH ROW keyword indicates that this is a row-level trigger, which executes for each row of the PropertyForRent table that is updated in the statement.

EXAMPLE 8.4 Using triggers to enforce referential integrity

By default, Oracle enforces the referentiel actions ON DELETE NO ACTION and ON UPDATE NO ACTION on the named foreign keys (see Section 7.2.4). It also allows the additional clause ON DELETE CASCADE to be specified to allow deletions from the parent table to cascade to the child table. However, it does not support the ON UPDATE CASCADE action, or the SET DEFAULT and SET NULL actions. If any of these actions are required, they will have to be implemented as triggers or stored procedures, or within the application code. For example, from Example 7.1 the foreigh key staffNo in the PropertyForREnt table should have the action ON UPDATE CASCADE. This action can be implemented using the triggers shown in Figure 8.5

Trigger I (PropertyForRent\_Check\_Before)

The trigger in Figure 8.5(a) is fired whenever the staffNo column in the PropertyForREnt table is updated. The trigger check before the update takes place whether the new value specified exists in the Staff table. If an Invalid\_Staff exception is raised, the trigger issues an error message and prevents the change from occuring.

--- Before the staffNo column is updated in the PropertyForRent table, fire this trigger

--- to verify that the new foreign key valie is present in the Staff table.

CREATE TRIGGER PropertyForENT\_CHECK\_Before

BEFORE UPATE 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 (metating\_table, -4091);

--- Use cursosr to verify that parent key valie exists.

--- Use FOR UPDATE OF to lock parent key’s row so that it cannot be deleted

--- by another transaction until this transaction completes

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

SELECT staffNO FROM Staff

WHERE staffNO = sn

FOR UPATE OF staffNo;

BEGIN

OPEN update\_cursor (:new.staffNo);

FETCH update\_cursor INTO dummy;

--- verify parent key. Raise exceptions as appropriate.

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;

--- Amutating table is table that is currently being modified by an INSERT, UPDATE,

--- or DELETE Statement, o one that might need to be updated by the effects of a declarative

--- DELETE CASCADE referential integrity constraint.

--- This error would raise an exception, but in this case the exception is OK, so trap it,

--- but don’t do anything

WHEN mutating\_table THEN

NULL;

END;

Figure 8.5(a) Oracle Triggers to enforce ON UPDATE CASCADE on the foreign key staffNo in the PropertyForRent table when the primary key staffNo is updated in the Staff table:

1. Trigger for the PropertyForRent table

Changes to support triggers on the Staff table

The three triggers shown in Figure 8.5(b) are fired whenever the staffNo column in the Staff table is updated. Before the definition of the triggers, a sequence number updateSequence is created, along with a public variable updateSeq ( which is acessible up the three triggers through the seqPackage package). In addition, the PropertyForRent table is modified to add a column called updateId, which is used to flag whether a row has been updated, to prevent it from being updated more than once during the cascade operation.

--- Create a sequence number and a public variable 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;

--- Add a new attribute to the PropertyForRent table to flag changed rows.

ALTER TABLE PropertyForRent ADD updated NUMBER;

--- Before updating the Staff table using this statement trigger, generate a new

--- sequence number and assign it to the public variable 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;

--- create a row after-trigger that cascades the update to the PropertyForRent table.

--- Only cascade the update if the child row has not already been updated by the trigger.

CREATE TRIGGER Cascade\_StaffNo\_Update2

AFTER UPATE 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;

--- Createa final statement after-trigger to reset the updatedid flags

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) Triggers for the Staff table.

Trigger 2 (Cascade\_StaffNo\_Update 1)

This (statement-level) trigger fires before the update to the staffNo column in the Staff table to set a new sequence number for the update.

Trigger 3 (Cascade\_StaffNo\_Update2)

This (row-level) trigger fires to update all rows in the PropertyForREnt table that have the old staffNo value (:old.staffNo) to the new value (:new.staffNo), and to flag the row as having been updated.

Trigger 4 ( Cascade\_StaffNo\_Update3)

The final (statement-level) trigger fires after the update to reset the flagged rows back to unflagged

Dropping triggers

Triggers can be dropped using the DROP TRIGGER <TriggerNAME> statement.

TRIGGER Privilege

In order to create a trigger on a table, the user either has to be the owner of the table (in which case the user will inherit the TRIGGER privilege) or the user will need to have been granted the TRIGGER privilege on the table (see Section 7.6).

Advantages and disadvantages of triggers

There are a number of advantages and disadvantages with database triggers. Advantage of triggers include:

Elimintation of redundant code. Instead of placing a copy of the fucntionality of the trigger in every client application that requires it, the trigger is stored only once in the database

Simplyfying modifications: Changing a trigger requires changing it in one place only: all the application automatically use the updated trigger. Thus, they are only coeded once, tested once, and then centrally enforced for all the applciations acessing the database. The triggers are usually controlled, or at least audited, by a skilled DBA. The result is that the triggers can be implemented efficiently.

Increased security: Storing the triggers in the database gives them all the benfits of security provied automatically by the DBMS

Improved integrity: Triggers can be extremely useful for implementing some types of integrity constraints, as we have demonstrated earlier. Storing such triggers in the database means that integrity constraints can be enforcved consistently by the DBMS across all applications

Improved processing power. Triggers add processing power to the DBMS and to the database as a whole.

Good fit with the client-server architecture. The central activation and processing of triggers fits the client-server architecture well (see Chapter 3). A single request from a client can result in the automatic performing of a whole sequence of checks and subsequent operations by the database server. In this way, performance is potentially improved as data and operations are not trnasferred across the network between the client and the server.

Triggers also have disadvantages, which include:

Performance overhead. The management and execution of triggers have a performance overhead that have to be balanced against the advantage cited previously.

Cascading effects. The action of one trigger can cause another trigger to be fired, and so on, in a cascading manner. Not only can this cause a signifiant change to the database, but it can also be hard to foresee this effect when designing the trigger.

Cannot be scheduled Triggers cannot be scheduled; they occur when the cent that they are based on happens

Less portable. Although now covered by the SQL standard, most DBMS implement their own dialect for triggers, which affects portability.

# 8.4 Recursion

Atomicity of data means that repeating groups are not allowed in the relational model. As a result, it is extremly difficult to handle recursive queries, that is, queries about relationships that a relation has with itself (directly or indirectly). To illustrate the new operation, we use the example simplified Staff relation shown in Figure 9.16, in the next chapter, which stores staff nubmers and the correpsding manager’s staff number. To find all the managers of all staff, we can use the following recursive query in 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;

This query creates a result table AllManagers with two columns taffNo and managerStaffNo containing al lthe mangers of all staff. The UNION operation is performed by taking the union of all rows produced by the inner block until no new rows are generated. Note, if we had specified UNION ALL, any duplicate values would remain in the result table.

In some situations, an applciation may require that data to be inserted into the result table in a certain order. The recusion statement allows the specifications of two orderings:

Depth-first, where each ‘parent’ or ‘containgin’ item appears in the result before the item that it contains, as well as before its ‘siblings’ (items with the same parent or container);

Breadth-first, where items follow their ‘siblings’ without following the siblings’ children.

For exemple, at the ned of the WITH RECURSIVE statement we could add the following clause:

SEARCH BREADTH FIRST BY staffNo, manager STaffNo

SET orderColumn

The SET clause identifie a new column name (orderColumn), which is used by SQL to order the result into the required breadth-first traversal.

If the data can be recursive, not just the data structure, an infite lopp can occur unless the cyle can be detected. The recursive statement has a CYCLE clause that instructs SQL to record a specified value to indicate that a new row has already been added to the result table. Whenever a new row is found, SQL check that the row has not been added previously by determining whether the row has been marked with the specified value. If it has, then SQL assumes a cycle has been ecountered and stops searching for further result rows. An examples of the CYCLE clause is:

CYCLE staffNo, managerStaffNo

SET cycleMark TO ‘Y’ DEFAULT ‘N’

USING cyclePAth

CycleMark and cyclePath are user-defined column names for SQL to use internally.

cyclePath is an ARRAY with cardinality sufficiently large to accommodate the number of rows in the result and howse element type is a row type with a column for each column in the cycle column list (staffNo and amanger STaffNo in our example).

Rows satisfying the query are cached in cyclePath. When a row satysfying the query is found for the first (which can be determined by its absence from cyclePath), the value of the cycleMark column is set to ‘N’. When the same row is found again (which can be determined by its presence in cyclePath), the cyucleMark column of the existing row in the result table is modified to the cycleMark value of ‘Y’ to indicate that the row starts a cycle.

Chapter Summary

The initials versions of the SQL language had no programming constructs; that is, it was not computationnaly complete. However, with the more recent versions of the standard, SQL is now a full programming language with extensions known as SQL/PSM (Persistent Stored Modules)

SQL/PSM supports the declaration of variables and has assigment statement, flow of control statemetns (IF-THEN-ELSE-END IF; LOOP-EXIT WHEN-END LOOP; FOR-END LOOP; WHILE-END LOOP), and exceptions.

A SELECT statemetn can be used if the query returns one and only one row. To handle a query that can return an arbritrary number of rows (that is, zero one, or more rows), SQL uses cursors to allow the rows of a query result to be acessed one at a time. In effect, the cursors acts as a pointer to a particular row of the query result. The cursor can be advanced by one to acess the next row. A cursor must be declared and opened before it can be used, and it must be closed to deactivate it after it is no longer required. Once the curor has been opened, the rows of the query result can be retrived one at a time using a FETCH statement, as opposed to a SELECT statement.

Subprogram are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two type of subprograms called (stored) procedures and functions. Procedures and functions can take a set of parameters given to them by the calling program and perform a set of actions. Both can modify and return data passed to them as a parameter. The difference between a procedure and a function is that a function will always return a single value to the caller, whereas a proceudre will not. Usually, procedures are used unless only one return value is needed.

A trigger defines an action that the database should take when some event occurs in the application, a trigger may be used to enforce some referential integrity constraints, to enforce complex integrity constraints, or to audit changes to data. Triggers are based on the Event-Condition-Action (ECA) model: the vent (or vents) that trigger the rule, the condition that determines whether the action should be executed, and the action to be taken.

Advantages of trigger include: eliminates redundant code, simplifie modifications ,increase security, improves integrity, improves processing power, and fits well with the client-server architecture. Disadvantages of triggers include: performance overhead, cascading effects, inability to be scheduled, and less portable.

Review Question

8.1 Explain the term “impedance mismatch” Briefly describe how SQL now overomes the impedance mismath.

8.2 Describe the general structure of a PL/SQL block.

8.3 Describe the control statemetns in PL/SQL Give eamples to illustrates your answers.

8.4 Describes how the PL/SQL statements differ from the SQL standard. Give examples to illstruates your answers

8.5 What are SQL cursors? Give an example of the use of an SQL cursor.

8.6 What are database triggers and what could they be used for?

8.7: Discuess the differences between BEFORE, AFTER, and INSTEAD OF triggers. Give examples to illustrate your answers.

8.8 Discuss the differences between row-level and statement-level triggers Give examples to illsutrate your answers.

8.9 Discuss the advantages and disadvantages of database triggers.

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

8.11 Create a database trigger for the folllowing situations:

(a) The price of all double rooms msut 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 INSTAD OF database trigger that will allow data to be inserted into the following view:

CREATE VIW 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.